Universidad Nacional de La Plata
Facultad de Informática

Tesis de Licenciatura en Informática

Funcionalidad Volátil en Juegos Web: Una Arquitectura de Soporte

por

Leandro Manuel Pérez

Director: Dr. Gustavo Rossi

La Plata, 2009

Dedicado a mis viejos Adolfo y Miriam, mi hermano Herny, mi abuela Tota y mi novia Romi.

Índice general

Agradecimientos

Quiero agradecer al Dr. Gustavo Rossi, por permitir desarrollar mi tesis bajo su tutoría, por sus aportes durante el desarrollo del trabajo y durante la escritura de este documento.

Agradezco al Dr. Federico Balaguer, por su ayuda, motivación y consejos, que elevaron la calidad de mi trabajo y me ayudaron a crecer profesionalmente.

Finalmente agradezco inmensamente a mi familia: mi papá y mi mamá, mi hermano, mi abuela y mi novia, que me aguantaron y ayudaron durante todos estos años, muchos y largos. Sin el apoyo incondicional de mis padres durante toda la carrera, su afecto constante, las palabras de aliento y sus consejos, no podría estar en el lugar que estoy. No hay palabras para agradecerles. Lo mismo va para mi hermano Herby, que siempre estuvo para soportar mi mal humor, me supo entender y me hizo reir. Por último, le doy las gracias a mi novia, Poni, que en los últimos años me apoyó incondicionalmente y siempre encontró la forma de ayudarme con alegría y amor, para avanzar y poder terminar de una vez con este trabajo.

Leandro Pérez.
Septiembre, 2009.

Capítulo 1

Introducción

Los sistemas web actuales están inmersos en un contexto altamente cambiante. La infraestructura que los soporta está en constante evolución (Internet, browsers, standards, protocolos, etc.) y la tecnología utilizada en los mismos es diversa, compleja y está en constante cambio. Esta evolución inevitablemente implica costos y tiempos de desarrollo más elevados. Estas cuestiones se contrastan con la dinámica de la web moderna, que demanda tiempos de desarrollo y evolución aún más rápidos [FF00]. Los clientes para los cuales se desarrollan los sistemas web, generalmente, poseen un alto grado de incertidumbre con respecto al entendimiento del problema y la consecuente comprensión de sus necesidades [LHs08, Bro87, LE02]. Los tiempos disponibles para el desarrollo suelen ser cortos, como consecuencia del valor de time-to-market requerido en este tipo de sistemas [LE02, Wik09e]. Como consecuencia de todo esto, surge la necesidad de una aproximación que permita el desarrollo de aplicaciones flexibles, que puedan evolucionar rápidamente, adaptándose a los cambios constantes en los requerimientos [LHs08].

En el contexto recién caracterizado, los cambios en los requerimientos aparecen durante todo el ciclo de vida del software [FY97] y dichos requerimientos tienden a ser muy volátiles [LE02]. Ciertos requerimientos necesitan la introducción de nueva funcionalidad, que solo será necesaria por un lapso de tiempo y luego tendrá que ser removida. A este tipo de funcionalidad, que por su naturaleza solo estará disponible por un período de tiempo, se la conoce como funcionalidad volátil [RNM+06].

Los sistemas exitosos son los que soportan el cambio [FY96, Bro87] y el software que no se puede adaptar a cambios en los requerimientos, perecerá [FY96]. Por lo tanto, es necesario poder responder a la aparición de nueva funcionalidad, en este caso, volátil, de manera exitosa. Para lograr esto, es necesario contar con una estructura de soporte. En esta tesis se propone una arquitectura de soporte

para funcionalidad volátil en aplicaciones web RIA.

1.1. Abstract

A medida que el software aumenta su complejidad, es más difícil de modificar. Esto es un obstáculo para la evolución y puede impedir la habilidad de manejar los cambios en los requerimientos. Los cambios en los requerimientos aparecen durante todo el ciclo de vida del software [FY97] y dichos requerimientos tienden a ser muy volátiles [LE02]. Ciertos requerimientos necesitan la introducción de nueva funcionalidad, que solo será necesaria por un lapso de tiempo y luego tendrá que ser removida. A este tipo de funcionalidad, que por su naturaleza solo estará disponible por un período de tiempo, se la conoce como funcionalidad volátil [RNM+06]

La incorporación de funcionalidad volátil de forma directa, reparando la aplicación existente a medida en que surgen nuevas necesidades, introduce errores y hace cada vez más caro y complicado el proceso de mantenimiento [RNM+06, FY97]. Este camino, conduce a una degradación del sistema que, potencialmente, puede desembocar en un "Big Ball of Mud"[FY97]. Se necesitan alternativas basadas en diseño para lograr una evolución exitosa [RNM+06, LHs08]. El continuo refactoring puede ser la solución que evite el camino a un sistema imposible de mantener[FY97].

En este trabajo se apoyan las ideas mencionadas en el párrafo anterior. Se busca, a través de la utilización de una arquitectura de soporte, minimizar el área de impacto de los cambios necesarios para incorporar y remover funcionalidad volátil, con el objetivo de obtener un proceso menos costoso y más rápido. El desarrollo de una arquitectura para el desarrollo de aplicaciones web que soporten funcionalidad volátil es el foco de esta tesis. Tal arquitectura tiene el objetivo de proveer los medios a través de los cuales, un sistema web pueda extenderse dinámicamente para soportar el cambio y evolucionar gradualmente[FY96]. Está pensada no solo para escribir y modificar las aplicaciones pensadas a partir de ella, sino también, para poder incorporar la funcionalidad volátil en aplicaciones existentes.

1.2. Estructura del resto del documento

- En el capítulo 2 se motiva el trabajo mediante la presentación en detalle del concepto de funcionalidad volátil. En este capítulo se presenta un análisis de los aspectos teóricos para brindar una introducción completa al tema de estudio del resto del documento.

- En el capítulo 3 Se introduce el contexto donde el tema de estudio fue evaluado: Juegos Web en Smalltalk (RIA), particularmente los juegos de

casino. Se presentan los prototipos desarrollados, a partir de los cuales fue surgiendo una arquitectura de soporte.

- En el capítulo 4 se presenta la arquitectura implementada para el desarrollo de aplicaciones web que soporten funcionalidad volátil. Se brinda un panorama general de la arquitectura introduciendo todos sus componentes en un nivel de granularidad alto.
- En el capítulo 5 se detallan los componentes de la arquitectura presentados en el capítulo anterior. Se brinda un análisis detallado de cada componente, explicando ventajas, desventajas y el rol del componente en la arquitectura.
- En el capítulo 6 se analizan algunos trabajos relacionados que fueron estudiados durante el desarrollo de la tesis.
- En el capítulo 7 se presentan conclusiones sobre el trabajo desarrollado. Se brinda un análisis sobre los resultados obtenidos, aportes y trabajo pendiente.
- En el apéndice A se incluye un análisis sobre el uso de Method Wrappers en esta tesis, como mecanismo para alterar comportamiento de manera no intrusiva.
- En el apéndice B se presentan las decisiones de diseño más importantes sobre los prototipos desarrollados. Se incluyen diagramas UML de los modelos de clases de los prototipos.

Capítulo 2

Funcionalidad Volátil

2.1. Introducción a la Funcionalidad volátil.

El concepto de funcionalidad volátil tiene relación directa con el alto dinamismo y la evolución constante de las aplicaciones web actuales. Factores económicos, tecnológicos, estratégicos y demás afectan constantemente el desarrollo y mantenimiento de los sistemas web. A menudo surgen nuevos requerimientos que deben ser plasmados en los programas mediante la implementación de nuevas funcionalidades. Los requerimientos que típicamente aparecen luego de que la aplicación web haya sido desarrollada y puesta en producción, y que son válidos por períodos cortos de tiempo, después de los cuales son descartados; son identificados como requerimientos volátiles.

A partir de la aparición de requisitos volátiles, surgne funcionalidades que solo necesitan ser instaladas en el sistema por un período de tiempo. Por ejemplo, en un sitio de ventas online, debido a una promoción, es necesario alterar la lógica de negocio por un tiempo. O en un periódico online debido a un evento internacional importante es necesario incluir funcionalidad asociada al seguimiento del evento. A estas funcionalidades, que por su naturaleza deben ser instaladas en un sistema para removerlas luego, se las conoce como funcionalidades o servicios volátiles [RNM+06, GDRU09]. La volatilidad yace justamente en el hecho que la funcionalidad se instala para ser removida en el futuro inmediato.

2.1.1. Ejemplos

En ocasiones, las funcionalidades volátiles se corresponden con nuevos requisitos que necesitan ser experimentados por un período de tiempo para analizar la reacción del usuario. Otras veces, son activadas puntualmente en respuesta a un evento específico o a un conjunto de condiciones. Más frecuentemente, son

periódicamente activadas y desactivadas en coincidencia con períodos específicos del año.

En un comercio web, como Amazon.com, ejemplos típicos de tales funcionalidades son las ofertas especiales disponibles en ciertos períodos del año (e.g. Navidad, San Valentín, etc.) sobre ciertos productos, la personalización de contenidos para nuevos lanzamientos, la funcionalidad para recaudar fondos luego de una catástrofe, etc. Ejemplos similares pueden ser encontrados en sitios de noticias, como CNN.com, para organizar discusiones sobre eventos inesperados, para incluir tipos de publicidades no previstos (ej. durante las elecciones presidenciales), etc.

Ejemplo concreto: Amazon + Visa card.

Amazon.com ofrece la oportunidad de obtener una tarjeta de crédito Visa que brinda ciertos beneficios al cliente. Luego de la obtención de la tarjeta de crédito Visa de Amazon, la lógica de checkout del usuario se modifica en varios sentidos: en primer lugar, el usuario tiene un descuento único de 30 dólares en la primer compra luego de la obtención de la tarjeta, luego con cada compra efectuada el usuario gana puntos para un programa de premios y descuentos. Este es un ejemplo de funcionalidad volátil que afecta al sitio en distintos niveles de profundidad. La lógica de negocio se altera durante el período que dura la promoción, otorgando descuentos y agregando un sistema de puntuación con premios que, antes no era parte del sistema original. Asi mismo, la navegabilidad del sitio y la interfaz de usuario son alteradas en multiples instancias: para hacer publicidad de la promoción, en el momento del checkout para informar de los puntos otorgados y los descuentos aplicados, etc. Seguramente esta funcionalidad apareció en Amazon como resultado de una alianza con Visa. Es probable que eventualmente dicha alianza se termine por algún motivo o que la estrategia de negocio de Amazon.com cambie, por lo que la funcionalidad será removida del sitio. Entonces, la lógica de negocio tendrá que ser alterada nuevamente, a si mismo la interfaz de usuario. No solo Amazon.com tiene este feature, BestBuy.com. Target.com y otros también lo han implementado.

En una aplicación web pueden haber diferentes tipos de requerimientos volatiles que den lugar a funcionalidad volatil. Algunos de ellos pueden surgir durante la evolución de la aplicación para probar la aceptación de los usuarios (ej. beta functionality) y luego pueden ser considerados, o no, servicios core de la aplicación (ej. user tags en Amazon.com) volviendose una adición permanente. Otro tipo de funcionalidad volatil solo está disponible por un corto y determinado período de tiempo, como la funcionalidad para aceptar donaciones luego de una catastrofe o ventas por un período fijo de tiempo (ej. navidad). Otros requerimientos son aún más irregulares, algunos tipos de descuentos de precio en tiendas electrónicas (ej. ventas de productos remanentes en stock de otras temporadas), sorteos

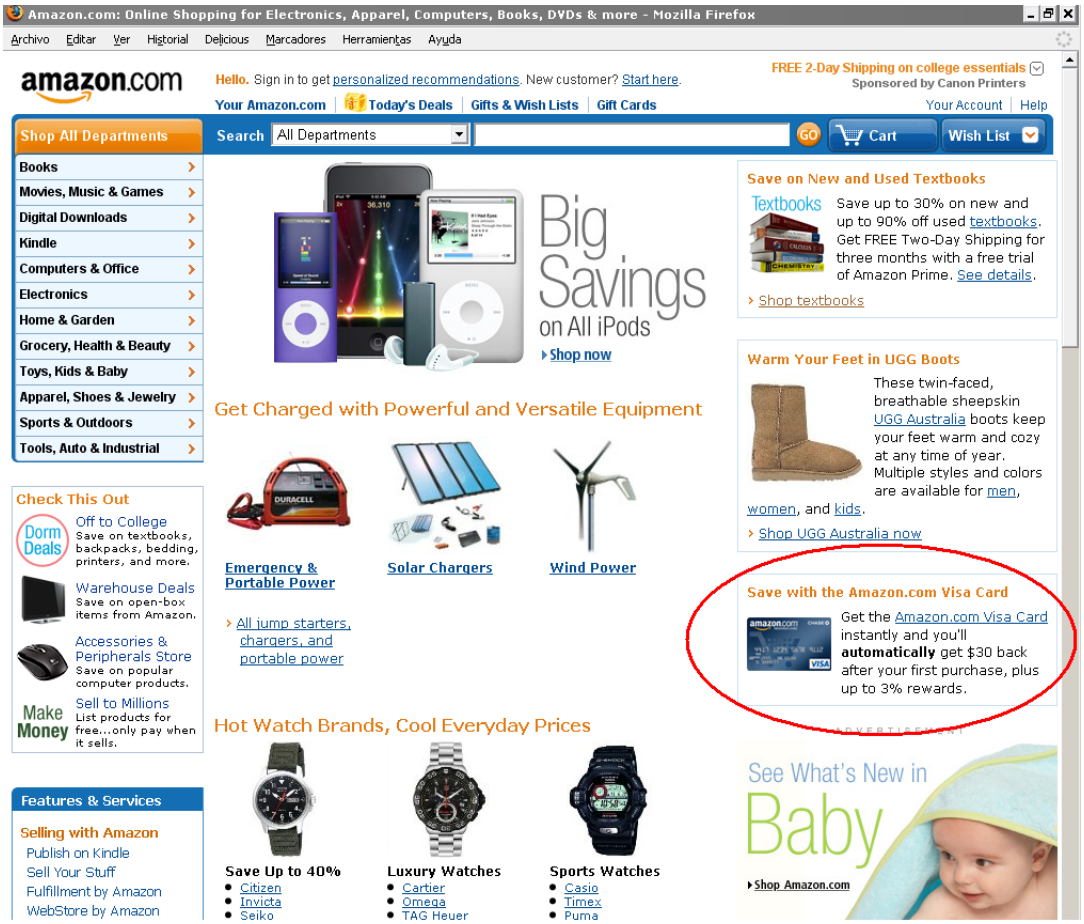

Figura 2.1: Ejemplo de Funcionalidad Volátil: Amazon Visa Card

y ventas de entradas para shows asociados con un artista o disco. En todos los casos nos encontramos con un serio problema: al ser esta funcionalidad volatil, quizás necesitemos desactivarla luego de cierto tiempo.

2.1.2. ¿Por qué funcionaldiad volátil?

En un entorno altamente cambiante y demandante, surgen a menudo requisitos que deben ser cumplidos y que probablemente luego tengan que ser revertidos. Este panorama es visible en la mayoría de las aplicaciones web actuales: "Una de las características pricipales de la mayoría de aplicaciones web es su alto dinamismo. Nuevas funcionalidades son implementadas y agregadas al sistema luego de que haya sido puesto en producción, a medida en que los requerimientos a parecen. Algunas de estas funcionalidades aparecen en la web como respuesta a un evento o fenómeno no esperado (como una catástrofe) luego del cuál tienen que ser removidas. Algunas otras son activadas periódicamente, concurrentemente con una fecha particular o período del año (como la vuelta al colegio, o Navi-

dad) "[RGDU08]. Las aplicaciones deben poder evolucionar y responder a estos cambios.

El manejo de requerimientos volátiles, a través de la incorporación de las funcionalidades correspondientes, es un problema que forma parte de una cuestión mayor: el problema de la evolución del software. El cambio y la evolución del software es un problema ampliamente estudiado. Las aplicaciones deben adaptarse rápidamente a las necesidades de negocio para poder subsistir [YJ02]. En [SvGB05] se indica que el costo de revertir una decisión de diseño es muy alto y en [SA08] se argumenta que la variabilidad, tratada por el diseñador del sistema es mejor controlada. Es por esto, que en dominios donde la aparición de funcionalidad volátil sea algo esperado, resulta provechoso atacar el problema en etapas tempranas (diseño) y de forma organizada (contando con una estructura que de soporte a posibles soluciones).

Entonces, estudiar funcionalidad volátil no es más que otra forma de atacar el problema mayor de la evolución continua y necesaria del software. Es necesario encontrar formas de diseñar e implementar modificaciones necesarias en los sistemas para minimizar costos y lograr aplicaciones que puedan evolucionar de forma adecuada y sean capaces de ser mantenidas con el paso del tiempo. El estudio recién mencionado y la consecuente incorporación de aproximaciones sistemáticas constituyen un posible camino para lograr esto.

2.2. Detalles relevantes de Funcionalidad Volátil

En esta sección se explicarán conceptos acerca de la funcionalidad volátil que servirán para el desarrollo del resto del trabajo.

2.2.1. Definición de Funcionalidad Volátil

En [RNM+06] el término volátil se aplica al tipo de funcionalidad que por su propia naturaleza está disponible solo por un período de tiempo. Luego en [RGDU08] el concepto de funcionalidad volátil se define como:

> *"Funcionalidades que son implementadas y activadas una vez, en conexión con un evento inesperado y luego son removidas definitivamente, o que son periódicamente activadas durante un período particular."*

2.2.2. Atributos

A continuación se explican algunas características marcadas en el trabajo [RGDU08] que serán útiles para el resto de este documento. En dicho trabajo de definieron los atributos Intent, Extent y Volatility Pattern, con el objetivo de caracterizar y entender mejor las funcionalidades volátiles. En este trabajo

se explica la necesidad de un nuevo atributo, llamado Connexion Pattern. Más adelante, en la sección 3.6.1, se presentará un pequeño análisis de estos atributos en un ejemplo concreto de side game, implementado como parte del desarrollo de esta tesis.

Intent

El atributo de funcionalidad volátil Intent describe los aspectos de la aplicación donde la funcionalidad volátil impacta. Para cada aspecto el intent define cuáles son los tipos de objetos de la aplicación que tienen que ser agregados y cómo. Los aspectos referenciados por el intent incluyen: Contenido, navegación, interfaz de usuario y comportamiento del sistema. Correspondientemente, los tipos de objetos de aplicación incluyen: tipos y vistas de contenido, nodos y paths navegacionales [Ros96, SR98], interfaces de usuario, artefactos de interacción con el usuario, operaciones del usuario, procesos de negocio y reglas de negocio.

Extent

Identifica el conjunto de objetos de aplicación (Instancias de los tipos identificados por el Intent) que son impactados por la funcionalidad volátil. Define el conjunto de instancias de los tipos de objetos mencionados en el Intent que deben ser agregadas o modificadas. Define si la funcionalidad volátil afecta a todas las instancias de un tipo de objeto o solo algunas en particular. Se especifica una forma de determinar las instancias involucradas por la funcionalidad volátil.

Volatility pattern

Describe el ciclo de vida de la funcionalidad volátil, o sea, las reglas que determinan los momentos de activación y deactivación de la misma. El patrón de volatilidad define reglas que establecen el momento en que se activa la funcionalidad volátil, cuanto dura activa y los patrones de repetición. Por ejemplo, cierta funcionalidad se activa en una fecha específica (el día anterior a la Navidad) y dura activa durante 2 días, repitiendo todos los días 24 de diciembre.

A partir de este atributo, los autores del trabajo en cuestión, explican el ciclo de vida de una funcionalidad volátil. Se distinguen 2 estados en el ciclo de vida, activo y pasivo. Inicialmente una funcionalidad volátil es introducida en el sistema en estado pasivo. Cuando un evento ocurre, como una fecha particular o un evento de la aplicación, la funcionalidad se pone en estado activo. En este estado, la funcionalidad volátil introduce sus nuevos comportamientos en la aplicación host. Luego, otro evento puede hacer que la funcionalidad vuelva al estado pasivo, dejando la aplicación host como antes. Este par de transiciones entre estados puede repetirse dependiendo del tipo de funcionalidad y de cómo el patrón de volatilidad rige a la misma.

Figura 2.2: Estados del ciclo de vida de la funcionalidad volátil de acuerdo al patrón de volatilidad

La activación de una funcionalidad ocurre en un momento dado, de acuerdo al patrón de volatilidad. Luego la funcionalidad dejará de estar activa y pasará al estado pasivo, como consecuencia de otro evento especificado en el patrón de volatilidad. El patrón de volatilidad puede estar determinado por un lapso de tiempo, una fecha, un evento externo (como una catástrofe), por el estado interno del sistema (cuando un contador alcance un valor) o una regla de negocio. A partir de la instanciación del patrón de volatilidad como elemento del sistema, el proceso de activación(deactivación) se podría automatizar. El patrón de volatilidad se podría diseñar e implementar en el sistema para que automáticamente la funcionalidad volátil sea activada y desactivada cuando sea necesario. Más adelante (en la sección 5.6) veremos cómo en esta tesis finalmente se logró hacer esto, con ciertas restricciones inherentes a la naturaleza del patrón de volatilidad de los side games.

En el mismo paper [RGDU08] los autores definen un DSL que permite especificar el patrón de activación de forma más formal. Así, se pueden definir las reglas que determinarán la política de activación de la funcionalidad de manera más formal y potencialmente automatizable. Ejemplo:

```
When
    NewFishBowlVideo(artist="Dolores O'Riordan")
Then
    Connect
    Concern FishbowlRiordan
```

Este ejemplo es una instancia del modelo simplificado de la especificación de activación de [RGDU08]:

```
When
    (Event_Pattern_Expression)
Then
    (Connect / Disconnect)
    Concern concern_name
```

Connexion pattern

En este trabajo se analiza un cuarto atributo que tiene que ver con la forma o el momento de instalación de la funcionalidad volátil. El patrón de conexión o de instalación, *Connexion pattern*. A continuación se explica este atributo, porqué es necesario y cuáles son las diferencias con el atributo Volatility pattern.

Como se explicó en los parrafos anteriores, los autores de asocian los procesos de activación y deactivación de la funcionalidad volátil con el atributo Volatility Pattern. Presentan un ejemplo de funcionalidad volátil encontrado en Amazon.com, acerca de un feature que es activado con la aparición en el mercado de un nuevo libro de Harry Potter. En tal ejemplo, el patrón de volatilidad indica el momento en que el feature debe ser instalado en la aplicación o activado para que los usuarios lo utilicen. Luego explican cómo se define el ciclo de vida de una funcionalidad volátil en terminos de los estados activo y pasivo de una funcionalidad, intercambiando los terminos active con connected y passive con disconnected, sin hacer distinción entre ellos. También se hace alusión a la configuración automática del proceso de activación y del proceso de conexión de la funcionalidad de manera indiferente. Finalmente, comparando su trabajo con otros, hacen incapié en que el atributo volatility pattern indica las mecánicas del proceso de conectar o desconectar los componentes volátiles.

Evaluando lo mencionado arriba, se puede ver que en [RGDU08] utilizan los conceptos de activación e instalación de la funcionalidad volátil de manera intercambiable. No se hace distinción entre los procesos de conexión y desconexión de la funcionalidad con los procesos de activación y deactivación de la misma. A partir del trabajo desarrollado en esta tesis, se encontró que la funcionalidad volátil debe ser instalada en el sistema primero, para poder luego cumplir con su patrón de volatilidad, es decir: ser activada o desactivada. Como se explicará luego, el proceso de instalación de la funcionalidad volátil no es algo trivial (Ver sección 5.6.2). Por lo tanto, aquí se plantea como necesario identificar este proceso y diferenciarlo del proceso de activación de la funcionalidad.

El proceso de activación ya explicado e identificado con el atributo Volatility pattern, tiene que ver con reglas de tiempo, de negocio, estado de la aplicación y demás, que determinan el momento en que la funcionalidad se vuelve disponible para ser utilizada. El proceso de instalación, tiene que ver con la mecánica en que la funcionalidad volátil es acoplada a la aplicación, o con el mismo significado, instalada en la misma. Tiene que ver con el momento en que se llevará a cabo la edición de código de la aplicación principal, o instalación via otros artefactos como AOP para introducir la funcionalidad volátil como un nuevo elemento en el sistema.

La funcionalidad volátil debe ser instalada en el sistema en algún momento, ya sea mediante la edición del código y posterior recompilación, o utilizando mecanismos más avanzados presentados en este trabajo o en [RNM+06, RGDU08].

No importa cuál sea el caso, en algún momento y bajo ciertas condiciones, la funcionalidad debe ser acoplada al sistema host. Entonces, se puede analizar para una funcionalidad volátil, la mecánica de instalación o conexión en el sistema. En este trabajo, se distingue el proceso de instalación del proceso de activación descrito en [RGDU08] en el atributo "Volatility pattern". Por ejemplo, un side game es instalado como funcionalidad volátil sobre un juego host y luego, cuando ocurre cierto evento del juego host, el side game es ofrecido al usuario (es activado). En este ejemplo se puede diferenciar el estado de conexión o acople de la funcionalidad volátil con el estado de disponibilidad o activación. Una funcionalidad volátil puede estar instalada en un sistema pero todavía no estar activa, esperando que ocurra algo en el sistema host para activarse.

Vale la pena mencionar explícitamente, que en esta tesis los terminos conexión, acople, instalación y plugging son tomados como sinónimos y corresponden al proceso en el que la funcionalidad volátil es agregada al sistema utilizando algún mecanismo particular. Estos terminos no equivalen al termino activar, que corresponde con el proceso en que la funcionalidad volátil se vuelve funcional, alterando el comportamiento de la aplicación host. Quedan entonces diferenciados los procesos de activación y de instalación de la funcionalidad volátil. A partir de esta diferenciación se introduce un nuevo atributo denominado *Connexion pattern* . El mismo define las reglas que determinarán el momento de instalación de la funcionalidad volátil en el sistema.

En el caso en que la funcionalidad pueda instalarse de forma automatizada, las reglas especificadas por el patrón de conexión, pueden reificarse en el sistema para formar parte de la automatización. De esta forma, se podría diseñar e implementar la funcionalidad, dejando que el proceso de instalación se realice automáticamente de acuerdo a las reglas pertinentes. Más adelante (ver sección 3.6.1) se mostrará un ejemplo de side game, en el que este atributo se puede diseñar e implementar en el sistema para lograr la automatización.

Los patrones de instalación y el patrón de volatilidad están muy relacionados. Para algunas funcionalidades volátiles el patrón de instalación será el mismo que el Volatility pattern, debido a que una vez instaladas las mismas permanecen activas. Por ejemplo, una funcionalidad temporal para una fecha como navidad, que ofrece reducciones de precios sobre productos es instalada e inmediatamente permanece activa hasta su desacople. En el ejemplo presentado en [RGDU08], mencionado arriba, sobre la promoción de Amazon.com cuando aparece el nuevo libro de Harry Potter, los atributos Connexion pattern y Volatility parttern son equivalentes, ya que la nueva funcionalidad es instalada y está disponible en ese mismo momento.

Con lo explicado hasta ahora sobre la diferenciación de los estados de activación e instalación, el ciclo de vida expuesto en [RGDU08] y explicado arriba, puede modificarse para incluir los estados de conexión. El ciclo de vida resultante

considerando el patrón de conexión se puede visualizar en la figura 2.3:

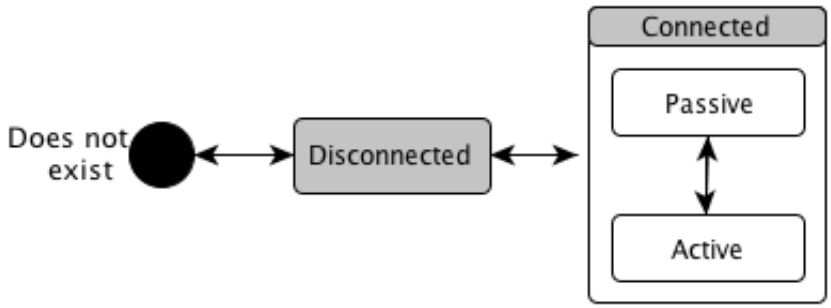

Figura 2.3: Estados del ciclo de vida de la funcionalidad volátil incluyendo procesos de instalación

Podemos ver en la figura 2.3 anterior que la funcionalidad volátil puede estar desconectada, esperando a que alguna regla de negocio se cumpla o ocurra algun evento de caracter temporal. Una vez que la funcionalidad se acopla al sistema, puede ocurrir que la misma esté en estado pasivo, esperando algún evento de la aplicación host. Así, cuando la funcionalidad volátil ya está acoplada al sistema, pueden ocurrir los cambios de estado entre activo y pasivo como se explicó anteriormente. Finalmente, en algún momento la funcionalidad se desacoplará de la aplicación host, dejándola como antes.

Además de alterar el ciclo de vida explicado en [RGDU08] también es necesario hacer una modificación al DSL mencionado arriba. Es necesario cambiar la forma de expresar el volatility pattern para dar lugar también al connexion pattern. Entonces, propongo las siguientes modificaciones, teniendo en cuenta un modelo simplificado del DSL:

```
Para el Volatility Pattern

    When
        (Event_Pattern_Expression)
    Then
        (Activate / Deactivate)
        Concern concern_name

El Connexion pattern quedaría entonces como era antes el Volatility pattern.

    When
        (Event_Pattern_Expression)
    Then
        (Connect / Disconnect)
```

```
Concern concern_name
```

Para finalizar este análisis mencionaré dos puntos importantes: el primero, la importancia de diferenciar los procesos y atributos hasta ahora explicados. El segundo, un posible cambio de nombre en el atributo "Volatility pattern".

- Primero:En un principio, las diferencias entre los atributos son sutiles, pero se hacen más visibles al considerar los procesos inherentes a cada atributo. Es importante poder diferenciar claramente estos procesos para lograr un mejor diseño, más claro, que permita una intergación correcta de las funcionaldiades volátiles. Justamente, considerar estos atributos por separado genera mayor claridad en la especificación de la mecánica de integración de las funcionalidades volátiles.

- Segundo:Quizás el nombre Volatility pattern no sea del todo correcto una vez que se introduce el concepto de Connexion Pattern. El termino Volatility pattern es batante general y no hace referencia directa al proceso que describe, el de activación. Creo que sería productivo renombrar el atributo a *Activation Pattern*, siendo este término más concreto y directo. Creo que el término Volatility pattern puede considerarse como algo más general que abarca tanto a Activation Pattern como a Connexion Pattern y que encierra los elementos necesarios para especificar la mecánica de integración de la funcionalidad volátil.

2.3. Componentes reusables (Funcionalidad volátil como plugin)

Construir software a partir de componentes reusables es importante para reducir costos [AT98]. Una de las premisas de la tesis fue poder programar funcionalidad volátil como componente reusable en distintas aplicaciones. Los side games fueron el caso de estudio. Los mismos dependen en cierto grado del juego base con el que corren. En algunos casos el grado de dependencia es tal que el side game no puede ser corrido junto con otro juego. Dicha dependencia puede ser debido a lógica de juego, semántica, temática de juego, gráfica, etc. Muchas veces no tiene sentido acoplar un side game a un juego simplemente porque la temática no corresponde, por ejemplo un side game de carrera de autos dificilmente pueda ser acoplado en un slot con temática elementos de cocina. Más allá de los factores que pueden hacer acoplable un side game con un juego particular, es deseable contar con side games que puedan, técnicamente, ser acoplados con cualquier juego. Con esto se abaratarían costos ya que un side game podría ser incluido en muchos juegos sin tener que desarrollar tal side game más de una vez. Un ejemplo de side game que sería útil para múltiples juegos es el doble o nada. El doble o

nada es una apuesta que se puede ver en juegos de slots o de cartas que en general puede acoplarse a cualquier juego que involucre entrega de premios mediante pago. La idea de esta tesis es poder llegar a implementar funcionalidad volátil que funcione como plugin. De esta forma, una vez diseñada e implementada una funcionalidad volátil, se podría reutilizar en distintas aplicaciones (cumpliendo por supuesto con ciertos requisitos). Como ejemplo se desarrollaron side games, que solo necesitan que ciertos eventos sean anunciados por un juego principal para poder ser incluidos. Esto constituye una pequeña prueba satisfactoria donde se obtuvieron componentes reutilizables que se pueden acoplar dinámicamente con una aplicación base.

2.4. Necesidad de una Arquitectura

En [RGDU08] se explica que debido a que el problema no está estudiado en profundidad y a que en general no hay una práctica conocida y utilizada para manejar la aparición de este tipo de funcionaliades, se pueden observar aproximaciones mas bien directas para solucionarlo. Entre las aproximaciones usuales, se puden destacar dos aproximaciones. La primera (code patching) es la más directa y usual, mientras que la segunda (refactoring models) trata de atacar el problema de forma más profunda, pero como se muestra, también tiene fallas visibles:

Code Patching

Es una práctica común que a partir de la aparición de nueva funcionalidad volátil, que luego será removida, se desarrolle código adicional y se modifique apropiadamente el código existente. Esta manera de proceder, según se manifiesta en [GDRU09], además de demandar tiempo y esfuerzo, al involucrar la edición intrusiva del código fuente de la aplicación, conlleva el riesgo de contaminarlo e introducir errores. Debido a restricciones temporales, por ej. time-to-market restrictions [Wik09e], la necesidad de nuevas funcionalidades volátiles es usualmente solucionada en instancias de implementación: nuevos componentes y código son agregados cuando la funcionalidad debe ser introducida y los mismos componentes son removidos o desabilitados luego, cuando la funcionalidad tiene que ser desactivada. A la larga, esta práctica tiene impactos negativos, causando diferencias entre el diseño y la implementación y llevando a un funcionamiento incorrecto y deterioro de la calidad.

Refactoring Models

Tratar la funcionalidad volátil a nivel de modelos también puede causar problemas. Debido a la naturaleza de los requerimientos volátiles, imprevistos, los

diferentes modelos de diseño de la aplicación (conceptual, navegación, inerface) no están usualmente preparados para las modificaciones necesarias. Es necesario refactorizar los modelos al menos dos veces, cuando se agrega y remueven las funcionalidades correspondientes. El refactoring de los modelos para acomodar la nueva funcionalidad volátil puede ser un desperdicio de esfuerzo ya que puede resultar en un sobre-diseño de la aplicación solo para introducir una funcionalidad que probablemente desaparecerá luego.

Al considerar los servicios volátiles desde etapas tempranas del desarrollo de la aplicación host, se apunta a acotar el impacto de la posterior incorporación de funcionalidad volátil. Al preparar el campo para la posterior incorporación de funcionalidad volátil, se mejora el seguimiento de las decisiones de diseño, en lugar de atacar el problema en etapas de codificación, donde cambios radicales costarían demasiado esfuerzo [GDRU09]. Entonces, una arquitectura que permita el diseño de aplicaciones que soporten funcionalidad volátil eventualmente, desde etapas tempranas, es una herramienta que puede ayudar a desarrollar sistemas con mayores posibilidades de evolución a menor costo.

A partir de lo recién expuesto y en pos de una forma de evolución más controlada, donde la incorporación y extracción de funcionalidades volátiles no deteriore el diseño, el código y el funcionamiento del sistema, se genera la necesidad de una aproximación específica para atacar el problema. Dicha aproximación debe soportar la incorporación de funcionalidad volátil de manera tal que el sistema se mantenga estable y mantenible. Algunos requisitos de una aproximación viable fueron expuestos en [RNM+06, RGDU08], aquí se enumeran junto con los encontrados durante el desarrollo de esta tesis:

- Las funcionalidades volátiles deben ser modeladas como funcionalidades de primera clase, de forma independiente de la aplicación host, para poder ser integradas de forma modular sobre la misma, sin alterar su diseño. De esta forma se reduce la necesidad de refactorizar elementos de la parte core una vez que la funcionalidad es removida.

- La funcionalidad volátil debe poder ser acoplada y desacoplada de la aplicación host de forma transparente: el código de ambas partes no debería ser modificado.

- La funcionalidad volátil debe poder comunicarse con la aplicación host de forma bidireccional, produciendo y recibiendo información para y desde la misma.

- La funcionalidad volátil, una vez acoplada, puede ser activada y desactivada de acuerdo a ciertos patrones, que pueden ser de caracter temporal o ser condiciones sobre los datos de la aplicación principal. Estos patrones se deberían poder especificar para automatizar la activación y desactivación.

Los requisitos recién mencionados delinean las características de una posible solución al problema de la aparición de funcionalidad volátil. En esta tesis se propone una arquitectura que cumpla con estos requisitos. La arquitectura presentada consiste en:

- Una metodología de trabajo para el diseño e implementación de funcionalidad volátil.Orientada a la integración de funcionaldiades volátiles en aplicaciones RIA, que provea una infraestructura para la integración de funcionalidades volátiles de forma transparente.
- Un modelo(arquitectura per se) para soportar la integración de nueva funcionalidad de forma transparente, manteniendo comunicación bidireccional con la aplicación host.
- Herramientas que pueden ser reutilizadas para: Especificar condiciones de (de)activación de funcionalidad, agregar código de integración de forma transparente, lograr comunicación bidireccional.

El objetivo de este trabajo es encontrar una posible solución para poder acompañar la evolución de un sistema de forma controlada. La evolución debe ser considerada como una necesidad global y debe ser soportada en todo el proceso de desarrollo del sistema. Esto implica que mecanismos, modelos y metodologías deben ser proveidos para facilitar la tarea de evolución al desarrollador [GCRFPL02]. Así, contar con una arquitectura que permita un mejor diseño y control de la funcionalidad volátil puede facilitar la construcción de un sistema que sea más fácilmente extensible y menos costoso.

Capítulo 3

Contexto

3.1. Introducción

Los juegos web pueden ser implementados con distintas tecnologías. La utilización de Flash como plataforma de desarrollo es muy común en este tipo de aplicaciones. Otra forma de construir juegos interactivos, aunque menos utilizada, es utilizar Ajax junto con mecanismos javascript, CSS y HTML para ejecutar en el browser la vista y parte de la lógica de los juegos, dejando en el servidor la mayor parte de la lógica de los mismos. En este trabajo se estudia esta última alternativa. El ámbito de juegos de entretenimiento y apuestas online fue escogido debido al trabajo desarrollado en el Laboratorio de Investigación y Desarrollo de Software de Entretenimiento de Lifia, donde actualmente se están evaluando alternativas para la incorporación de Side Games a juegos de casino. La industria de los juegos y entretenimiento constituye un campo que en la web está en gran expansión y necesita ser explorado. Actualmente es posible jugar a miles de juegos de casino online.

Las Slotmachines [Wik09d] online son juegos muy populares. En este contexto se evaluó la incorporación de funcionalidad volátil. En concreto, la funcionalidad volátil estudiada consiste en juegos secundarios accesorios y simples, mientras que la aplicación host es un juego de casino online completo.

En los siguientes párrafos se describirán sucintamente algunas de las tecnologías utilizadas en el desarrollo de esta tesis. Algunas sirven solo como herramientas para lograr el objetivo de esta tesis, mientras que otras constituyen objetos de estudio en si mismas, siempre en relación al problema recurrente: la funcionalidad volátil. Se explicará rápidamente que es Seaside y cómo se acopla con Scriptaculous y Prototype para proveer al desarrollador herramientas para la construcción de RIA. Otras tecnologías que se utilizaron, como Glorp[Kni00] y PostgreSQL son introducidas en los apartados correspondientes. Luego de intro-

ducir las tecnologías utilizadas como parte del contexto de desarrollo de esta tesis, se explicarán los conceptos relacionados con juegos de casino web, importantes para entender el trabajo aquí presentado.

3.2. Juegos(RIAs) en Smalltalk + Seaside + Scriptaculous.

Los juegos web constituyen una de las muchas formas de juegos online disponibles. Con el crecimiento de la WWW, los browsers fueron haciéndose más sofisticados, transformándose en un buen lugar para ofrecer juegos online. Mientras tanto, distintas tecnologías se integraron a HTML, como JavaScript y Flash, siendo esta última la más popular en la actualidad para implementar juegos que corren en browsers. En los últimos años, tecnologías como JavaScript y CSS resurgieron y ganaron popularidad, como parte del conjunto de tecnologías denominadas Ajax, convirtiéndose en una posible alternativa a Flash para implementar juegos web.

Ajax, en pocas palabras, constituye lo necesario para comunicar el browser con el servidor sin causar un refresco completo de página. En el esquema de comunicación típico de HTML, el browser efectúa un request a un servidor, y este le devuelve una respuesta, causando un refresco de página completo. La comunicación entre el cliente y el servidor, desde el punto de vista del usuario final, es sincrónica, ya que el mismo debe esperar a que el browser reciba la respuesta y recargue la página por completo. Ajax introduce mecanismos asincrónicos, que permiten enviar un request al servidor y recibir la respuesta sin la necesidad de recargar la página completamente. Así, se pueden refrescar solo porciones del contenido que visualiza el usuario final, logrando resultados que acercan la experiencia web a la experiencia desktop.

Ajax no es más que un conjunto de tecnologías usadas desde hace muchos años (HTML, JavaScript, DOM, XML, CSS) que fueron agrupadas en un término. Desde que el término fuera acuñado por Jesse James Garrett en el año 2005, se popularizó y convirtió en objeto de investigación y desarrollo intensivo. Uno de los signos de su crecimiento y popularidad es la aparición y uso masivo de librerías y frameworks [CBL07]. Actualmente la mayoría de los frameworks más utilizados para desarrollar aplicaciones web (Struts, Spring, Ruby on Rails, Symfony, Seaside, etc.) tienen soporte para Ajax o permiten la integración de librerías que lo hagan, como Google DWT, Yahoo YUI, MooTools, Prototype, etc.

Prototype [Wik09c] es un framework, que provee extensiones para el lenguaje JavaScript, facilitando la interacción con el objeto XMLHttpRequest, vital para Ajax. Scriptaculous es una librería que utiliza Prototype para ofrecer un nivel más elevado (abstracto) de programación proveyendo widgets, efectos visuales y comunicación asincrónica. Permite utilizar mecanismos visuales y Ajax escribiendo

pocas lineas de código en un nivel de abstracción elevado.

Seaside es un framework para desarrollar aplicaciones web sofisticadas en Smalltalk. Provee un conjunto de abstracciones sobre HTTP y HTML que permiten construir rápidamente aplicaciones web altamente interactivas, mantenibles y reusables[DLR07]. Seaside provee integración completa con Scriptaculous y Prototype, permitiendo el uso de las librerías sin tener que escribir código JavaScript, desarrollando completamente en Smalltalk. Todos los mecanismos Ajax, efectos visuales, widgets, pueden ser configurado e integrados utilizando objetos Smalltalk.

En este trabajo se utilizaron las tecnologías introducidas hasta ahora de forma integral, utilizando Cincom Smalltalk como plataforma de soporte. Smalltalk y particularmente Seaside permiten la programación web utilizando Ajax de forma flexible, dinámica y con un alto nivel de abstracción, sin tener que necesariamente utilizar JavaScript. Este alto nivel de abstracción provisto por la integración de Prototype y Scriptaculous en Seaside permite efectuar request asincrónicos, introducir efectos visuales y otros mecanismos complejos de forma directa en Smalltalk. Así se puede escribir código como:

```
html anchor
    onClick:(html request callback:[self doSomething]);
      onClick:(html effect disappear)
      with:'async request and disappear'
```

Ejemplo de código Ajax usando Seaside y Script.aculo.us: Este código genera un link que al ser clickeado, efectúa un request asincrónico al servidor, que ejecutará el bloque de código [self doSomething] y genera un efecto visual, desapareciendo gradualmente.

El ejemplo anterior muestra como mediante código escrito en smalltalk se logra comportamiento en el browser que utiliza javascript. El código necesario para lograr el mismo efecto utilizando solamente JavaScript, Prototype y script.aculo.us sería parecido al siguiente:

```
<script type="text/javascript">
    function FadeEffect(element){
        new Effect.Fade(element, { duration:1});
    }
    function MakeRequest(){
       new Ajax.Request(url, { method: 'get'});
    }
</script>
...
```

```
<a id='link' onclick="MakeRequest(); FadeEffect('link')">
    async request and disappear
</a>
```

Podemos ver que gracias a la integración transparente de script.aculo.us y Prototype con Seaside, se utilizan indirectamente mecanismos complejos de JavaScript y Ajax solo escribiendo código Smalltalk. De esta forma se evitan las complejidades de JavaScript y reciben los beneficios del dinamismo de Smalltalk: hot debugging, introspección, modificación de código on the fly, herramientas de refactoring, entre otros.

3.3. Slotmachines

Una slotmachine es una máquina de apuestas (ver figura 3.1) que posee al menos 3 reels. Un reel es una especie de rueda o cinta continua que posee símbolos. Estos reels giran cuando se presiona un botón llamado botón de spin [Wik09d]. El jugador recibe premios de acuerdo a patrones predefinidos de símbolos que quedan dispuestos una vez que los reels terminan de girar. Las slots surgieron a fines del siglo XIX y su evolución fue notable, comenzando con aparejos mecánicos complejos e inmensos, continuando por versiones electrónicas más pequeñas y atractivas para los jugadores. Con el avance de la tecnología, el aumento de las prestaciones y la disminución de los costos y tiempos de fabricación, las Slotmachines fueron ganando popularidad y esparciéndose entre los casinos de todo el mundo. Se han convertido en unos de los juegos más populares y se estima que los ingresos de un casino están constituidos en un 70-80% por las máquinas tragamonedas. Las versiones online de las slomachines físicas y otros juegos de casino están ganando popularidad y convirtiendose en gran parte de la industria. Constituyen una fuente económica no solo para casinos, sino para empresas dedicadas a este tipo de emprendimientos: online gambling y todos los sectores secundarios que son afectados como desarrolladores de software, fabricantes de hardware, diseñadores, proveedores de internet, etc.

Los juegos en linea no están limitados unicamente a las slotmachines, miles de juegos tradicionales son implementados en sus versiones online, como la Ruleta, Poker, carreras de caballos, etc. y miles de nuevos juegos son posibles gracias a internet y las nuevas tecnologías relacionadas como dispositivos móviles.

3.4. Side Games

Los side games son juegos optativos secundarios que mantienen alguna relación con un juego principal y son ofrecidos al jugador bajo alguna circunstancia dada. Por ejemplo, en un Bingo online, que llena los cartones automáticamente mientras salen los números, se puede ofrecer un juego secundario rápido, como

Figura 3.1: Ejemplo de Slotmachine: Slot de ID Interactive

una carrera de caballos o un spin de una slotmachine. Los side games suelen ser utilizados para introducir juegos en el mercado paulatinamente, ofreciéndolos primero como juego secundario para evaluar el impacto en los consumidores. También pueden servir como forma de promocionar juegos presentando versiones compactas, como publicidad en juegos mayores ya instalados. En resumidas cuentas, se busca en general, uitilizar los side games como mecanismos para aumentar las ganancias disminuyendo los costos de producción; ya que un mismo side game se puede utilizar en distintos juegos.

Hay ciertas características deseables en los juegos que los hacen redituables para la empresa que los brinda y atractivos para el jugador. Deben ser fáciles de producir y mantener, de aprender y jugar, divertidos, brindar grandes premios y proveer un grado de entusiasmo considerable e incremental a medida en que el jugador progresa en el juego. Estas son características que entre otras constituyen a que un juego a la larga produzca ganancias para el proovedor y sea atractivo para el jugador. En ciertas ocasiones los juegos no cuentan con estas características, por lo que resultan necesarias modificaciones o alguna ayuda extra. Por ejemplo, en el tradicional blackjack el jugador no experiencia el grado de entusiasmo esperado ya que el top payout es bajo y cuando se utilizan estrategias de juego que minimizan los riesgos, el juego se puede convertir en aburrido. Como una ayuda a estos problemas en diversos juegos de casino, se introducen apuestas alternativas conocidas como side bets y en algunos casos, juegos secundarios conocidos como side games. Estas características brindan la oportunidad de ganar más jugando al mismo juego, aumentando los riesgos, incrementando el entusiasmo del jugador,

brindando mayor diversión al jugador y ganancias al casino.

3.4.1. Side Games: Funcionalidad volátil en juegos de casino.

Los side games pueden ser implementados como plugins que se acoplan y desacoplan a los juegos principales. Este acoplamiento puede ser temporal debido a, por ejemplo, una promoción del sitio o del casino. Es deseable, a su vez, que esto de produzca dinámicamente (en tiempo de ejecución), para no recompilar el juego y abaratar costos. El caracter temporal y dinámico de la integración entre juegos completos y side games, sienta las bases para la implementación de los últimos como funcionalidad volátil. De esta forma, los side games en este trabajo son estudiados como ejemplo de funcionalidad volátil en juegos online de casino (que cumplen el rol de aplicación host-Rich Internet Application). La figura 3.2 muestra un esquema con lo recién explicado.

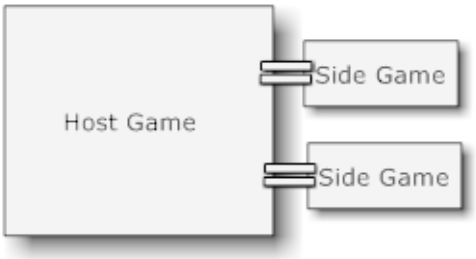

Figura 3.2: Side Game como Plugin.

3.5. Juegos implementados como caso de estudio

Para poder evaluar el problema presentado, se desarrolló un prototipo de slot-machine online. La slot elegida se basa en el juego de dados Generala [Wik09b] y cuenta con todas las características de una slotmachine comercial: elección de cantidad de lineas de apuesta, spin, respin, recuperación ante caidas, cash in, cash out, etc. La aplicación que corresponde al juego principal de slot se situa en el contexto de una aplicación mayor: un casino online o sitio de apuestas donde los usuarios pueden elegir entre múltiples juegos de apuestas. Este ejemplo sirve como aplicación principal donde se incorporán las funcionalidades volátiles. Se desarrollaron 2 side games para integrar como funcionalidad volátil del juego principal. El primero fue una versión de la apuesta "doble o nada"[Wik09a] basada en una tirada de dos cartas. El segundo side game se trata de una versión del juego *Card War* [Wik09f]

3.5.1. Slotmachine Generala

El juego es una adaptación a slotmachine del juego de dados del mismo nombre. Consta de 5 reels, que giran al presionar el boton spin. Solo hay 3 hileras de sílmbolos o "lanes ", por lo que la cantidad visible de símbolos por reel es 3. El juego brinda la posibilidad de apostar en múltiples lineas. Cada linea consta de un símbolo por reel, formando un conjunto de 5 símbolos (correspondiente a 5 dados). Cada linea apostada conforma un conjunto de símbolos que puede otorgar premios de acuerdo a la disposición de los mismos. En la figura 3.3 se pueden ver estas características en un screenshot del juego implementado.

La tabla de pagos del juego establece los posibles premios a partir de la combinación de símbolos resultante del spin. Las combinaciones posibles se corresponden con las distintas jugadas del juego original generala, como poker, escalera, fullhouse, etc.

En el juego original de dados, el jugador comienza su partida arrojando los 5 dados. Este evento se traduce en el juego de Slot adaptado mediante la ejecución de un Spin, que hace girar los 5 carriles que contienen las posibles combinaciones de dados. Además de esto, la lógica del juego original establece que luego del primer tiro se pueden retener dados a elección, uno o más, para realizar una segunda y luego una tercera tirada de dados. Esta característica se adaptó al juego de slot mediante el feature respin. El Respin es una accion mediante la cuál el jugador puede hacer girar solo un subconjunto de reels. Mediante la elección de los reels a girar el jugador determina cuáles serán los dados que se arrojarán. El jugador puede elegir de 1 a 4 reels para retener y realizar hasta dos respins de los reels restantes. En cada respin se calculan las combinaciones de símbolos entre los reels retenidos y los reels de respin para presentar los premios obtenidos. El jugador puede elegir luego del spin e incluso luego del primer respin, hacer un respin incluso si posee premios.

El juego permite al jugador retirar su dinero en cualquier momento, depositando los créditos en la cuenta que mantiene en el sitio. Por supuesto, el jugador puede transferir créditos de su cuenta al juego de manera arbitraria siempre que posea créditos disponibles.

El prototipo posee recuperación automática frente a desconexión. Mediante un sistema de persistencia de eventos se puede recuperar la sesión de juego del jugador. Esto es importante para mantener consistencia principalmente en la cuenta donde el jugador deposita su dinero. Cómo consecuencia de este sistema, el jugador también puede elegir salvar su juego para continuar con la sesión luego, de manera explícita.

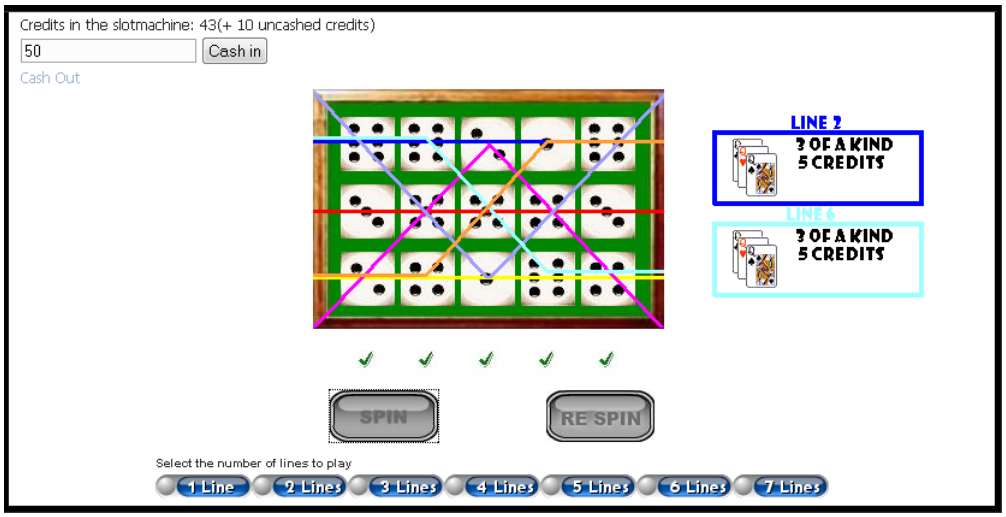

Figura 3.3: Screenshot del juego de Slots implementado: Generala

3.5.2. Side Game: Doble o Nada

Doble o nada es en realidad un tipo de apuesta que se ofrece una vez que un juego termina, o sea cuando el resultado de otra apuesta es presentado. El doble o nada consta en duplicar el valor de lo apostado en una siguiente apuesta. El jugador que gana puede duplicar su ganancia y el jugador que pierde puede perder el doble. La versión implementada (ver figura 3.4) varía de la original en que solo es ofrecida cuando el jugador obtiene un premio en el juego principal. En ese momento, se ofrece al jugador la posibilidad de duplicar su premio o perderlo por completo. El jugador elige un color, negro o rojo y luego una carta es escogida al azar, si el color de la carta se corresponde con el color elegido por el jugador, este resulta ganador. Cuando el jugador resulta ganador, los créditos ganados son depositados en la cuenta del juego principal.

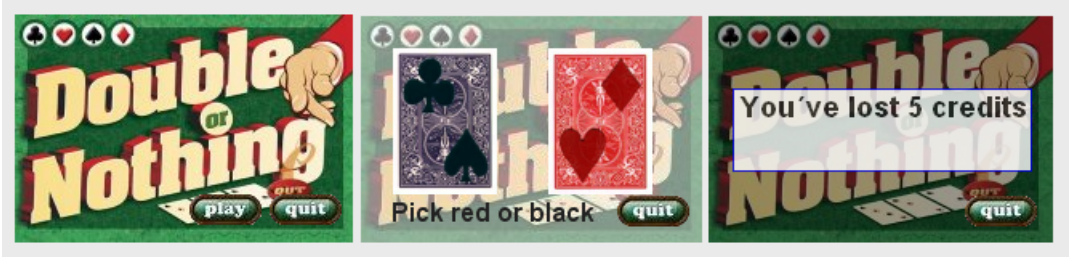

Figura 3.4: Screenshots de un side game implementado: Doble o Nada

3.5.3. Side Game: Card War

Cada vez que en el juego principal giran los reels, el side game tiene una probabilidad de ser ofrecido al jugador. Cuando giran los reels se evalúa un evento aleatorio configurado con cierta probabilidad, y si tal evento resulta satisfactorio, el side game es ofrecido. Así, el juego secundario es ofrecido por ejemplo el 60% de las tiradas del juego principal. Una vez ofrecido, el jugador puede elegir apostar al juego secundario. El monto de la apuesta es a elección y una vez determinado el juego comienza. El mismo consta en la elección aleatoria de dos cartas, una carta pertenece al jugador y la otra al casino. La carta más alta es la ganadora. En caso de igualdad ocurre un empate y el jugador puede elegir desempatar apostando nuevamente o perder la mitad de su apuesta rindiendose. Cuando el jugador resulta ganador, los créditos ganados son depositados en la cuenta del juego principal.

Figura 3.5: Screenshot de otro side game implementado: Card Wars

3.5.4. Aplicación contenedora: Sistema Casino

El juego de slot generala forma parte de un paquete de juegos que es ofrecido en un casino online. Se implementó un prototipo de casino online en el que los usuarios pueden registrarse con un nombre de usuario y contraseña. Luego de la registración el usuario podrá loguearse para elegir un juego disponible (slot generala, ver figura 3.6). El usuario por supuesto puede salir del sitio realizando logout.

En el casino online el usuario posee una cuenta con créditos en la que puede realizar solamente depósitos. Las extracciones de la cuenta se corresponderían con transferencias bancarias hacia una cuenta real. Esto por supuesto no fue implementado. En todo momento el jugador puede tener acceso a su cuenta de créditos para depositar o simular la extracción total de créditos.

El sitio ofrece juegos disponibles que pueden ser utilizados por los usuarios para realizar apuestas. El sitio está preparado para soportar múltiples juegos, pero por cuestiones del alcance del trabajo solo se implementó el juego slot generala. Luego veremos que la aparición de side games acoplados al slot generala permite ofrecer distintos paquetes que simulan la existencia de distintos juegos. Por ejemplo el casino puede ofrecer el slot generala y además el slot generala con el side game doble o nada acoplado. Así el jugador puede elegir entre dos paquetes.

Los jugadores pueden elegir un juego, jugarlo y luego terminar de jugar, salvando la instancia del juego o simplemente abandonarlo. Esto se debe a que el juego implementado soporta la recuperación de sesiones de juego. El sitio casino online fue implementado de manera tal que sea compatible con la recuperación de sesiones en caso de algún tipo de error.

Figura 3.6: Screenshot de la aplicación Casino

3.6. Características de los side games como funcionalidad volátil.

En el capitulo anterior (2.2.2), se explicaron los conceptos de extent, intent, volatility pattern y connexion pattern. A continuación se presenta una pequeña

evaluación de los atributos en el side game doble o nada que sirve para ejemplificar los conceptos y ayuda a profundizar las ideas.

Por supuesto el patrón de volatilidad es particular para cada side game. No obstante, en general la activación de un side game no dependerá de factores temporales, sino de eventos del juego host donde está instalado el side game. Por ejemplo, el side game card war, es ofrecido dependiendo de una probabilidad prestablecida cuando el juego principal termina su jugada. Cuando esto ocurre, se determina si el side game será activado evaluando la probabilidad con la que el mismo está configurado.

El patrón de conexión establece la mecánica de acople del side game, determinando el momento en que el mismo será instalado en el juego principal. Es posible una automatización del proceso de instalación utilizando el patrón de conexión para evaluar el momento que que se disparará el proceso. Los side games en general son instalados en un juego cuando se quiere promocionar otro juego, para alterar temporalmente el payout total de la máquina, o el hit ratio, conceptos de juegos de casino. Como consecuencia, el patrón de instalación será probablemente especificado en terminos de eventos que quizás no se puedan especificar en un DSL, como "Se inició la promoción del juego X".

Alternativamente, los side games también pueden ser instalados como consecuencia de algún evento o cambio de estado en el juego principal o contexto donde este se ejecuta. Por ejemplo, se podría ofrecer una serie de side games cuando un usuario es promovido en su categoría de cuenta en el casino. Cuando un usuario compra una cuenta premium, el casino podría ofrecer side games como bonificación o incentivo para hacer más atractivo el paquete. En ese momento los side games se instalan en los juegos que el usuario tiene disponibles y cada side game será activado de acuerdo a su patrón de activación.

3.6.1. Analisis de los atributos de funcionalidad volátil en un caso de estudio implementado.

Se presenta aquí un pequeño caso de estudio sobre los atributos de la funcionalidad volátil. En la sección 2.2.2 se presentó la información necesaria para comprender los atributos evaluados aquí.

Caso de estudio: Side game doble o nada.

- **Intent.**

  Content: El side game se puede instalar en todos los juegos del casino que devuelvan premios.

  Interface: El side game altera el look and feel del juego principal.

  Business processes: El side game altera el proceso de acreditación de premios ya que de jugarse el side game, los premios se ven afectados.

User operations: Las operaciones del sistema total se aumentan con las operaciones que provee el side game.

Nota: El atributo intent define los lugares de impacto de la funcionalidad volátil, para más información sobre los conceptos Content, Interface, Business processes y User operations ver [RNM+06, RGDU08].

- **Extent.**

  El side game se instalará solo sobre el juego de Slots Generala.

- **Volatility pattern.**

  El side game se activará cuando el juego principal devuelva un premio como resultado de una jugada. Se desactivará si el usuario decide rechazar el side game o realizar un play ignorando

  Ejemplo De Volatility pattern para el side game doble o nada.

```
"activate volatile functionality DoubleOrNothing
    when User.Account > 100
         and PlayDoneEvent := event
         and event.prizeWon = true ".
```

- **Connexion pattern.**

  El side game se acoplará al sistema de acuerdo a políticas de promoción de juegos del casino o empresa que lo ofrece. Puede ser resultado de una campaña para promocionar un juego o para ofrecer combinaciones más atractivas en un conjunto de juegos, como mayor payout o hit ratio.

  Ejemplo De Connexion pattern para el side game doble o nada.

```
"Connect volatile functionality DoubleOrNothing
    when UserAccountPromoted := event and
         event.newCategory = PremiumUser ".
```

3.7. A Continuación

Una vez explicado el contexto en detalle donde se estudiarán las cuestiones de funcionalidades volátiles se puede comenzar a explicar cuales fueron las decisiones tomadas para solucionar las cuestiones de diseño e implementación. En el siguiente capítulo se introducen estas decisiones a través de una arquitectura que brinda soporte para funcionalidad volátil.

Capítulo 4

Arquitectura

4.1. Introducción

Anteriormente se mencionaron los problemas de incorporar funcionalidad volátil a nivel de código directamente (Ver sección 2.4). Se explicó también que incluso existen problemas atacando la funcionalidad volátil a nivel de modelos (Ver sección 2.4). Se planteó entonces la necesidad de atacar estos problemas con anticipación a un nivel más elevado, por lo que una arquitectura que de soporte sería útil. En los trabajos [RNM+06, RGDU08] se introduce una arquitectura que permite el manejo de funcionaldiad volátil. A continuación se explicará cómo en esta tesis se estudian dichos trabajos y se encuentra la necesidad de una aproximación con algunas diferencias.

En los estudios mencionados en el parrafo anterior se introduce el problema de funcionalidad volátil y se lo estudia desde el punto de vista de los aspectos de diseño conceptual y navegacional principalmente. Se propone modelar la funcionalidad volátil separadamente del sistema host, utilizando técnicas de ingeniería de software basadas en OODHM[Ros96]. Con respecto a aspectos de implementación, presentan una extensión al framework de Java Struts[Web:Struts] que permite alterar la forma en que se construyen las páginas resultantes, para incluir el contenido correspondiente a la funcionalidad volátil. Dicha extensión es parte de una arquitectura que permite comunicación desde la parte volátil hacia la parte core. Tal comunicación se logra mediante relaciones entre ambas partes, utilizando inversión de control para lograr transparencia con respecto a la parte core (Los modelos e implementación core, ignoran la existencia de funcionalidad volátil).

En esta tesis se siguen las mismas lineas del trabajo recién mencionado. No obstante, en el mismo no se estudia completamente el problema de la comunicación bidireccional entre vistas y modelos, debido a que está orientado al diseño.

El trabajo aquí presentado posee una inclinación a los aspectos de implementación, por lo que resulta necesario estudiar cómo lograr comunicación bidireccional entre modelos y vistas para poder implementar el tipo de juegos web aquí propuestos. Un side game necesita información constantemente del juego en el que está instalado. No es suficiente que un side game conozca al juego principal y pueda consultar información y disparar acciones en el mismo. Es necesario que el juego principal notifique al side game una serie de eventos o condiciones que ocurren durante el desarrollo del juego. Esto se debe a que el desarrollo de un side game depende del desarrollo del juego principal, por definición de side game. Entonces la aproximación delineada en este trabajo presentará un modelo en el que la comunicación bidireccional entre modelos y vistas sea primordial.

Se hace la distinción entre comunicación entre vistas y modelos porque este tipo de juegos requieren una coordinación entre lo que pasa en el modelo (invisible al usuario) y lo que el usuario percibe (animaciones, resultados, etc.). Por ejemplo, cuando el jugador presiona el botón de spin, comienza una animación de los reels en el browser, mientras que en el servidor automáticamente se genera una jugada y se almacenan los resultados de la misma, pagando premios, decrementando créditos y demás. Una vez que la animación termina, el browser debe mostrar el estado consistente del juego, con todos los resultados. Entonces claramente existe una diferencia entre los estados del modelo y la vista. Entonces, si a este entorno es necesario agregar un side game, que debe ser mostrado cuando el jugador gana ciertos créditos y solamente después de que la animación de los reels haya finalizado, se deben coordinar eventos tanto de los modelos de los juegos como de las vistas.

Un side game podría ser incluido en más de un juego principal. Por ejemplo, el side game doble o nada, se puede incluir en cualquier juego que posea una apuesta y presente resultados. Teniendo esto en cuenta, resulta un desperdicio desarrollar un side game de estas características y no poder incluirlo en más de un juego sin tener que realizar grandes esfuerzos. Es por esto que en este trabajo se trató de que la funcionalidad volátil pueda ser manejada como un feature acoplable y desacoplable dinámicamente, que pueda ser reutilizado en distintos contextos cumpliendo determinadas reglas.

Entonces, al trabajo desarrollado en [RNM+06, RGDU08] se incorporaron dos nuevos desafíos:

- Comunicación bidireccional entre vistas y modelos y
- Funcionalidades volátiles como posibles plugins.

Considerando lo anterior, se buscó una solución que de soporte a la incorporación de funcionalidad volátil completamente conectada con la aplicación host (información en ambas direcciones, modelos y vistas) y que permita maximizar

las posibilidades de generar plugins a partir de tales funcionalidades. Como consecuencia, se siguió una linea de trabajo que permita manejar la funcionalidad volátil como algo separado de la aplicación principal, que pueda ser acoplado a la misma de la forma más transparente posible. Esto quiere decir, sin alterar significativamente el código y diseño de la aplicación host. A su vez se trató de mantener cierto nivel de independencia de las partes volátiles hacia las partes core, con el objetivo de que las funcionalidades volátiles se conviertan en posibles plugins para distintas aplicaciones.

A través del diseño incremental, implementación y refactoring continuo de los prototipos antes mencionados(Ver sección 3.4), fueron apareciendo componentes o patrones bien identificables en los modelos y en el código de ambas partes core y volátiles. Algunos de estos componentes se fueron afianzando como elementos que podrían ser reutilizados genéricamente en otros desarrollos de la misma índole, que requieran soporte de funcionalidad volátil. Así, siempre siguiendo los conceptos propuestos en [RNM+06, RGDU08] y la idea de soportar comunicación bidireccional maximizando la posibilidad de reutilizar partes volátiles, fue emergiendo paulatinamente una arquitectura.

Esta arquitectura puede ser considerada como un marco de trabajo y guía conceptual para desarrollar aplicaciones con soporte de funcionalidad volátil. Es necesario mencionar que la arquitectura aquí presentada es el resultado de meses de trabajo sobre solo una aplicación y algunas funcionalidades volátiles, por lo que seguramente, la misma aún dista de ser completa o definitiva. Por supuesto, la implementación de funcionalidad volátil en otro tipo de aplicaciones web podría necesitar de modificaciones en la utilización de estas ideas.

4.2. Ejes de la arquitectura

En los parrafos anteriores se explicó resumidamente la naturaleza de la arquitectura aquí presentada y quedaron delineados algunos puntos importantes que la misma debe cumplir, como comunicación bidireccional. En los siguientes parrafos se detallarán las ideas anteriormente introducidas, sobre las cuales se centra la arquitectura en cuestión.

Esta arquitectura posee cuatro ejes conceptuales relacionados entre si:

- Diseño e implementación por separado
- Integración transparente
- Comunicación bidireccional
- Reutilización de la Funcionalidad volátil.

Existe cierta simbiosis entre estos conceptos, pero también cierto nivel de tensión entre ellos. Por ejemplo, a medida en que se maximiza la funcionalidad

volátil como elemento reutilizable, se incorporan restricciones de comunicación con el sistema host. A su vez, es dificil lograr una integración transparente de la funcionalidad volátil cuando la misma esta altamente conectada o comunicada con el sistema host. Por su parte, el diseño e implementación por separado ayuda a que la integración sea transparente y a que la funcionalidad volátil sea potencialmente reutilizable. Todo esto hace dificil introducir funcionaldiad volátil completamente reutilizable, altamente conectada con el sistema host y de forma transparente. Esta arquitectura tratará de proveer la posibilidad de encontrar cierto balance entre estos ejes.

A continuación se explican un poco más en detalle los cuatro conceptos planteados arriba:

4.2.1. Diseño e implementación en forma separada

La idea es que las funcionalidades volátiles se desarrollen por separado de la aplicación core y de forma transparente. Al agrupar funcionalidad por conjunto de requerimientos, pertenecientes al mismo tema o interés, identificado como concern, y al desacoplar claramente estos concerns en cada etapa de desarrollo se obtiene software que es más capáz de evolucionar [GDRU09]. Al corresponder a diferentes concerns, la funcionalidad volátil y la aplicación core, es conveniente desarrollaros por separado. En [GDRU09], se explica que los potenciales enriedos en el diseño y códificación, generados por la implementación de más de un concern en el mismo módulo (aplicación core y funcionalidad volátil juntos) son una fuente de problemas de mantenimiento. Aún utilizando ambientes de configuración lo suficientemente capaces, volver a la configuración inicial deseada de la aplicación, luego de que la funcionalidad volátil tenga que ser removida, puede ser una pesadilla.

En los trabajos [RNM+06, RGDU08, GDRU09] se remarca la importancia de la integración de funcionalidad volátil maximizando el concepto de "Obliviousness "por parte de los componentes core de la aplicación. Los autores definen obliviousness, en el contexto estudiado, como la falta de conocimiento específico de relaciones entre componentes. Esto se traduce en que los componentes core sean agnósticos o ajenos a la existencia o inexistencia de componentes volátiles. En esta tesis se busca, además, maximizar el grado de obliviousness por parte de los componentes de funcionalidad volátil para aumentar la posibilidad de reutilización (Ver 4.2.4).

Independientemente del nivel de dependencia de las funcionalidades con respecto la aplicación host, el diseño de las mismas no debe ser limitado por incapacidades de la arquitectura a la hora de incorporarlas en la aplicación core. Por ejemplo, si la arquitectura utilizada no posee la capacidad de transportar información desde la aplicacion host hacia las funcionalidades volátiles, el diseño de las mismas se verá acotado por dicha imposibilidad. De la misma manera pero

en menor grado, el proceso de implementación debe ser idealmente independiente y transparente, siendo nuevamente la arquitectura un incidente directo. Al igual que los trabajos estudiados, en esta tesis se buscó lograr una solución que permita el diseño e implementación de los componentes de manera separada, buscando una integración posterior transparente.

Se recomienda la utilización del patrón MVC[Ree79, Ree03] en el diseño de los distintos componentes. Este patrón es útil para lograr modularidad, flexibilidad y claridad en el diseño e implementación de las aplicaciones web. La utilización de MVC es algo que puede resultar útil pero no aplicable a todos los contextos, frameworks de desarrollo web, etc, por lo que no es un requerimiento excluyente para el uso de la arquitectura propuesta. Este trabajo fue desarrollado utilizando Seaside, que permite el desarrollo de aplicaciones MVC de forma cómoda. La arquitectura trata de atacar el tema a un nivel más general para lograr una metodología de diseño que permita la incorporación de nuevas funcionalidades en aplicaciones existentes. Por lo tanto no se encuentra restringida al uso de MVC, aunque como veremos a continuación, su utilización resulta muy conveniente.

La idea es que en primer lugar se desarrolle la aplición core o principal, sin tener en cuenta las posibles funcionalidades volátiles. De utilizar MVC como base para el desarrollo, seguramente se utilizarán mecanismos de observers para lograr actualizaciones de la vista. De esta forma la aplicación core ya contará con ciertos puntos donde se anunciarán cambios. Más adelante se explicará cómo esto es importante para la posterior incorporación de nueva funcionalidad. Resumidamente, las funcionalidades volátiles estarán atadas a ciertos eventos de la aplicación host para poder recibir información de la misma. A medida en que la aplicación host se ejecuta y se modifica su estado, las funcionalidades volátiles pueden mantenerse actualizadas, recibiendo información constantemente a partir de eventos de la aplicación host.

4.2.2. Integración Transparente

La integración transparente tiene que ver con el impacto de la incorporación de la nueva funcionalidad en el diseño y en el código de la aplicación host y de la misma funcionalidad a introducir. La integración será completamente transparente cuando justamente, no produzca alteraciones en los componentes. Con las continuas adiciones de funcionalidad y su respectiva eliminación posterior, el código se puede volver cada vez más complejo y dificil de mantener. La posibilidad de llegar a un diseño conocido como Big Ball of Mud [FY97] está latente si la nueva funcionalidad no es integrada y removida de forma transparente. Si es necesario alterar el diseño y el código de la aplicación host cada vez que se incorpora o remueve una nueva funcionalidad, se introducen efectos secundarios, posibles bugs y se encarece el proceso de mantenimiento. Entonces, se busca transparencia en la integración para lograr un sistema más robusto, mantenible y extensible.

4.2.3. Comunicación bidireccional.

Cierto tipo de funcionalidad necesita comunicarse con la aplicación host para invocar un servicio o para procesar información. En tal caso, la funcionalidad volátil conoce a la aplicación host y se comunica con la misma utilizando una capa de servicios, o está más cercamente acoplada y puede interactuar directamente con componentes de la misma. En [RNM+06, RGDU08] se estudia este tipo de funcionalidad y se presentan ejemplos. Este tipo de funcionalidades, posee un esquema de comunicación de un solo sentido: desde la funcionalidad volátil hacia la aplicación host. Existe otro tipo de funcionalidad, que fue mencionado anteriormente, donde este esquema de comunicación no es adecuado o suficiente. Ciertas funcionalidades necesitan que la aplicación host notifique los eventos que acontecen para poder reaccionar de alguna manera. Entonces la comunicación, o el flujo de información, posee el sentido inverso que la anterior: desde la aplicación host hacia la funcionalidad volátil. Los side games son un ejemplo de funcionalidad que necesita utilizar este esquema de comunicación, como también el primero.

4.2.4. Funcionalidad volátil como elemento reutilizable

Para poder acoplar funcionalidad volátil en una aplicación, se deben cumplir ciertas condiciones que luego se explicarán con detalle. Dichas condiciones no son independientes de la aplicación host donde correrá la funcionalidad volátil ya que por supuesto, no siempre será posible acoplar funcionalidad en cualquier aplicación. No obstante esto, es necesario tratar de maximizar la independencia para aumentar la reusabilidad de la funcionalidad volátil.

Asumiendo que la nueva funcionalidad volátil es diseñada sabiamente, podría ser utilizada en otros contextos de la misma o de otras aplicaciones. Entonces, distintos desarrolladores pueden trabajar en distintos concerns de forma independiente, evitando conflictos entre ellos. Aún más, otros componentes, desarrollados por tenreceros con la misma filosofía, podrían ser más fáciles de integrar. “Tal es el caso de Google Maps, que fue incorporado en muchas aplicaciones web que no son parte de Google. ”[GDRU09]. Contar con componentes reutilizables disminuye potencialmente los costos futuros de desarrollo.

Algunas funcionalidades son tan particulares de una aplicación host que nunca podrían ser incluidas en otro contexto. La reutilización de la funcionaldiad volátil, al igual que el diseño e implementación independiente, tienen solo un cierto alcance. Esto se debe a que siempre será necesario cumplir con las reglas necesarias para acoplar la funcionalidad en la aplicación host. De esta forma se restringe el espectro de aplicaciones host y funcionalidades que se pueden integrar. Entonces, la funcionalidad volátil como elemento reutilizable, está limitada a las reglas de integración, que determinan el tipo de aplicaciones donde pueden

integrarse.

Estos son los cuatro ejes que se siguieron a lo largo del desarrollo y que fueron dando forma a la arquitectura que se presentará abajo. Como mencioné anteriormente, estos conceptos friccionan constantemente y es necesario encontrar un balance entre ellos.

4.3. Outline

A continuación se explican a grandes razgos los componentes que integran la arquitectura desarrollada. Se tratará de dar una visión general de la misma mediante la explicación de sus partes. Luego se introducirá en detalle cada componente, incluyendo una explicación completa del objetivo y de la forma de utilización. También se incluye un análisis, explicando ventajas y desventajas de las alternativas elegidas en cada parte de la arquitectura.

En el nivel más general, de componentes de la arquitectura, se distinguen tres partes: core, volatile y una capa intermedia de integración (ver figura 4.1). Esto no difiere del trabajo estudiado, mencionado ya varias veces. Considerando la pauta de desarrollo independiente, no es extraño que la arquitectura presente por separado las partes core de las volátiles. Al separar las funcionalidades volátiles de la aplicación core, dejando en una capa intermedia los aspectos de integración, se busca aumentar la capacidad de diseñar por separado. Luego, el diseño y las capacidades que ofrezca la capa intermedia serán determinantes para lograr integración transparente de las funcionalidades.

Los componentes core y volátiles, están diseñados de acuerdo al patrón MVC. Como fue explicado anteriormente, MVC además de desacoplar modelo y vista de cada parte, facilita mecanismos de integración entre las partes debido al mecanismo de observers que utiliza. Con esto se pueden definir eventos que sirvan para comunicación entre parte core y volátil.

Lo más relevante de esta arquitectura es la capa de integración, que será clave para determinar la forma en que las funcionalidades se acoplarán e interactuarán con la aplicación host. Las diferencias con el trabajo [RNM+06, RGDU08], son sutiles y en general referentes a cuestiones de implementación visibles en la capa intermedia.

Finalmente, la arquitectura cuenta con componentes que proveen mecanismos de instalación de la funcionalidad volátil. Estos componentes pueden variar con distintos casos de uso de esta arquitectura, pero en general, se encargarán de instanciar los artefactos necesarios de la capa intermedia, para que las funcionalidades puedan ser integradas.

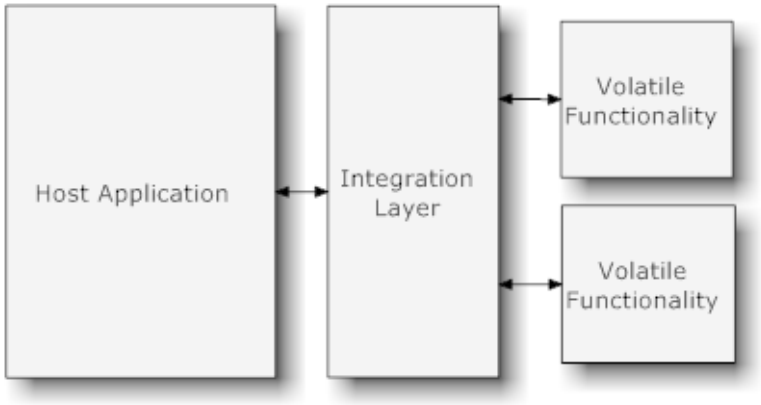

Figura 4.1: Componentes de la Arquitectura. Nivel de granularidad Alta.

4.4. Capa de integración

La capa intermedia provee mecanismos para lograr comunicación entre los componentes core y volátiles. Permite desacoplar de ambas partes los elementos que sirven para lograr la integración. El objetivo es que dicha integración sea transparente, dejando a los componentes core y volátiles lo más cohesivos e independientes posible. Se busca no contaminar el código y diseño de las partes, permitiendo desarrollar por separado las funcionalidades volátiles y la aplicación host.

Para acoplar funcionalidad volátil en una aplicación host, se deben implementar reglas de activación, actualización y mecanismos de comunicación. Estos elementos no pertenecen a los componentes core ni volátiles de la arquitectura, pertenecen a la capa de integración. Estas reglas y mecanismos de comunicación, serán implementados en los elementos de la capa de integración. La misma funcionalidad volátil puede ser luego acoplada en otra aplicación host, que necesite de distintas reglas de activación y actualización. En ese entonces, el costo de acoplar dicha funcionalidad se reducirá al costo de implementar nuevas reglas en una capa separada.

La capa intermedia consta de mecanismos de actualización y comunicación entre los modelos y vistas de las partes. Está constituida por 4 partes: Sistema de Eventos, Condiciones, Updaters y Pointcuts. A continuación se describen sucintamente las 4 partes que constituyen a la capa intermedia. Más adelante se explicarán en detalle.

4.4.1. Sistema de eventos

Anteriormente se mencionó la importancia de la comunicación bidireccional para cierto tipo de aplicaciones y funcionalidades volátiles. Una forma de man-

tener comunicación es la propuesta por [RNM+06, RGDU08], donde la funcionalidad volátil está constituida por una serie de decorators y commands sobre la aplicación core. En este esquema, la parte volátil conoce a la parte core y dicha relación de conocimiento es instanciada mediante dependency injection. Dependency injection [Fow04] permite reducir la dependencia entre los componentes y mejorar la reusabilidad de los mismos. No obstante esto, con este esquema de comunicación todavía sigue insatisfecha la necesidad de la funcionalidad volátil de recibir información cuando ocurre algo en la parte core.

Para solucionar el problema recién mencionado, se propone un esquema de comunicación basado en el pattern Observer [GHJV94]. Cuando un evento ocurre en alguna de las partes (core o volátil), los cambios son anunciados y mediante observers los objetos pertinentes son notificados. De esta forma se logra comunicación bidireccional y desacople entre la aplicación host y las funcionalidades volátiles. Particularmente, las funcionalidades volátiles pueden ser notificadas de los cambios que acontecen en la aplicación core. Para utilizar este esquema es necesario definir un conjunto de eventos que deben ser anunciados por cada parte en instancias determinadas de la ejecución del programa. Una vez determinados los eventos que cada participante anuncia y necesita del otro, se pueden determinar las acciones a ejecutar. Estas acciones estarán supeditadas al cumplimiento de ciertas condiciones, que tendrán que definirse y se evaluarán en el momento en que los eventos sean notificados. Así, el sistema de eventos aquí propuesto permite al programador definir puntos en los cuales la aplicación host se comunicará con las funcionalidades volátiles y viceversa.

A través de este mecanismo, la capa intermedia cumple el rol de canal de comunicación entre las funcionalidades volátiles y la aplicación host. Los mecanismos de comunicación utilizan en conjunto con el sistema de eventos una libreria de condiciones que será explicada a continuación. El sistema de eventos es una parte importante de esta arquitectura y permite a las funcionalidades volátiles mantenerse informadas sobre cambios en la aplicación host y viceversa.

4.4.2. Condiciones

Cuando ocurre un evento en la aplicación host, es probable que la funcionalidad volátil reaccione solo si ciertas condiciones se cumplen. Lo mismo ocurre en el sentido inverso. Por lo tanto es necesario que se puedan especificar y evaluar ciertas condiciones para que, en conjunción con los eventos, se puedan definir puntos en los cuales ocurre la interacción.

Para poder definir condiciones que se dan en la aplicación host es necesario referirse a estados concretos de la misma y poder evaluarlos. Una condición, como es concebida en este trabajo, encapsula el acceso y la forma de evaluar el estado de la aplicación para determinar un resultado. Por ejemplo, una condición sobre el juego de slots implementado es PrizeWonCondition, que contiene lo necesario

para evaluar si el juego otorgó premios en el último spin. Una vez programada, esta condición se puede utilizar en el contexto de un evento, por ejemplo SpinDone, para evaluar si un side game debe ser activado o no. De esta forma, se pueden especificar aserciones complejas sobre el estado de la aplicación host o de una funcionalidad volátil y encapsular dichas aserciones en un elemento concreto que tiene mayor significado, es más facilmente referenciable y reutilizable.

Las condiciones aquí presentadas, al igual que los eventos y pointcuts, son artefactos que elevan el nivel de abstracción y permiten integrar más facilmente las funcionalidades. El programador puede epecificar situaciones complejas utilizando elementos de alto nivel de abstracción. La utilización de Condiciones facilita la reutilización de elementos de integración de funcionalidad. Por ejemplo, un conjunto de condiciones sobre una aplicación host puede constituir una librería que puede ser compartida por el proceso de integración de distintas funcionalidades volátiles.

Ejemplo: La condición GameHasWonPrizes indica si el juego ha ganado premios en el último spin. Este objeto puede ser utilizado por la capa intermedia sin necesidad de acceder a la parte core directamente, logrando mayor modularidad. Más aún, esta condición puede formar parte de una librería que puede ser compartida por distintas funcionalidades volátiles.

La utilización de condiciones permite escribir reglas de activación, o de actualización de manera más intuitiva y clara. Junto con los eventos y Pointcuts es posible especificar momentos en la ejecución del programa para establecer políticas de actualización o activación, utilizadas por updaters.

4.4.3. Updaters

La capa intermedia provee mecanismos para especificar las reglas de activación correspondientes al patrón de volatilidad utilizando artefactos denominados Updaters. Los mismos poseen lógica para reaccionar a partir de un conjunto de condiciones que se evaluan cuando ocurre un evento determinado. Encapsulan el comportamiento de integración dada una situación específica del estado de la aplicación host o de la funcionalidad volátil. Por lo que sirven no solo para especificar reglas de activación, sino también reglas de actualización. Por ejemplo, un updater puede especificar cómo actualizar el juego principal cuando un sidegame finaliza su ejecución. Otro updater puede específicar cómo activar un side game cuando el juego entra en un estado determinado.

Los updaters encapsulan políticas de activación o reglas de actualización de la funcionalidad volátil. Utilizan el sistema de eventos y condiciones para poder especificar las reglas de actualización o activación. Sirven para mantener actualizadas las vistas y modelos de los componentes. Un updater utiliza un evento para determinar el momento en el cuál se evaluará un conjunto de condiciones para posteriormente ejecutar una acción. Así, cuando ocurre algún cambio en la

funcionalidad volátil que debe ser reflejado en el modelo o en la vista de la aplicación core (o viceversa), los updaters correspondientes se encargarán de realizar la actualización pertienente.

De acuerdo al impacto de la acción a ejecutar, los updaters se dividen en dos taxonomías. La primera diferencia el componente de la arquitectura que el updater afecta, core o volátil. Si las acciones asociadas al updater afectan a la aplicación host, el updater pertenece al tipo CoreUpdater, de lo contrario será un VolatileUpdater. La segunda taxonomía corresponde a la parte del componente que el updater actualiza, la vista o el modelo. Un updater puede simplemente actualizar un widget de la interfaz de la aplicación o ciertas variables del modelo de la misma. Por ejemplo, la capa de integración del side game doble o nada posee un updater que actualiza la vista del juego principal cuando el side game finaliza, eliminando todos los widgets correspondientes a los premios del juego host e introduciendo información del resultado de la apuesta doble o nada.

Las taxonomías recién mencionadas surgieron a partir de la construcción de distintas acciones de actualización, donde fue paulatinamente necesario realizar separaciones para mantener claridad. Fue útil la separación para poder comprender la manera en que las funcionalidades impactan en la aplicación host y viceversa. Esta forma de clasificar, permite encapsular las políticas de actualización y activación de manera más clara, facilitando la implementación de los mecanismos de integración.

Al utilizar estas taxonomías: modelos/vistas y core/volatile, se logran separar las reglas de integración y obtener una mayor comprensión de cómo interactúan los componentes core y volátiles. Toda la lógica de comunicación entre los componentes se especifica en lugares bien definidos de acuero a el cómponente donde se efectúan los cambios: core o volátil, y de acuerdo a la parte de dicho componente: modelo o vista.

Los Updaters generan mayor claridad en la implementación de las políticas de integración. El conjunto de updaters definido para una funcionalidad volátil, permite encontrar rápidamente cuáles los impactos de la misma en la aplicación host, ya que comprende un lugar común donde buscar las modificaciones introducidas con la funcionalidad.

4.4.4. Pointcuts

Como complemento de los updaters, se provee otro conjunto de artefactos, los Pointcuts. El concepto Pointcut proviene de AOP. En tal contexto, un pointcut permite describir un punto en la ejecución de un programa y asociar ciertas acciones que se ejecutarán en ese momento [KHH+01]. Concretamente un pointcut asocia un conjunto de condiciones que se evalúan en un momento dado con un conjunto de acciones que se ejecutarán solo si las condiciones se cumplen. Al conjunto de acciones se lo denomina advice y a las condiciones junto con el momento

en que se evalúan, se las denominan jointpoint.

En el contexto del sistema de eventos de esta tesis, explicado previamente, un Pointcut se configura con una serie de eventos que pueden disparar su ejecución. Así se determina el momento en que se evalúa el pointcut. Además de los eventos, se especifican cuáles son las condiciones a evaluar, conformando así el conjunto de joinpoints. Finalmente se asocia código a ejecutar, que constituye el advice. De esta manera, los pointcuts son instalados en el sistema y luego, cuando en el mismo ocurre alguno de los eventos correspondientes, el pointcut evalúa sus condiciones y posteriormente ejecuta su advice si fuera necesario.

En este trabajo, los pointcuts son utilizados por los updaters para especificar una política de actualización o de activación mediante la asociación de comportamiento con un momento en la ejecución del programa. De esta forma, permiten especificar con mayor claridad las reglas de activación de la aplicación, utilizando el sistema de eventos y condiciones para especificar un punto en la ejecución de la aplicación core. Este sistema permite al programador especificar las situaciones en las cuales la funcionalidad volátil se debe activar o desactivar. Por ejemplo:

```
CWSideGameUpdater>>deactivationPoints
^Array with: (VF.GamePointcut
            game: self game
            sideGame: self sideGame
            events: Slots.SLSpinDone , Slots.SLReSpinDone
            condition: VF.SideGameEnabledCondition)
    with: (VF.SideGamePointcut
            game: self game
            sideGame: self sideGame
            events: SG.SideGameFinished
            condition: VF.NoCondition)
```

En este ejemplo, el updater del side game card war, especifíca las políticas de desactivación del side game. Se pueden observar ver dos Pointcuts, uno sobre la aplicación core (`VF.GamePointcut`) y otro sobre la funcionalidad volátil (`VF.SideGamePointcut`). El primer pointcut se evaluará cuando alguno de los eventos `Slots.SLSpinDone` o `Slots.SLReSpinDone` sea anunciado. Cuando esto ocurra, se evaluará la condición `VF.NoCondition`, que siempre se evalúa positivamente. Luego, estos pointcuts son instalados con el código necesario para desactivar el side game (el advice):

```
VF.SideGameUpdater>>installDeactivationPointcuts
self deactivationPoints do:
        [:each | self installPointcut: each
                    withAdvice: [self sideGame disable]]
```

En este trabajo, fue útil implementar directamente pointcuts para reunir en un mismo objeto, estos conceptos relacionados, eventos, condiciones y acciones. Así, existe una jerarquía de Pointcuts de acuerdo al target de los mismos. El target es el objeto sobre el cuál se evaluarán las condiciones del pointcut y puede ser: la aplicación core o las funcionalidades volátiles. Además de proveer directamente objetos que representan pointcuts, el concepto también es utilizado de manera menos explícita, pero manteniéndose presente en los updaters (Ver sección 5.4). Por ejemplo:

```
DONGameUpdater>>installPointcuts
self
    when: self sideGame
    triggers: SG.SideGameFinished
    underCondition: VF.NoCondition
    do:[self game emptyPrizes.
        self sideGame hasWon
            ifTrue: [self game addCredits: self sideGame finalCredits]]
```

Este ejemplo muestra el manejo implícito del concepto, sin utilizar objetos concretos de la jerarquía de Pointcuts. El `GameUpdater` del side game doble o nada especifica la lógica de actualización del juego, cuando el side game finaliza bajo cualquier condición. Se pueden ver claramente los componentes del pointcut: El evento que dispara la ejecución del pointcut (`SG.SideGameFinished`) , la condición que se evalúa (`VF.NoCondition`) y el advice que se ejecuta:

```
 [self game emptyPrizes.
self sideGame hasWon
      ifTrue: [self game addCredits: self sideGame finalCredits].
```

4.5. Proceso de de instalación

La funcionalidad volátil, al igual que los objetos de la capa intermedia - updaters, pointcuts, condiciones y demás- deben ser instanciados y acoplados a la aplicación core de alguna manera. Estas actividades se dan en el proceso de instalación de la funcionalidad volátil. En el mismo, la funcionalidad debe ser instanciada y configurada con los elementos que sirven para integrarla a la aplicación core. Es necesario crear los updaters que servirán para mantener la comunicación entre ambas partes. Así mismo, los updaters necesitan de pointcuts, para especificar el momento y el comportamiento de actualización, que también deberán ser instalados. Como se mencionó anteriormente, los pointcuts necesitan que se anuncien determinados eventos, tanto en la aplicación principal como en la funcionalidad volátil. De esta forma, el proceso de instalación consiste en una

primera parte: en la instanciación de los elementos de la funcionalidad volátil y de todos los elementos de la capa intermedia necesarios para que se logre el acoplamiento. La segunda parte del proceso de instalación consiste en la conexión de los elementos de la capa intermedia con los elementos de la aplicación host y de la funcionalidad volátil.

Otra de las actividades que deben llevarse a cabo durante el proceso de instalación, es forzar la compatibilidad entre la aplicación host y la funcionaldiad volátil a instalar. Esto significa que todos los eventos necesarios para la integración deben sean propiamente anunciados en la aplicación host y en la funcionalidad volátil. Puede ocurrir que haya una diferencia entre los eventos requeridos por la capa de integración de la funcionalidad volátil y los eventos que la aplicación host anuncia. Así, es necesario traducir los eventos que produce la aplicación host, en eventos que necesita la funcionalidad volátil. En la sección de explicación de esta parte de la arquitectura se mostrará que, de la misma manera que en [RNM+06], también es necesario decorar componentes de la aplicación core para alterar su comportamiento. Ya se explicó que idealmente este proceso debería ser llevado a cabo sin realizar modificaciones intrusivas en el código o el modelo de la aplicación host. Siguiendo esta última idea se presentará una aproximación para solucionar este problema, que utiliza mecanismos de AOP (Method Wrappers) para alterar el comportamiento de la aplicación host sin contaminar su código.

El proceso de instalación de la funcionalidad volátil está compuesto por las tareas de instanciación de los elementos necesarios y por la modificación de comportamiento de partes de la aplicación host. Estos procesos deben ser llevados a cabo en algún momento, que está determinado por el patrón de conexión de la funcionalidad volátil. Este atributo de la funcionalidad volátil, descrito en la sección 2.2.2, define las reglas bajo las cuales se debe instalar la misma. En un entorno ideal, este atributo se podría incluir en el sistema para que el proceso de instalación se pueda realizar de forma automática, disparado por reglas de negocio, temporales y demás, especificadas en el patrón de conexión. En este trabajo no se llegó a tal punto de automatización, sino que se implementó una solución más cercana a la realidad. Se implementó una herramienta que permite la instalación dinámica de la funcionalidad volátil, de acuerdo a ciertas reglas diseñadas e implementadas en el sistema. La herramienta permite al operador instalar side games en el juego principal de manera fácil y en tiempo de ejecución, mientras el sitio sigue operando. La herramienta fue desarrollada gracias a la utilización de esta arquitectura y permite la incorporación de funcionalidades volátiles sin alterar el código de la aplicación principal, permitiendo restaurar el comportamiento de la misma en cualquier momento, en tiempo de ejecución.

El proceso de instalación recién mencionado y los artefactos que son utilizados en el mismo, se pueden ver fuera de la capa intermedia, como un accesorio de la misma. Las reglas de instalación de la funcionalidad, pueden depender del estado

de la aplicación host, por lo que el proceso de instalación puede considerarse un cliente de la capa de integración, que utiliza sus mecanismos de comunicación a través de eventos de la aplicación host.

4.6. Resumen de la arquitectura

Hasta aquí se presentaron los rasgos generales de los componentes principales de la arquitectura propuesta en esta tesis. La misma permite el desarrollo por separado de componentes volátiles y de la aplicación host, permitiendo la integración de estas partes de forma transparente. Está basada en una capa intermedia que brinda los mecanismos de acople y comunicación que permiten la interacción de los componentes volátiles con la aplicación host. También forma parte de esta arquitectura, aunque de manera menos crucial un mecanismo de instalación de componentes volátiles que se puede ver en este trabajo en una herramienta concreta. La figura 4.2 muestra un diagrama completo detallado de la arquitectura

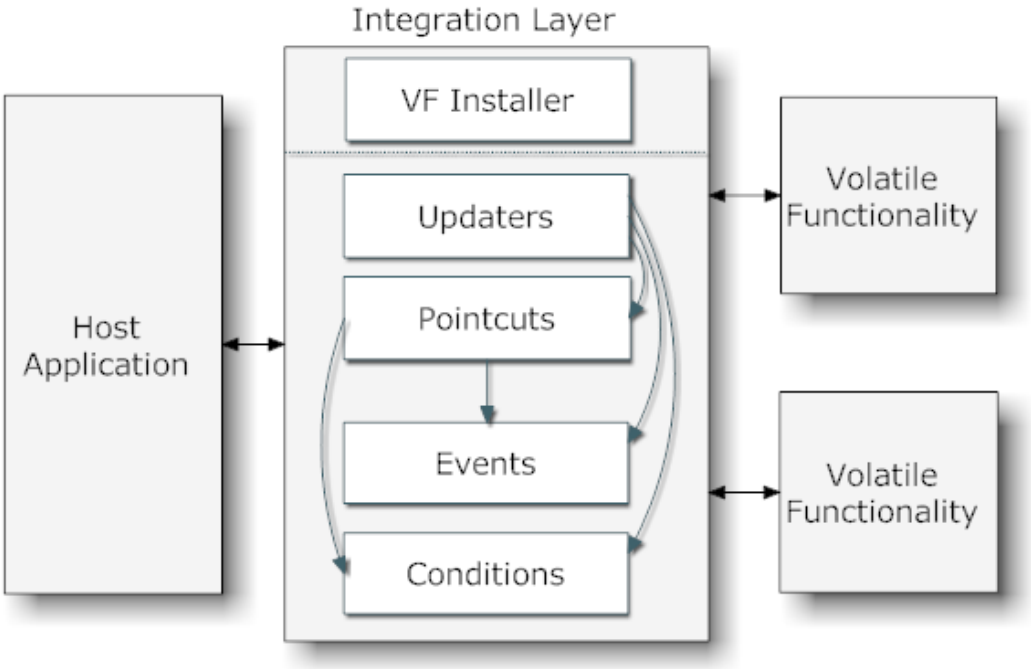

Figura 4.2: Diagrama de la arquitectura

Los updaters comprenden el elemento más importante de la capa intermedia, son responsables de llevar a cabo la comunicación entre las partes mediante la actualización de sus modelos y vistas a partir de reglas predefinidas. Los updaters utilizan el sistema de eventos para especificar el momento en que se realiza la

actualización de un componente. En el momento determinado por el anuncio de un evento, se evalúa una serie de condiciones, que de ser favorable generará la ejecución de las actividades de actualización del updater. Estas condiciones son acumuladas en una librería que puede ser dependiente de un side game o de la aplicación host. Los Pointcuts también pueden ser utilizados por los updaters para especificar el momento, condiciones y acciones a realizar para actualizar componentes. Finalmente el mecanismo de instalación de todos estos elementos comprende el último elemento de la capa de integración.

En el siguiente capítulo se muestra en detalle cada uno de los componentes de la arquitectura.

Capítulo 5

Arquitectura en Detalle

5.1. Introducción

En esta sección se explicarán detalles de la arquitectura presentada en el capítulo anterior. En el mismo se explicaron los aspectos macro de la arquitectura para permitir al lector una mayor comprensión de su funcionamiento y de cómo interactúan sus componentes. A continuación se presenta una explicación detallada sobre los aspectos de la arquitectura presentados anteriormente. Si bien se trata de brindar información concerniente a la arquitectura en cuestión, es inevitable en ocasiones mencionar aspectos intrínsecos de la implementación desarrollada. Si bien estos aspectos no son determinantes para la implementación de la arquitectura en otros contextos, son importantes para el mayor entendimiento de la misma.

5.2. Sistema de eventos.

5.2.1. Introducción

Ciertas funcionalidades volátiles (los side games particularmente) requieren de información de la aplicacion principal en el momento en que dicha información se genera. Por ejemplo el side game doble o nada, necesita saber en el momento en que el juego principal realiza spin, si se entregó o no un premio para poder iniciar su ejecución.

Para integrar este tipo de funcionalidades no basta con proveer solamente un mecanismo para consultar a la aplicación host, utilizando Decorators[GHJV94], como se muestra por ejemplo en [RNM+06]. En dicho trabajo, Rossi et. al. presentan un mecanismo de comunicación en un solo sentido basado en Commands y Decorators. Con esto los autores logran un sistema que provee comunicación en

un solo sentido, desde la funcionalidad volátil hacia la aplicación host. A partir del desarrollo de esta tesis, se encontró que lo anterior resulta insuficiente para el tipo de aplicaciones que aquí se tratan. Esto se debe a que la comunicación debe ser también en sentido inverso. Para que las funcionalidades volátiles se enteren de lo que ocurre en la aplicación principal, la misma debe notificar sus cambios de alguna forma.

El problema mencionado en el párrafo anterior es bien conocido y una posible solución es utilizar el patrón Observer[GHJV94]. De esta forma se logra implementar la forma de comunicación deseada evitando incluir lógica de comunicación hacia la funcionalidad volátil en la aplicación host. En esta tesis se utilizó un sistema de eventos basado en este patrón para solucionar la comunicación de dos vías, o como se menciona en [BVD07] "integración de dos vías ". La funcionalidad volátil es observer de la aplicación principal y la aplicación principal observer de la funcionalidad volátil. De esta manera se logra mantener los componentes desacoplados, pero a su vez comunicados mediante los mecanismos de Observers mencionados.

El panorama general es como se explicó en el parrafo anterior. Es necesario observar cambios en la aplicación host desde la funcionalidad volátil y viceversa. Una aproximación directa, para implementar lo recién mencionado, consistiría en implementar la funcionalidad volátil y la aplicación host como observers entre sí de forma directa, de acuerdo al patrón Observer. El inconveniente de esta aproximación es que para lograr tales relaciones sería necesario alterar el comportamiento de ambas partes. La aproximación seguida en este trabajo trata de solucionar este inconveniente, evitando la edición intrusiva de cualquiera de las partes.

En este trabajo se propone que, ni la parte core ni las partes volátiles sean observers entre y se introduce una capa intermedia que cumple esta función. Esta capa está compuesta por objetos que cumplen el rol de observadores y saben cómo reaccionar a partir de la ocurrencia de un evento de la parte core o de las funcionalidades volátiles. La capa de integración está compuesta por observers que son instanciados en el momento de la instalación de la funcionalidad volátil. Los mismos están configurados para poder reaccionar a determinados eventos de las distintas partes y así mantener actualizados los modelos y las vistas. De esta forma se logra comunicación bidireccional entre la aplicación principal y las funcionalidades volátiles. Es así, que los eventos que la aplicación host y la funcionalidad volátil anuncian y necesitan son de vital inportancia para la correcta integración de la funcionaldiad volátil.

5.2.2. Sistema de eventos para la comunicación de modelos.

Los eventos hasta ahora mencionados se encuentran en el marco de un sistema que utiliza el framework Announcements [Byk05] de Smalltalk. Este framework

fue utilizado en el desarrollo de esta tesis para representar eventos y poder manejar el proceso de comunicación entre partes debilmente conectadas. En dicho framework los objetos pueden realizar anuncios utilizando instancias de la clase Announcement o alguna de sus subclases. Cada subclase de Announcement define un evento que puede ser anunciado. Por ejemplo el juego de Slot, dispara el anuncio SpinDone (subclase de Announcement) cuando el usuario hace click en el boton spin.

A grandes rasgos, el sistema implementado funciona de la siguiente manera: la aplicación principal anuncia cambios a través de Announcements y las partes volátiles reaccionan cuando tales cambios ocurren. Lo mismo ocurre en el sentido inverso. Es necesario, por supuesto, programar en la funcionalidad volátil el comportamiento esperado como reacción a los eventos de la parte principal. Este comportamiento es programado como parte de un handler del anuncio en cuestion. Un ejemplo de la utilización de esto se puede ver en la figura 5.1.

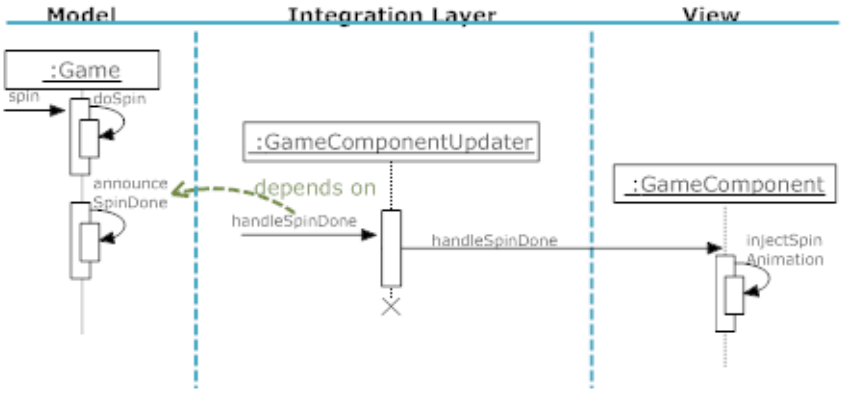

Figura 5.1: Ejemplo del anuncio de un evento y del manejo del mismo (Spin).

5.2.3. Sistema de eventos para la comunicación de vistas.

Las aplicaciones web implementadas y estudiadas en esta tesis, por su naturaleza, necesitan de coordinación entre los modelos y las vistas para mantener consistencia y aún así presentar al usuario información que en algunos momentos es "falsa". Esto significa, que en general, cuando un juego de casino es ejecutado, en el servidor se computan los resultados y se pagan los premios correspondientes al instante. No obstante, el jugador visualiza animaciones que pueden durar segundos y dan la impresión de cierta mecánica de juego que necesita tiempo para ejecutarse. Durante el tiempo que duran las animaciones, la información que percibe el jugador en la vista no es consistente con la información almacenada en los modelos subyacentes, inmediatamente actualizados en el momento de la jugada. Es por eso que este tipo de aplicaciones necesitan de coordinación entre

los modelos y las vistas.

Al integrar funcionalidad volatil en una aplicación de la naturaleza mencionada en el párrafo anterior, es necesario establecer comunicación a nivel de modelos y a nivel de interfaz gráfica entre la aplicación host y las funcionalidades volátiles. Cuando un side game es ofrecido en conjunto con un juego principal, es necesario que ciertos cambios que ocurren en el juego principal sean reflejados en la interfaz gráfica del side game de forma coordinada. Cuando ocurre algo en la interfaz de la aplicación principal (comienza una animación, por ejemplo) es necesario quizás que ocurra algo en la interfaz de las funcionalidades volátiles. Por ejemplo: el juego doble o nada se debe ofrecer cuando el usuario obtiene un premio en el juego principal. Por esto, cuando el juego principal muestra el premio obtenido, se debe presentar al usuario el ofrecimiento del doble o nada. Entonces, además de los mecanismos explicados para obtener comunicación entre modelos desacoplados, se necesitan mecanismos de comunicación y coordinación entre vistas.

Para que se vean reflejados los cambios en la vista de una RIA con ajax, en el browser se utilizan técnicas que emplean JavaScript y DOM para refrescar porciones del documento HTML en lugar del documento completo [Gar05]. En esta tesis se implementó un mecanismo que utiliza estas técnicas para la coordinación y comunicación entre la vista y el modelo de las aplicaciones desarrolladas. Se utiliza JavaScript y DOM para realizar las modificaciones necesarias para la actualización del documento HTML y la consecuente actualización de la vista de las aplicaciones. La técnica empleada consiste en la injección de código JavaScript que manipula el DOM del documento HTML para actualización por demanda de porciones de la vista. Para la actualización de una aplicación y la coordinación de su modelo y vista, se elige el momento adecuado para la injección de código en el script. Por ejemplo, cuando las animaciones pertinentes de una jugada finalizan, se ejecuta logica en el servidor que realiza las modificaciones necesarias en un script, que finalmente se ejecutará en el browser y actualizará el documento HTML para mostrar los resultados necesarios.

El mecanismo mencionado en el párrafo anterior fue empleado también para la coordinación y comunicación entre las vistas de la aplicación principal y de las funcionalidades volátiles. Se empleó el mismo framework de Announcements mencionado anteriormente, en conjunción con mecanismos JavaScript provistos por Seaside y Scriptaculous, para utilizar el mecanismo de scripts de manera coordinada. El objetivo fue injectar código JavaScript, en un script destinado a la actualización y coordinación de vistas, en el momento adecuado. Se implementó para esto una nueva taxonomía de subclases de Announcement que se instancian utilizando un script de Scriptaculous. Cuando un anuncio de tal taxonomía es recibido y manejado por un handler, se injecta en el script asociado al anuncio el código JavaScript inherente a la actualización. Como ya se mencionó, esto provee una manera adecuada de injectar lógica de actualización en el script,

en el momento adecuado. Es decir, en el momento en que se debe llevar a cabo la coordinación entre vistas.

Anteriormente se explicó el funcionamiento del side game doble o nada (Ver sección 3.5.2). Cuando el juego principal, sobre el cuál se instala el side game, ejecuta la acción Spin, se inicia una animación que simula los Reels de la Slotmachine girando. El side game doble o nada debe ser mostrado cuando el Spin finaliza en el juego principal. Esto trae aparejado al menos dos cuestiones de comunicación y coordinación. La primera corresponde a la comunicación entre modelos: el side game debe ser notificado en su modelo de la entrega de premios ocurrida en el juego principal (Esto ya fue explicado en la sección anterior, ver 5.2.2). La segunda cuestión se refiere a la coordinación de vistas: cuando el juego principal termina de actualizar los premios, posteriormente a la finalización de la animación del spin, se deben mostrar los componentes visuales del side game doble o nada. Con todo esto, es necesario coordinar la actualización de modelos y de vistas de forma ordenada. Con la utilización de mecanismos JavaScript, DOM y Announcements, explicado en los párrafos anteriores, se logra tal coordinación. En el ejemplo mencionado, la vista del juego principal injecta la lógica de renderización necesaria para mostrar los premios en un script. Luego de incluir dicha lógica en el script, se realiza un anuncio que contiene dicho script y avisa que los premios fueron renderizados en la vista del juego principal. Como consecuencia de dicho anuncio, se injectará en el script en cuestión el código necesario para añadir los componentes visuales del side game doble o nada. De esta forma, a través de injección de código JavaScript en el momento adecuado, se manipula el documento HTML, y se logra mantener coordinadas las interfaces de la aplicación web host y de las funcionalidades volátiles. Es esquema de la comunicación explicada en este ejemplo se puede ver en la figura 5.2:

5.2.4. El sistema de eventos en la capa de integración.

Si la funcionalidad volátil se diseña para acoplarse en una aplicación host conocida, es posible que el conjunto de eventos necesarios sea, desde el principio, anunciado por dicha aplicación. Eventualmente, la misma funcionalidad volátil puede ser necesitada en otro sistema, por lo que los eventos necesarios quizás no sean anunciados por el nuevo sistema. Para solucionar este problema resulta útil que el sistema de eventos resida en la capa intermedia y que la funcionalidad volátil dependa de dichos eventos y no de eventos de la aplicación host directamente. De esta forma, se pueden definir mecanismos para alcanzar compatibilidad entre la aplicación host y la funcionalidad volátil. Por ejemplo, el sistema de eventos puede adaptar los eventos del sistema host para que sean compatibles con la funcionalidad volátil.

Se explica a continuación un ejempolo de lo introducido en el párrafo anterior. La aplicación Slotmachine utiliza el evento `SpinDone` cuando los reels terminan

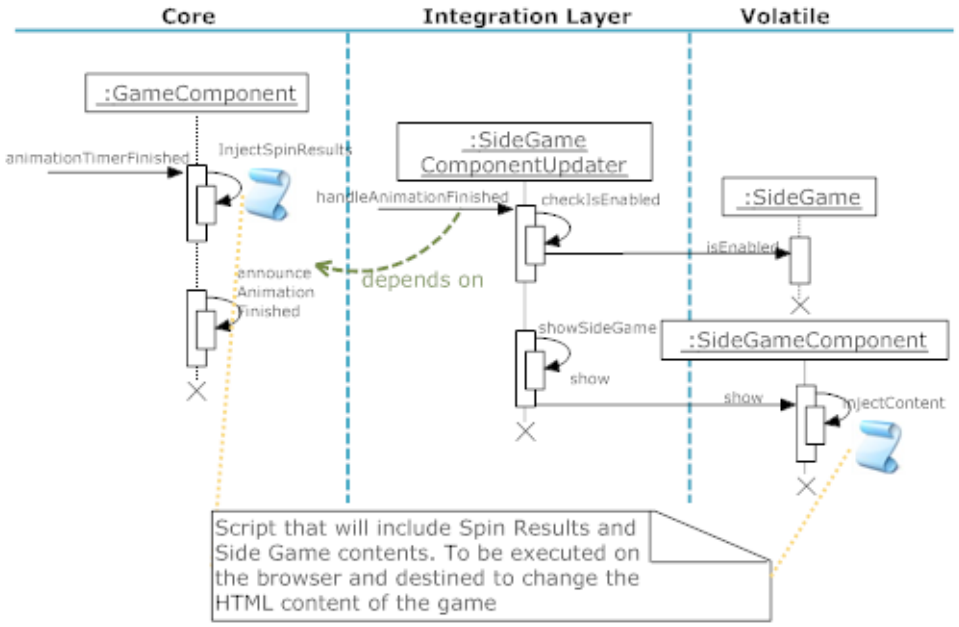

Figura 5.2: Ejemplo de un handler de evento de la vista (Spin)

de girar. El side game doble o nada necesita que la aplicación host anuncie el evento `PlayDone` para poder ejecutarse. De esta forma, la capa intermedia puede realizar la conexión entre estos eventos equivalentes, para alcanzar la compatibilidad requerida y lograr la comunicación. Entonces, cuando la aplciación anuncia el evento `SpinDone`, la capa intermedia lo atrapa y anuncia el evento `PlayDone` para que la funcionalidad volátil pueda ejecutarse. Esto se puede observar en la figura 5.3

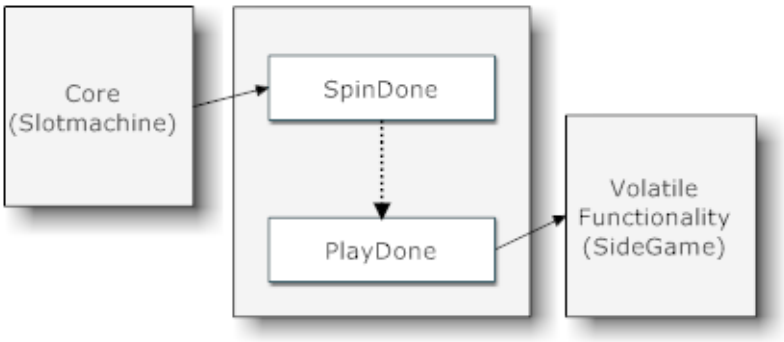

Figura 5.3: Traducción de eventos en la capa intermedia

Se necesita que la aplicación principal anuncie cada evento para que las funcionalidades volátiles puedan reaccionar. Esto presenta un problema ya que la aplicación principal no necesariamente dispara eventos como parte de su meca-

nismo interno. En el caso de la aplicación desarrollada para esta tesis, se utilizaron anuncios para poder mantener los componentes internos desacoplados desde un principio. Esto fácilitó mucho la incorporación de la funcionalidad volátil ya que para realizar la integración se utilizaron anuncios ya existentes. No obstante eso, en el caso donde la aplciación principal no utilize eventos será necesario efectuar algunas modificaciones en el código para lograr compatibilidad con la funcionalidad volátil a integrar. Más adelante (Ver apéndice A, compatibilidad mediante method wrappers) se propone una solución a este problema -incorporar anuncios de eventos alterando código de la aplicación host-.

5.2.5. Evaluación del sistema de eventos.

El sistema de eventos se puede comparar con un Rules Engine (Ver trabajo relacionado, sección 6.4). Cómo remarca Fowler en un artículo de su blog [Fow09]:

> *"Es fácil escribir un sistema de reglas pero muy dificil de mantener. En un sistema de este tipo, el flujo de programa no es fácil de entender. La lista de reglas es fácil de comprender y ver que tiene sentido, mientras que la interacción entre las reglas puede ser muy compleja. Cuando las acciones de una regla cambian el estado sobre el cuál se basan las condiciones de otra regla, se forma una relación compleja entre las dos reglas. A diferencia del modelo computacional imperativo, el flujo de programa generado por el sistema de reglas es difícil de comprender. Con un sistema de reglas, parece fácil llegar a un punto donde un simple cambio en un lugar causa muchas consecuencias inesperadas, lo que rara vez resulta bien."*

Lo que dice Fowler es innegable: el mantenimiento de un sistema de las características mencionadas es dificil. La complejidad del sistema desarrollado creció mucho al utilizar extensamente eventos para mantener los componentes desacoplados. El proceso de debugging se volvió más complejo. En algunos casos los efectos laterales son muy difícil de encontrar. Cuando las reglas están dispersas resulta extremadamente difícil entender el comportamiento del sistema. Estos aspectos negativos son verdaderos, pero también existen aspectos positivos por considerar. El sistema de eventos utilizado permite desacoplar componentes, obtener independencia y poder reutilizar partes, como side games. Esta fue la base para poder incorporar y remover funcionalidad volátil de forma dinámica. Permitió mantener separados los modelos y unirlos de forma dinámica.

Aunque aspectos de eficiencia escapan al alcance de este trabajo se puede mencionar al respecto que: resulta claro que la utilización de announcements, evaluación de condiciones y acciones trae complejidad computacional que en otros contextos podría considerarse prohibitiva.

5.2.6. Conclusión sobre el sistema de eventos.

Es uno de los mecanismos principales que en esta tesis se utilizaron para poder integrar modelos y vistas de una aplicación core con funcionalidades volátiles. Se mencionaron sus posibles riesgos y desventajas y se remarcó en la importancia del sistema para lograr desacoplar funcionalidades y aumentar la reusabilidad. El mecanismo de eventos y anuncios constituye uno de los pilares sobre los cuales se desarrolló la arquitectura aquí presentada.

5.3. Updaters

En la sección anterior se explicó como el mecanismo de eventos era utilizado en la capa intermedia para lograr comunicación entre la funcionalidad volátil y la aplicación host. En esta sección se explicarán los mecanismos que utilizan a los eventos y logran realizar finalmente las acciones de comunicación y actualización.

5.3.1. Introducción

El objetivo de la capa intermedia es lograr integración de la nueva funcionalidad manteniendo los componentes core y volátiles desacoplados, para minimizar los problemas mencionados anteriormente (Ver sección 2.4). La necesidad de mantener comunicación bidireccional entre los componentes core y volátiles es uno de los requisitos para la integración. Otro requisito es que dicha comunicación se logre manteniendo bao acoplamiento entre los componentes. Para lograr esto último pueden utilizarse mecanismos de observers para evitar el acoplamiento. Como se mencionó en la sección 5.2.1, la aproximación más simple, utilizando este tipo de mecanismos, consiste en implementar la funcionalidad volátil como observer de la aplicación host y viceversa. De esta forma se logra la comunicación deseada minimizando el acoplamiento entre las partes. El potencial problema de esta aproximación es que puede ser necesario alterar el código de alguna de las partes -core o volátil-. Por ejemplo, si la funcionalidad volátil es observer de la aplicación host, es necesario que los componentes de la última anuncien los cambios necesarios y que provean un protocolo esperado por la nueva funcionalidad. De no ser así, deberán realizarse modificaciones sobre la aplicación principal para que cumpla con tales requisitos. En caso de ser inaceptable la modificación de la aplicación host, se deberá modificar la funcionalidad volátil para que esté adaptada a los eventos anunciados por los componentes core.

El problema de la aproximación directa, recién explicado puede ser resuelto implementando los mecanismos de observers en la capa intermedia. Por supuesto, siempre será necesario que tanto los componentes core como volátiles anuncien ciertos eventos. Para el caso en que estos eventos no sean anunciados, se explica luego un mecanismo de modificación de comportamiento que permite solucionar

problemas de compatibilidad sin alterar el código original de las partes (Ver sección A). De esta forma, las potenciales modificaciones necesarias por problemas de compatibilidad serán implementadas en elementos de la capa de integración. Así, dejando intactos los componentes core y volátiles, se maximiza su reutilización y minimizan los impactos en el proceso de mantenimiento. En esta arquitectura, el papel de observers en la capa de integración es interpretado por los Updaters.

5.3.2. Definición

Los updaters son los elementos de la arquitectura que permiten especificar las políticas de actualización, reemplazando a las formas tradicionales de comunicación entre componentes mediante conocimiento explícito y envío de mensajes. Son parte de la capa de integración y definen las acciones de actualización a realizar a partir de una circunstancia dada en el programa principal o en la funcionalidad volátil. Dicha circunstancia está determinada a partir de un evento que ocurre en el componente observado por el updater. Así, como se menciono anteriormente, los Updaters cumplen el rol de Observers. Cuando ocurre un evento, el updater evalúa una serie de condiciones con las cuales fue configurado. Si las condiciones evaluadas cuando ocurre el evento son satisfactorias, las acciones de actualización son ejecutadas sobre el componente a actualizar. Estos elementos recien mencionados son las partes que componen un Updater en esta arquitectura: Evento, Condiciones, Acciones, Notificador del evento y Receptor de las acciones.

El updater es notificado de cambios que ocurren en el componente observado a traves de la recepcion de un evento disparado por dicho componente. Este componente, que puede pertenecer a la aplicación host o ser parte de la funcionalidad volátil, se denomina componente Notifier. Por supuesto, dicho componente debe ser un objeto capáz de anunciar eventos para cumplir su rol de notificador. A partir de los cambios anunciados por el Notifier, el udpater ejecutará sus acciones de actualización sobre otro componente; generalmente perteneciente a la capa opuesta. Tal componente, sobre el cuál el updater ejecuta sus funciones de actualización, es denominado componente Target. El target puede residir en la aplicación host o ser parte de la funcionalidad volátil de acuerdo al tipo de updater. Por ejemplo, para el side game card wars, existe un updater encargado de actualizar el modelo del juego principal cuando el side game entra en guerra y el jugador gana (ver sección 3.5.3 para conocer las reglas del juego). En la tabla 5.3.2, a modo de ejemplo, se muestran las partes involucradas en el updater recién mencionado.

En el ejemplo anterior se puede ver que el target del updater corresponde a un componente de la aplicación host (Slotmachine) y el notifier pertenece a un componente de la funcionalidad volátil (Side Game). En ese caso el updater está encargado de actualizar componentes de la aplicación host. También puede darse el caso contrario, donde el updater actualizará elementos de la funcionalidad

Componente	Instancia	Descripción
Notifier	CWSideGame	Componente de Funcionalidad volátil: parte del modelo del side game.
Target	Slotmachine	Componente de Aplicación host: parte del modelo de la aplicación host
Evento	CWWarDone	Evento de la funcionalidad volátil: que el side game entre en "guerra"
Condiciones	SideGameWon	Condición sobre la funcionalidad volátil: que el jugador gane el side game
Acción	GameCredits = GameCredits + SideGameWager * 3	Acción sobre la aplicación host: incrementar los créditos del juego

Tabla 5.2: Updater del modelo de la aplicación host para el juego card wars

volátil a partir de cambios que ocurran en componentes core. Sin importar el tipo de comunicación, generalmente, el notifier pertenererá a una parte (core o volátil) y el target pertenezca a su contraparte (volátil o core respectivamente). De esta forma los updaters actúan como canal de comunicación y elemento integrador de las funcionalidades volátiles y la aplicación host. Este mecanismo constituye una modificación del mecanismo de observers tradicional para lograr la modularidad deseada evitando los problemas de edicion intrusiva de codigo inherentes a la introduccion de la nueva funcionalidad.

Es necesario especificar las políticas de actualización de las distintas partes (modelo y vista de la aplicación host, modelo y vista de la funcionalidad volátil). Para esto se deben definir updaters concretos para cada parte que necesita ser actualizada. La actualización es consecuencia de acciones llevadas a cabo o en la aplicación host o en la funcionalidad volátil, según corresponda. Concretamente, por cada side game que se desea incorporar, es necesario especificar las acciones de actualización del modelo y de la vista tanto del side game como de la aplicación host. De esta forma se establece de manera completa cómo afectará el nuevo side game al sistema y viceversa.

5.3.3. Jerarquía de updaters

Como se mostró en el ejemplo anterior, los updaters pueden actualizar componentes del modelo de la aplicación host. También pueden especificarse updaters para actualizar modelos de la funcionalidad volátil a partir de cambios en la

aplicación host. Lo mismo ocurre con la vista, tanto de la aplicación host como de las funcionalidades volátiles que se incorporen. Pueden especificarse updaters que se encargarán de mantener actualizadas las vistas (definidas mediante componentes Seaside en este trabajo). Es así, que de acuerdo al componente Target, perteneciente a la vista vista o al modelo, se pueden clasificar los updaters en una primera taxonomía. Otra posible clasificación se obtiene al considerar si el componente target pertenece a la aplicación host o a la funcionalidad volátil. Entonces, se pueden distinguir dos posibles formas de clasificar a los updaters, dos taxonomías de acuerdo al componente Target del updater.

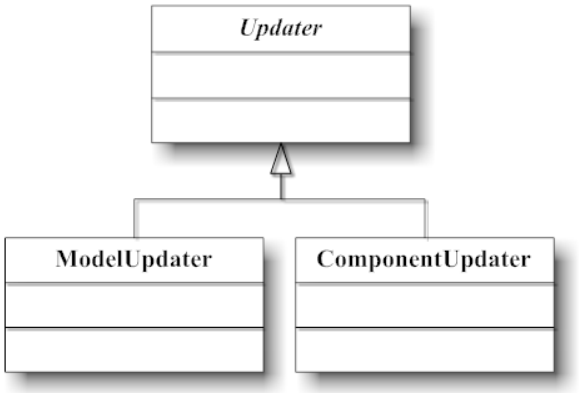

Figura 5.4: Primer Taxonomía de Updaters: Modelo vs. Vista

En la figura 5.4 se presenta el diagrama UML de la primer taxonomía mencionada. Para esta taxonomía, se considera el lugar que será actualizado en el modelo MVC: la vista o el modelo; ya sea de de la aplicación host o de la funcionalidad volátil. Los beneficios de contar con esta clasificación tienen que ver con el tipo de acciones de actualización y los elementos que dichas acciones involucran. Dada la naturaleza de los elementos a actualizar, esta clasificación resulta provechosa para mantener encapsulados en updaters de uso general ciertos elementos que son necesarios para llevar a cabo la actualización. Por ejemplo, para actualizar la vista de la aplicación host (o de la funcionalidad volátil) es necesario contar con un script que debe ser modificado con la lógica de actualización para que cuando el request finalice y se envíe el response, el browser muestre el contenido actualizado. Para conocer cómo los updaters de vistas actualizan el documento HTML dinámicamente ver la sección 5.2.3.

La segunda forma de clasificar a los updaters tiene que ver con una división más general, correspondiente a si la actualización se lleva a cabo sobre la aplicación host o sobre la funcionalidad volátil. En la figura 5.5 se muestra el diagrama UML correspondiente a este esquema de clasificación.

En este trabajo se utilizó una conjunción de estas dos taxonomías que permite construir updaters de modelo o de vistas de la aplicación principal o de la

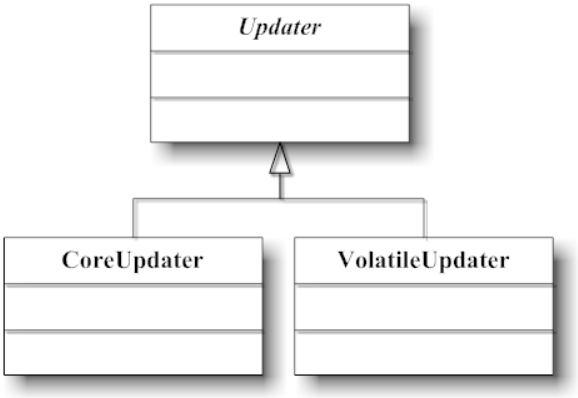

Figura 5.5: Segunda Forma de clasificar Updaters: Core vs. Volatile

funcionalidad volátil. Si bien la taxonomía resultante (ver figura 5.6) es bastante simple, incluso naive, se explica a continuación cómo resultó útil para el trabajo desarrollado.

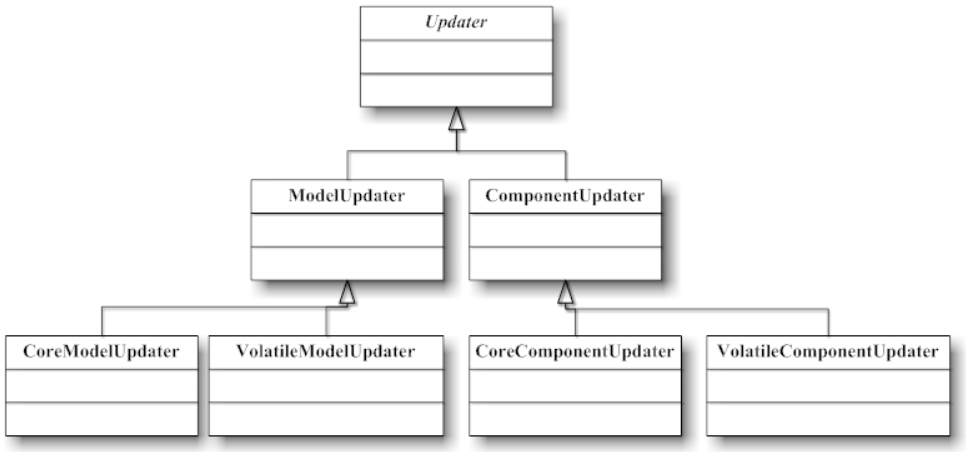

Figura 5.6: Combinación simple de las taxonomías.

La implementación de esta taxonomía permite al desarrollador especificar los procesos de actualización de manera organizada en lugares bien definidos. La incorporación de nueva funcionalidad volátil genera la necesidad de especificar mecanismos de actualización de distintas partes, como modelo y vista de la aplicación host, por ejemplo. La taxonomía presentada divide los lugares de actualización en cuatro: modelo-core, modelo-funcionalidad volátil, vista-core, vista-funcionalidad volátil. Al mantener bien definidas las partes que deben actualizarse y el lugar donde debe programarse la lógica de actualización correspondiente, se logra un proceso más controlado y sencillo. Utilizando estos mecanismos, es necesario especificar -en el peor caso- cuatro updaters que poseerán toda la lógica de ac-

tualización necesaria. Cada updater es encargado de actualizar una parte bien definida del sistema. La tabla 5.3.3 muestra un ejemplo de lo recién explicado.

	Aplicación Host	Funcionalidad Volátil
Modelo	CoreModelUpdater	VolatileModelUpdater
Vista	CoreComponentUpdater	VolatileComponentUpdater

Tabla 5.4: Updaters de acuerdo al Target

Otro beneficio de la implementación de esta jerarquía es la reutilización. Los mecanismos de herencia permiten reutilizar comportamiento, en este caso: lógica de actualización genérica. Esta parte de la arquitectura fue implementada creando una estructura de updaters general que sirve para cualquier side game. La jerarquía de updaters combina las dos taxonomías explicadas arriba. Esto permite integrar side games con políticas de actualización genéricas sin la necesidad de especificar concretamente el modo de actualización para cada una de sus partes. Por ejemplo, para el side game doble o nada solo se necesita especificar explícitamente la forma de actualizar el modelo del juego principal y el modelo propio del side game, mediante los updaters DONGameUpdater y DONSideGameUpdater respectivamente. En tal ejemplo la actualización de la vista del juego principal por ejemplo, fue implementada en los updaters genéricos; por lo que no fue necesaria una especificación concreta de lógica de actualización. Cabe destacar que la actualización de la vista del juego principal para los side games implementados como prueba solo consiste en la actualización de los premios del juego principal.

Finalmente, la incorporación de lógica de integración a través de la jerarquización de updaters trae como consecuencia la disminución de la tasa de incorporación de errores. En la sección 5.6 se explica cómo se definieron mecanismos para instanciar todos los elementos necesarios de la capa intermedia para lograr la integración de la funcionalidad. Tal proceso de instalación de la funcionalidad incluye la creación y configuración de los updaters registrados con la funcionalidad volátil. La instalación y configuración automática de updaters como parte de un proceso de instalación más general genera menos trabajo por parte del desarrollador. El desarrollador solo debe implementar la lógica de actualización de sus componentes, dejando de lado el proceso de instalación de dicha lógica en el sistema. Como consecuencia de esto se disminuye la tasa de incorporación de errores en el proceso de integración de nueva funcionalidad. En la sección 5.6.2 se muestra concretamente cuál es el proceso de instalación.

5.3.4. Comunicación entre updaters, modelos y vistas.

A partir de la incorporación de funcionalidad volátil, se deben cumplir reglas de interacción con la aplicación host. Dichas reglas afectan a los modelos de las partes involucradas y tambien a sus vistas. Con respecto a las vistas, se necesita coordinar la actualización de manera tal que la información presentada al usuario sea siempre consistente y refleje el estado de cada modelo. Como se expicará abajo, la comunicación entre la funcionalidad volátil y la aplicación host a través de elementos de la capa de integración hace que en la misma capa de integración se entretejan mecanismos de comunicación. Se establecen redes de comunicación no solamente entre la capa intermedia (updaters) y la aplicación host y la funcionalidad volátil, sino también dentro de la capa intermedia a través de los propios updaters.

En la figura 5.7 se muestran las relaciones de dependencia y asociación entre updaters y los elementos de la aplicación host y funcionalidad volátil:

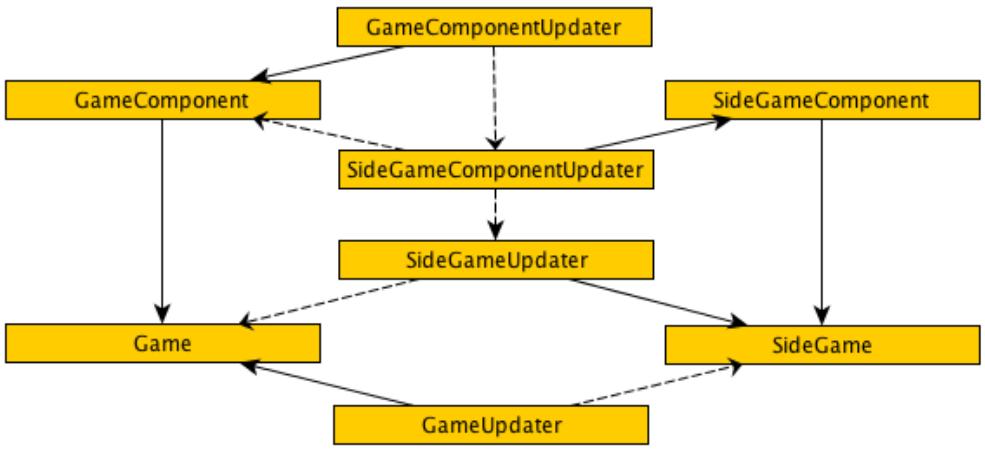

Figura 5.7: Relaciones entre updaters, modelo y vista de core-volatile.

Para explicar las relaciones plasmadas en el gráfico anterior(5.7) es útil un ejemplo concreto que se encontró en ambos side games implementados y que en general se puede encontrar en todo side game de similares características. Cuando un side game es ejecutado, el costo asociado a su ejecución es debitado de los créditos del juego principal y su vista presenta alguna animación que simula su ejecución. Al finalizar su ejecución, si el jugador resulta ganador, los créditos correspondientes son depositados en el juego principal. El ejemplo que se utilizará es justamente la ejecución de un side game cualquiera y sus correspondientes modificaciones como también las necesarias en el juego principal.

Se puede ver en el gráfico 5.7 que todo updater posee una relación de conocimiento con el objeto que actualiza, su target. Además cada updater depende de

uno o más elementos para observar los cambios pertinentes y poder desencadenar las actualizaciones pertinentes. A continuación se explica sucintamente cada updater y su rol en la actualización de elementos para lograr la integración de la funcionalidad volátil:

`GameComponentUpdater` es encargado de actualizar la vista del juego principal, correspondiente a la aplicación host. Como consecuencia de la ejecución de un side game, la vista del juego principal se necesita actualizar -por ejemplo, para mostrar los créditos correctos-. Algunos side games requieren de animaciones en la parte visual para mostrar sus resultados. Una vez que un side game finaliza su ejecución, su parte visual se actualiza impactando luego en la parte visual del juego principal, requiriendo de cierta coordinación para mostrar información adecuada en el momento correcto. Entonces, para poder mostrar los créditos del juego principal actualizados es necesario que se modifique primero su modelo, inmediatamente luego de que el modelo del side game finalice su ejecución. Posteriormente a la actualización del modelo del juego principal, luego de que el side game finalice su parte visual, se debe actualizar su vista.

Para lograr la coordinación mencionada en el parrafo anterior, el `GameComponentUpdater` debe asegurarse que el modelo del side game y la vista del mismo finalizaron su actualización y luego realizar las modificaciones en el juego principal. Es por eso que `GameComponentUpdater` es observer del updater que actualiza la vista del side game (`SideGameComponentUpdater`). El último, luego de actualizar la vista del side game realiza un anuncio para que el `GameComponentUpdater` pueda saber en qué momento terminó la actualización de la vista del side game. El motivo por el cuál `GameComponentUpdater` observa a `SideGameComponentUpdater`, y no al `SideGameComponent`, para saber si la vista del side game se actualizó es que `SideGameComponent` no tiene porqué anunciar que se actualizó. De esta forma se logra menor intrusión en el código de la funcionalidad volátil. Finalmente, la relación de conocimiento con `GameComponent` corresponde al objeto de la vista del juego principal que resulta actualizado.

Los updaters incluyen lógica de actualización de la aplicación host o de la funcionalidad volátil. Es posible que dicha lógica de actualización genere cambios en los elementos actualizados. También es posible que dichos cambios tengan que ser plasmados en otra parte del sistema. Por ejemplo, si un updater de modelo de funcionalidad volátil actualiza el modelo de un side game, es probable que sea necesario actualizar posteriormente la vista del mismo. Un ejemplo concreto en la aplicación desarrollada ocurre con la actualización de side games a partir de jugadas que ocurren en el juego principal.

El juego principal ejecuta sus jugadas en su modelo mientras que en su vista genera animaciones para simular dichas jugadas. Los cambios en el modelo ocurren instantaneamente mientras que la vista muestra información atrasada debido a la necesidad de presentar animacione. Así el modelo del juego y su vista

funcionan a destiempo intencionalmente: el modelo cambia y también lo hace su interfaz pero de forma más lenta, mostrando animaciones y luego sus resultados (Ver figura 5.8):

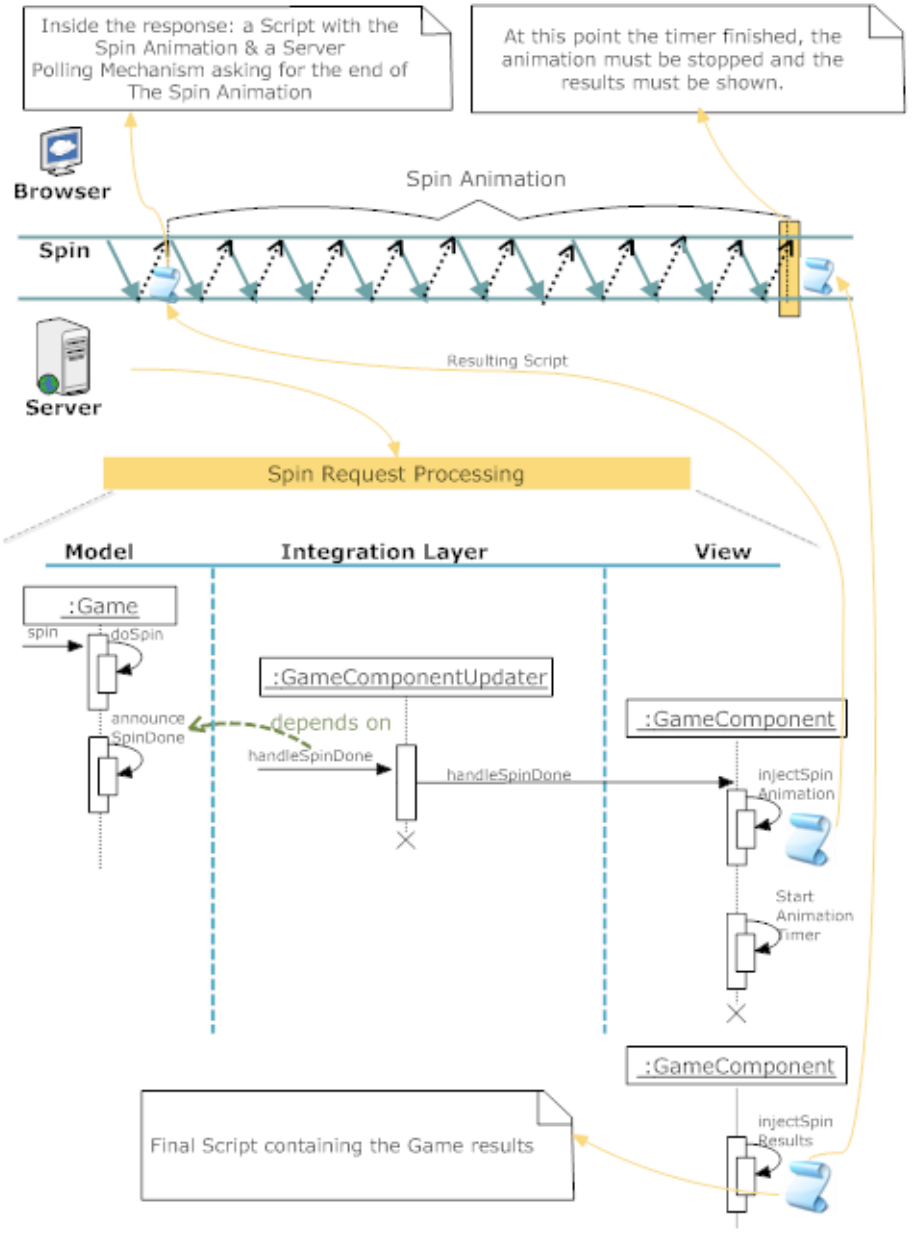

Figura 5.8: Gráfico de modelo y animación spin del juego principal.

Las animaciones en la aplicación desarrollada fueron presentadas en el cliente pero manejadas en el servidor. El cliente mediante de un mecanismo de poll asincrónico consulta al servidor para saber si puede finalizar la animación y pedir el siguiente contenido visual. El servidor maneja los tiempos de la animación

para simular una jugada de máquina que dura cierto tiempo y luego deja saber al cliente que la animación terminó y que puede mostrar los siguientes resultados.

Como consecuencia de la ejecución del juego principal, algunos side games deben ser activados o deactivados dependiendo de circunstancias particulares (como el resultado de la jugada, créditos disponibles, etc.). Cuando el juego principal ejecuta la acción Spin y se otorgan créditos al jugador, el side game doble o nada debe activarse. Cuando no se otorgan créditos el side game debe desactivarse. Debido a que la interfaz y el modelo del juego principal funcionan a destiempo, el side game debe ser actualizado de manera coordinada con dichos cambios. Es por esto que el actualizador de la vista del side game, `SideGameComponentUpdater`, necesita conocer eventos del modelo y de la vista del juego principal para asegurar una coordinación adecuada.

Como consecuencia de los cambios que ocurren en el modelo del juego principal, el updater de modelo del side game `SideGameUpdater` actualiza el estado del juego secundario y anuncia que el side game fue actualizado (es activado o deactivado). Dicho anuncio es recibido por el `SideGameComponentUpdater` para permitir que luego, cuando el evento de vista del juego principal sea recibido, la vista del side game sea actualizada:

```
SideGameComponentUpdater>>listenToModelNotifier: aSideGameUpdater
aSideGameUpdater when: SG.SideGameEnabled
    do: [:announcement | self component mustRenderSideGame:true].
aSideGameUpdater when: SG.SideGameDisabled
    do: [:announcement | self component mustRenderSideGame:false]
```

SideGameComponentUpdater depende de los cambios que ocurren en el modelo del side game para coordinar y desencadenar cambios en su vista y también depende de los cambios que ocurren en la vista del juego principal. Por esto, este updater depende del componente Seaside del juego principal, que avisará cuando el juego fue actualizado visualmente.

```
SideGameComponentUpdater>>listenToViewNotifier: aSlotmachineComponent
aSlotmachineComponent when: Slots.SLSpinButtonPressed ,
                            Slots.SLReSpinButtonPressed
                    do: [:ann | ann script ifNotNil:
            [self component cleanScriptsUsing: ann script]].

aSlotmachineComponent when: Slots.SLSeasideAnnouncement
                    do: [:ann | ann render notNil ifTrue:
            [self component renderScriptsOn: ann render]]
```

Recién se mostró que SideGameUpdater necesita realizar anuncios de eventos. Al incluir funcionalidad de actualización en objetos de la capa intermedia (up-

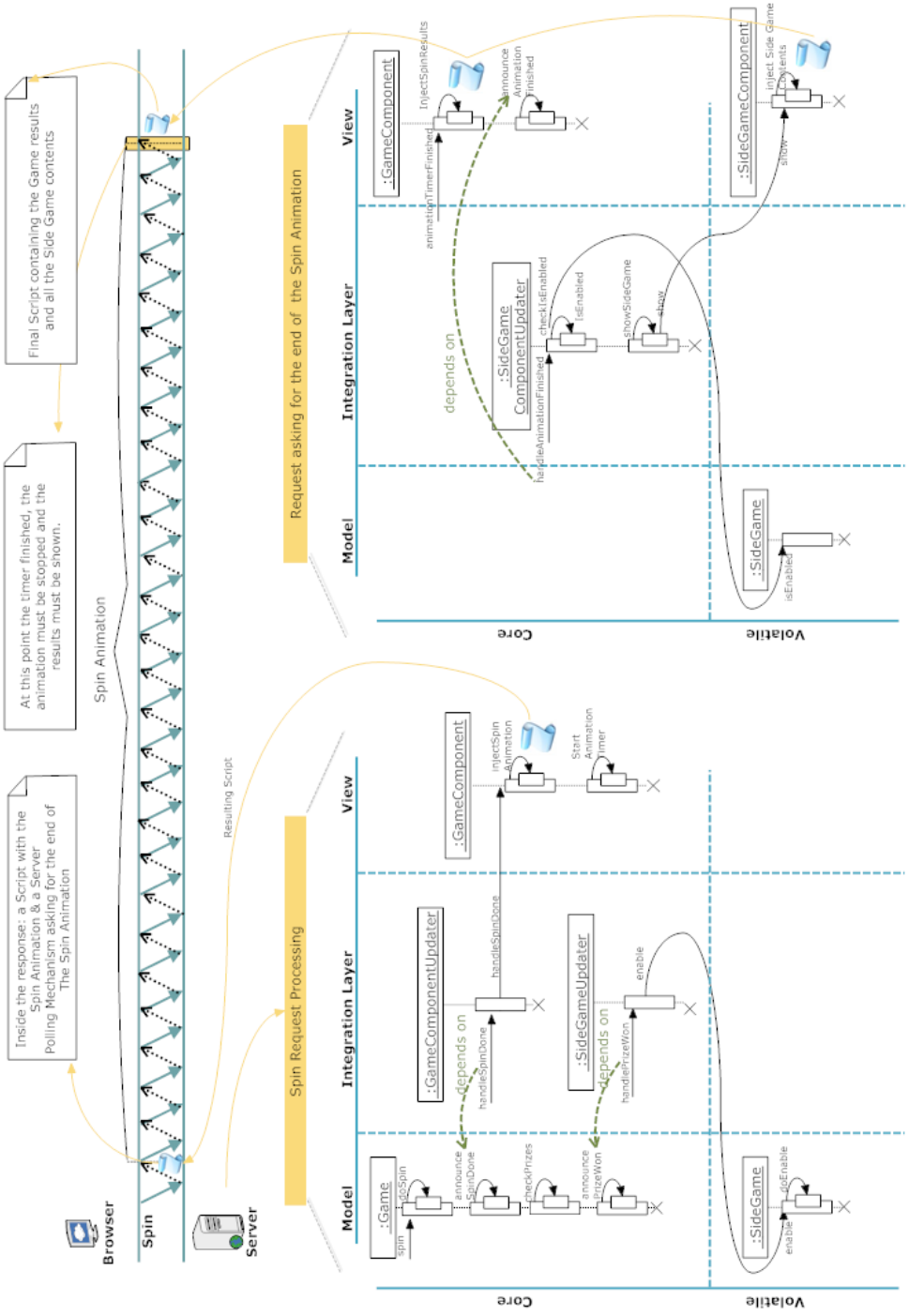

Figura 5.9: Gráfico spin + activación el side game.

daters), fuera de la aplicación host y también fuera de la funcionalidad volátil, puede ser necesario reaccionar a esta funcionalidad. El ejemplo anterior es válido para esto. Para poder reaccionar a funcionalidad implementada en los updaters es necesario, siguiendo los mecanismos propuestos, poder observar eventos provenientes de los mismos updaters. Dichos eventos se deben anunciar dentro del código que representa a la nueva funcionalidad, código que es parte de un updater. Es por eso que un updater debe ser capaz de anunciar eventos. En el caso particular del trabajo desarrollado, fue útil construir la jerarquía de updaters como subclases de Announcer, debido a la capacidad de anunciar eventos.

5.3.5. Uso alternativo de updaters.

Como fue explicado arriba, en este trabajo se utilizan los updaters como elemento de comunicación entre las partes volátiles y core. Esto no quita la posibilidad de usarlos como artefacto para la comunicación interna de alguna de las partes. Se puede crear un updater que cuyo target y notifier pertenezcan a la misma capa, por ejemplo: que ambos componentes sean de funcionalidad volátil. Tal updater puede ser utilizado para implementar comunicación entre elementos internos -notifier y target- minimizando el acoplamiento entre ellos. La idea es la misma explicada en el párrafo anterior, comunicar y desacoplar, pero empleada en una aplicación o funcionalidad concreta, sin tener en cuenta comunicación con componentes externos. Esta es una forma de desarrollar componentes reutilizables a través de aplicaciones mediante el aumento de la modularidad. Por ejemplo, un menu de acciones en una aplicación es un elemento que podría reutilizarse en distintas aplicaciones. Un menú está compuesto por una serie de items que el usuario puede elegir y que disparan la ejecución de acciones. Como consecuencia de la elección de un ítem de menú, y de la ejecución de sus acciones asociadas es posible que el mismo menú u otros elementos, como el frame principal de la aplicación, necesiten ser actualizados. Este caso típico lo podemos encontrar en aplicaciones como Amazon.com, BestBuy.com y otras, donde los menus son construidos dinámicamente de acuerdo a las acciones que llevan al usuario a la página actual.

En esta tesis se utilizó una aproximación similar utilizando Announcements para implementar un menú de acciones configurable dinámicamente. Se utilizó como canal de comunicación que permite comunicar y desacoplar los componentes renderizados en el frame principal de los elementos que aparecen en el menú de acciones del usuario (Ver sección B.2 de implementación de aplicación casino). Así, esta aproximación permite reutilizar el menú de acciones en distintas aplicaciones.

5.3.6. Conclusiones sobre updaters.

Para finalizar la explicación de este mecanismo se explica a continuación, resumidamente cuáles son los beneficios y desventajas de la incorporación de estos mecanismos en la arquitectura:

Organización

A través de la implementación de estos mecanismos se logra una integración de la funcionalidad de manera organizada y en lugares concretos. La funcionalidad que no corresponde naturalmente a los componentes volátiles ni la aplicación core y cuya finalidad es integrar dichos elementos es implementada de forma organizada utilizando estos artefactos. Esto permite programar toda la lógica de integración de la funcionaldiad volátil en un lugar neutro y accesible, de fácil modificación. La lógica de integración de funcionalidad volátil puede ser interpretada y dividida en cuatro partes bien definidas: actualización de modelo core, modelo volátil, vista core y vista volátil.

Mantenimiento y manejo de errores.

Al estar toda la funcionalidad de integración programada en lugares bien definidos es fácil encontrar y analizar cuál es el impacto de la funcionalidad volátil en el sistema host. De la misma manera el proceso de mantenimiento y se hace más acotado ya que es directo encontrar cuáles son las modificaciones posibles a partir de la incorporación de funcionalidad volátil. Así, es más fácil encontrar y reparar errores de comunicación y consecuencias de la incorporación erronea de funcionalidades.

Reutilización

Permite la reutilización de funcionalidad volátil ya que el comportamiento inherente a la integración particular con la aplicación host es implementado en la capa intermedia. Bien se podría haber solucionado para cada side game la integración, las consecuencias de su ejecución y la actualización de las partes desde el código del side game propio. Al incluir este comportamiento en la capa intermedia se fácilita la reutilización de funcionalidad volátil. Cuando se quiere utilizar el mismo side game en otro juego será necesario implementar los updaters según sea corresponda, pero no se deberá cambiar la funcionalidad volátil en sí.

Complejidad

Si bien todo lo anterior es positivo, no se puede dejar de lado el aspecto negativo y es necesario remarcar que la incorporación de estos mecanismos aumenta la complejidad total del sistema. Es necesario ser cuidadoso y tener un buen registro

de los eventos que hay que registrar y de cómo cada updater impacta en una u otra parte a partir de tales eventos.

5.4. Pointcuts

En la sección anterior se explicó el concepto de updater y cómo sus instancias se valen de eventos del sistema host y de la funcionalidad volátil para actualizar los componentes necesarios. En esta sección se detalla otro componente de la arquitectura, que brinda ayuda a los updaters para lograr comunicación entre las partes, los Pointcuts.

5.4.1. Introducción

Luego de programar Updaters de distinto tipo y con distintos cometidos - dedicados a actualizar modelo, vista, aplicación core y funcionalidad volátil- la complejidad y la cantidad de puntos de integración entre la aplicación core y la funcionalidad volátil fue incrementándose. Durante la construcción de los mismos fue surgiendo paulatinamente un nuevo elemento accesorio a los updaters: el Pointcut.

Los pointcuts surgieron al desarrollar lógica de activación y deactivación de side games en los updaters de los modelos correspondientes. Los side games son activados como una reacción a acontecimientos que se dan en el juego principal. Los distintos side games poseen políticas de activación/deactivación diversas, que consisten en puntos específicos en la ejecución del juego principal y determinan el momento en que el side game debe activarse/deactivarse. Para poder programar la lógica de activación/deactivación de los side games fue necesario poder hacer referencia a dichos puntos específicos en la ejecución del programa. Un punto en la ejecución puede determinarse por un evento y una serie de condiciones que deben cumplirse. Este concepto ya se explicó anteriormente en la sección de Updaters. Entonces, para instalar un side game se necesitan una serie de puntos de integración que determinan la forma en que el mismo debe activarse o deactivarse. Para poder referirse concretamente a estos puntos específicos y asociarles funcionalidad de activación o deactivación se incorporó el concepto de Pointcut, tomado prestado del mundo de AOP.

A continuación se explicará cómo surgió este concepto en la arquitectura, sus objetivos, formas de uso y la interpretación del concepto de Pointcut de AOP en este trabajo. Finalmente se incluye un análisis y conclusiones sobre este elemento de la arquitectura.

5.4.2. Definición y componentes de un pointcut

Como se mencionó anteriormente, el concepto Pointcut proviene de AOP. En tal contexto, un pointcut permite describir un punto en la ejecución de un programa y asociar ciertas acciones que se ejecutarán en ese momento [KHH+01]. Concretamente un pointcut asocia un conjunto de condiciones que se evalúan en un momento dado con un conjunto de acciones que se ejecutarán solo si las condiciones se cumplen. Al conjunto de acciones se lo denomina advice y a las condiciones junto con el momento en que se evalúan, se las denominan jointpoints. En esta tesis se utilizó el concepto de Pointcut de AOP para definir un punto particular en la ejecución del programa, en el cuál se realizará alguna tarea de actualización sobre un side game o la aplicación core.

En la implementación desarrollada para esta tesis, un Pointcut se configura con una serie de eventos que pueden disparar su ejecución, un conjunto de condiciones que deben evaluarse y finalmente las acciones que se ejecutarán. El conjunto de eventos determina el momento en el cuál se evalúa el pointcut. Cuando ocurre alguno de los eventos, el pointcut evaluará las condiciones con las cuales fue configurado. El conjunto de condiciones junto con los eventos determinan los *Joinpoints*. Si la evaluación de las condiciones tiene un resultado positivo, se ejecutarán las acciones pertinentes. Las acciones con las cuales el pointcut fue configurado corresponden el *Advice* del mismo. En la figura 5.10 se muestran las partes recién explicadas.

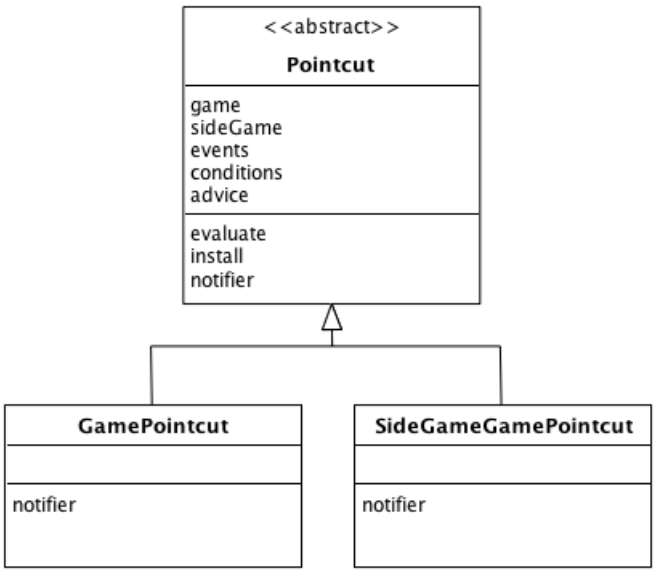

Figura 5.10: Diagrama UML de la jerarquía de la clase Pointcut.

La figura 5.10 muestra dos subclases de Pointcut: SideGamePointcut y Ga-

mePointcut. A continuación se explican los componentes de la jerarquía y las diferencias:

- Variables de instancia game y sideGame: El pointcut es configurado con el modelo de aplicación core (variable de instancia game) y con el modelo de la funcionalidad volátil (variable de instancia sideGame) como variables de instancia. Dependiendo de su tipo, el pointcut observará los cambios en la funcionalidad volátil (sideGame en el caso de SideGamePointcut) o en la aplicación core (game en el caso de GamePointcut). No obstante, ambas variables de instancias, game y sideGame son necesarias para la ejecución del pointcut. Esto se debe a que algunas de las condiciones pueden necesitar consultar cualquiera de estas variables.

- La variable de instancia events corresponde a la colección donde se almacenan los eventos sobre los cuales el pointcut está interesado. Cuando alguno de estos eventos es anunciado y recibido por el pointcut, el mismo disparará su ejecución. La ejecución del pointcut, corresponde a la evaluación de las condiciones y la posterior ejecución del advice.

- El pointcut posee una colección de condiciones que se evaluarán en el momento de ejecución del pointcut. Esta colección es almacenada en la variable de instancia conditions. La evaluación positiva de las condiciones desencadenará en la posterior evaluación del advice. Las condiciones son elementos de la arquitectura que serán detallados en la siguiente sección.

- Finalmente la variable de instancia advice corresponde al código que se ejecutará cuando ocurra alguno de los eventos necesarios y se den las condiciones correctas.

5.4.3. Uso de pointcuts

Los pointcuts son utilizados por los updaters para especificar una política de actualización o de activación. Se definen pointcuts que determinan puntos específicos de la ejecución del programa y son utilizados por un updater para asociar lógica determinada. El pointcut es creado y configurado como parte de la configuración propia del updater que lo utiliza. Las condiciones y eventos son especificadas al momento de crear el pointcut, mientras que el advice se configura utilizando la lógica de actualización del updater en el momento de la instalación del pointcut. A continuación, se incluye un ejemplo de la definición de pointcuts como parte de la especificación de un updater:

```
CWSideGameUpdater>>deactivationPoints
^Array with: (VF.GamePointcut
            game: self game
```

```
                sideGame: self sideGame
                events: Slots.SLSpinDone , Slots.SLReSpinDone
                condition: VF.SideGameEnabledCondition)
        with: (VF.SideGamePointcut
                game: self game
                sideGame: self sideGame
                events: SG.SideGameFinished
                condition: VF.NoCondition)
```

Este ejemplo fue incluido para mostrar cómo los pointcuts permiten especificar con claridad las reglas de activación de la funcionalidad volátil, utilizando el sistema de eventos y condiciones para especificar un punto en la ejecución de la aplicación core. Muestra la definición de los puntos que determinan la deactivación del side game CardWar. Se puede ver que hay dos puntos en los que es necesario deactivar el side game:

1. Cuando el juego principal realiza una jugada (Spin o Respin) y el side game está activado.

2. Cuando el side game finaliza su ejecución (SideGameFinished)

Mediante la utilización de dos pointcuts, el updater define en un método (#deactivationPoints) la lista de puntos en los que el side game debe ser deactivado. El primer Pointcut observa eventos de la aplicación core (VF.GamePointcut) y el segundo de la funcionalidad volátil (VF.SideGamePointcut). El primer pointcut se evaluará cuando alguno de los eventos Slots.SLSpinDone o Slots.SLReSpinDone sea anunciado. Cuando esto ocurra, se evaluará la condición VF.AnyCondition, que siempre se evalúa positivamente y se ejecutará el advice asociado. El advice contiene la lógica propia de la deactivación, y se incluye en el pointcut en el proceso de su instalación.

El proceso de instalación de un pointcut consiste en instalar los observers necesarios para escuchar los eventos correspondientes y asignar el bloque de código que se ejecutará eventualmente (advice)

```
VF.SideGameUpdater>>installDeactivationPointcuts
         self deactivationPoints do:
         [:each | self
                     installPointcut: each
                     withAdvice: [self sideGame disable]]

VF.SideGameUpdater>>installPointcut: pointcut withAdvice: aBlock
         pointcut advice: aBlock;
                  install
```

```
VF.Pointcut>>install
         self notifier
            when: self events
            do: [:announcement | self evaluate ifTrue:
                        [self advice cull: announcement]]
```

El código anterior incluye tres métodos necesarios para comprender el proceso de instalación de un pointcut. El primer método corresponde a la instalación de todos los pointcuts de un updater utilizando el advice correspondiente (en este caso la lógica de deactivación de un side game). La segunda parte (el método #installPointcut:withAdvice:) corresponde a la instalación particular de un pointcut y consiste en asignar el advice y luego invocar al método #install de la instancia del pointcut. Finalmente el método #install consiste en la instalación de observers para los eventos necesarios del pointcut, con la siguiente lógica de ejecución asociada: primero se evalúan las condiciones del pointcut (self evaluate) y luego se ejecuta el advice como consecuencia, utilizando el evento pertinente como parámetro (self advice cull: announcement).

5.4.4. Uso alternativo (implícito) de pointcuts

Hasta aquí se mostró como instanciar un pointcut, configurando sus eventos, condiciones y modelos necesarios para su ejecución. Luego se explicó mediante un ejemplo cómo realizar la instalación de los mismos y se mostró el código implementado para dicho proceso. A continuación se explica otra forma de implementar el concepto de Pointcut que se utilizó en este trabajo.

Además de proveer directamente objetos que representan pointcuts, el concepto también fue utilizado de manera menos explícita, pero manteniéndo presente su semántica. Como alternativa a la utilización de los objetos presentados en esta sección se proveyó, como parte del protocolo de updaters, un método que permite especificar de manera clara un punto de ejecución en conjunto con la funcionalidad a ejecutar. Se mantiene la misma idea de pointcuts pero se implementa sin utilizar los objetos Pointcuts, de manera implícita. Los pointcuts son utilizados por updaters para especificar lógica de actualización en un punto dado de la ejecución del programa. Se construyó un método que permite a los updaters realizar tal tarea: #when:triggers:underCondition:do:. Así, es posible determinar pointcuts sin utilizar instancias de la jerarquía presentada anteriormente. Se explica esta idea a través del siguiente ejemplo:

```
DONGameUpdater>>installPointcuts
self
    when: self sideGame
    triggers: SG.SideGameFinished
```

```
underCondition: VF.AnyCondition
do:[self game emptyPrizes.
    self sideGame hasWon
        ifTrue: [self game addCredits: self sideGame finalCredits]]
```

Este ejemplo muestra el manejo implícito del concepto pointcut a través del uso del método recién mencionado. El GameUpdater del side game doble o nada especifica la lógica de actualización del juego, a través de la invocación a #when:triggers:underCondition:do:. El significado del código anterior es el siguiente: cuando el side game anuncia el evento SG.SideGameFinished, que indica que ha finalizado su ejecución, se analiza la condición VF.AnyCondition, que es evaluada positivamente y luego se ejecuta el código especificado en el bloque final. Se pueden ver claramente los componentes del pointcut: El evento que dispara la ejecución del pointcut (SG.SideGameFinished), la condición que se evalúa (VF.AnyCondition) y el advice que se ejecuta:

```
[self game emptyPrizes.
 self sideGame hasWon ifTrue:[self game
                                  addCredits: self sideGame finalCredits]].
```

El método #when:triggers:underCondition:do: es parte del protocolo de la clase Updater y se define como sigue:

```
when: aNotifier
triggers: events
underCondition: anIntegrationCondition
do: aBlock

    aNotifier when: events asAnnouncementClasses
              do: [:ann | (anIntegrationCondition evaluateOnGame: self game
                                                  sideGame: self sideGame)
                            ifTrue: [aBlock cull: ann]]
```

La utilización de este método permite a los updaters especificar pointcuts implícitamente. Aquí se muestra un ejemplo equivalente a la utilización de pointcuts mediante el método mencionado. Se puede ver como se definen en un mismo método los pointcuts de manera implícita, pero a la vez clara y concisa:

```
CWGameUpdater>>installPointcuts
self
    when: self sideGame
    triggers: SG.SideGamePlayDone , CWWarDone
```

```
        underCondition: VF.NoCondition
        do: [:ann | self game subtractCredits: ann sideGame wager].
    self
        when: self sideGame
        triggers: SG.SideGamePlayDone
        underCondition: VF.SideGameWon
        do: [:ann | self game addCredits: ann sideGame wager * 2].
    self
        when: self sideGame
        triggers: CWWarDone
        underCondition: VF.SideGameWon
        do: [:ann | self game addCredits: ann sideGame wager * 3].
    self
        when: self sideGame
        triggers: CWPlayerSurrender
        underCondition: VF.NoCondition
        do: [:ann | self game addCredits: ann sideGame wager / 2]
```

El ejemplo anterior muestra la utilización de Pointcuts de manera implícita a través del método *#when:triggers:underCondition:do:*

5.4.5. Análisis de ventajas y desventajas

Los Pointcuts son una forma alternativa de implementar código de actualización de updaters. En la sección anterior se mostró como un updater especifica lógica de actualización de, por ejemplo, la vista de un side game:

```
SideGameComponentUpdater>>listenToViewNotifier: aSlotmachineComponent
aSlotmachineComponent when: Slots.SLSpinButtonPressed ,
                            Slots.SLReSpinButtonPressed
                      do: [:ann | ann script ifNotNil:
            [self component cleanScriptsUsing: ann script]].

aSlotmachineComponent when: Slots.SLSeasideAnnouncement
                      do: [:ann | ann render notNil ifTrue:
            [self component renderScriptsOn: ann render]]
```

Esta lógica de actualización podría haberse programado utilizando un Pointcut.

```
Pointcut p = SideGamePointcut
               game: self game
               sideGame: self sideGame
```

```
                events: Slots.SLSpinButtonPressed, Slots.SLReSpinButtonPressed
                condition: VF.AnyCondition.
   p advice:  [:ann | ann script ifNotNil:
                                [self component cleanScriptsUsing: ann script]].
   p install.
```

Esto significa que los pointcuts también pueden programarse de manera standard, sin utilizar la clase Pointcut ni el método #when:triggers:underCondition:do:, como se mostró en la sección de updaters 5.3.4. Por ejemplo:

El siguiente código corresponde a la lógica de deactivación de un side game y utiliza Pointcuts:

```
VF.SideGameUpdater>>installDeactivationLogic
            self installPointcuts: self deactivationPoints
                 withAdvice: [self disableSideGame]

DONSideGameUpdater>>deactivationPoints
      ^Array with: (VF.GamePointcut
            game: self game
            sideGame: self sideGame
            events: Slots.SLSpinDone , Slots.SLReSpinDone
        condition:(VF.NothingWonCondition and: VF.SideGameEnabledCondition))
       with: (VF.SideGamePointcut
            game: self game
            sideGame: self sideGame
            events: SG.SideGameFinished
            condition: VF.NoCondition)
       with: (VF.GamePointcut
            game: self game
            sideGame: self sideGame
            events: Slots.SLCashOutWasDone
            condition: VF.NoCondition).
```

De no contar con este mecanismo, los puntos en los que un side game debe ser activado o deactivado deben ser programados como parte del código de actualización de un pointcut. Una posible implementación alternativa, sin utilizar Pointcuts, utilizando mecanismos standard provistos por Smalltalk, hubiese sido:

```
    DONSideGameUpdater>>installDeactivationLogic
       self game when: Slots.SLSpinDone , Slots.SLReSpinDone
                    do: [|result condition|
       condition := (VF.NothingWonCondition and: VF.SideGameEnabledCondition).
```

```
result = condition evaluateOnGame: self game sideGame: self sideGame.
result ifTrue:[self deactivateSideGame
              ]].

self sideGame when: SG.SideGameFinished
          do: [self deactivateSideGame].
self game when: Slots.SLCashOutWasDone
          do: [self deactivateSideGame].
```

De esta manera, queda claro que se pueden especificar puntos de integración sin utilizar los mecanismos explicados en esta sección. Como consecuencia, surge una pregunta obvia: ¿Por qué proveer dos mecanismos similares para realizar la misma acción? En primer lugar, no fue la intención inicial proveer un nuevo mecanismo, ya que surgió paulatinamente y apareció como una forma de implementar lógica de actualización. Segundo, para la lógica de actualización correspondiente a la activación y deactivación de side games, este mecanismo brindó mucha facilidad y claridad para especificar claramente cuáles eran las situaciones concretas en las que se debía activar o deactivar un side game. Los Pointcuts y las Condiciones constituyen mecanismos que permiten al programador separar concerns pragmáticamente en lugar de atarse a programar comportamiento en el lugar del código donde debería ocurrir [FF00]. Todo el código necesario para saber cuáles son los momentos y condiciones en los cuales un side game debe ser activado están condensados en el mismo lugar. Esto brinda gran claridad para comprender una parte muy importante de la integración de funcionalidad volátil: el patrón de activación o Volatility Pattern. Lo mismo ocurre con la lógica para deactivar un side game.

Los costos de implementar este componente deben ser tenidos en cuenta. Utilizando Smalltalk y a través de BlockClosures y Annoucements el costo de implementación fue mínimo. La complejidad introducida en el framework de integración de side games fue relativamente baja ya que solo se necesitó implementar la jerarquía de pointcuts y los mecanismos de integración con los updaters (solo 3 métodos).

5.4.6. Conclusiones

Se presentó el concepto de Pointcut y se explicó el uso del mismo en este trabajo. Se desarrollo el concepto como parte de la arquitectura y se proveyeron dos formas para su utilización: explícita e implícita. También se mencionó la posibilidad de no utilizar estos elementos. Los Pointcuts brindan claridad y poseen costos de implementación bajos. No obstante, se puede lograr lo mismo utilizando otros mecanismos estándares. La conclusión final es que este componente de la arquitectura resultó útil y brindó cierto nivel de claridad, pero bien podría

quitarse de la arquitectura. Si bien introduce claridad en el proceso de especificar puntos de integración (como la implemetación del volatility pattern -ver sección 2.2.2-) y se aumenta la capacidad de expresión, los costos de su incorporación y utilización en la capa intermedia pueden ser superiores a los beneficios asociados. Por lo que la incorporación en otra posible implementación de una arquitectura resulta opcional.

5.5. Condiciones

5.5.1. Introducción

A medida en que se utilizaron eventos para describir puntos de integración y las acciones que la funcionalidad volátil o la aplicación host deben efectuar, fue apareciendo otro componente: las condiciónes. Cuando un evento es anunciado, es necesario corroborar que ciertas condiciones se cumplan para determinar si las acciones correspondientes deben ser ejecutadas. Por ejemplo, cuando el juego principal efectúa una jugada, es necesario saber si la misma generó premios; para poder así ofrecer el side game doble o nada. En Smalltalk sería directo evaluar tales condiciones utilizando un bloque de código lo suficientemente complejo que permita acceder a todas las partes necesarias y realizar las consultas pertinentes. No obstante, es útil poder referirse a las condiciones necesarias de manera concreta y simple, para poder desarrollar la integración con mayor claridad. Este concepto es central para la apreciación de los objetos condición y será explicado en los párrafos que siguen.

Los pointcuts utilizan eventos para determinar el momento en que sus acciones asociadas serán ejecutadas. En algunos casos la ejecución de tales acciones depende de ciertas condiciones que se deben evaluar sobre el modelo de la aplicación core o de la funcionalidad volátil. Como se mencionó en el parrafo anterior, las condiciones pueden ser evaluadas accediendo al modelo de forma directa utilizando bloques de código. Estos bloques pueden ser codificados junto con las acciones a ejecutar para determinar la condicion de ejecución. El problema de tal aproximación es que, al incorporar el código que determina las condiciones de ejecución, el código de la lógica de actualización se oscurece. Considerando el concepto de pointcut y en terminos de Separation of Concerns [W.D74, HL95], el código de actualización y el código de evaluación de condiciones corresponden a dos Concerns diferentes y sería un problema juntarlos (tangled code, al menos). Con el fin de especificar las acciones de un pointcut de forma clara, cohesiva y sin oscurecer el código de las mismas, se separó la acción a ejecutar de las condiciones que determinan su ejecución.

Independientemente de la separación entre condiciones y acciones, propuesta en el párrafo anterior, las condiciones pueden ser programadas utilizando bloques de código lo suficientemente complejos que provean acceso a todos los elementos

necesarios. Un bloque de código Smalltalk consiste en una secuencia de código diferido (listo para ejecutarse eventualmente), que puede recibir parámetros [GR83]. Tales bloques de código solo necesitarían acceder a los elementos a evaluar y contar con la lógica de evaluación que utilice dichos elementos. Esta forma de evaluar las condiciones es simple y directa ya que la lógica se programa al configurar un pointcut utilizando un BlockClosure (bloque de código Smallltalk) pero no obstante, oscurece la especificación de pointcuts. Como la forma de entender la condición que determina el bloque requiere la lectura y comprensión del mismo, al momento de la especificación del Pointcut esto agrega complejidad. Es por esto que la solución implementada en esta tesis es un tanto diferente.

La aproximación utilizada en esta tesis permite la especificación de las condiciones de evaluación de forma más clara y directa para lograr aumentar la legibilidad del código de integración -código de especificación de pointcuts-. Para lograr esto, se dejó de lado la alternativa de codificación directa mediante bloques. Se optó por encapsular la lógica de evaluación de condiciones en objetos representativos de tales condiciones. Estos objetos, denominados justamente IntegrationCondition, permiten identificar lógica compleja de evaluación de condiciones mediante un nombre autoexplicativo. Por ejemplo, PrizeWonCondition.

En este apartado se explicarán las decisiones que se tomaron en esta aproximación y las ventajas y beneficios de la utilización de la misma. Se aclarará cómo se definen y utilizan las condiciones y también otros detalles pertinentes como la jerarquía propuesta para implementar condiciones y la posible composición de las mismas.

5.5.2. Definición y funcionamiento

El elemento de la arquitectura que representa a los objetos condición, y sirve para especificar pointcuts, se denominó IntegrationCondition. Consiste en un objeto que encapsula el acceso y la lógica necesaria para evaluar el estado de un componente de la aplicación core o de la funcionalidad volátil y determinar un resultado. Dicha lógica es especificada mediante bloques de código Smalltalk que serán evaluados, eventualmente, utilizando los componentes necesarios como parámetro. La ejecución de una IntegrationCondition consiste en la parametrización de la misma con los componentes a evaluar mediante la invocación de un método pertinente.

En el contexto de un pointcut, la condición sirve para referirse a un estado particular del modelo- de forma directa y claramente identificable- y determinar si el pointcut debe ejecutar o no sus acciones asociadas. El siguiente código muestra: primero, la utilización de la condición en el contexto de un pointcut y segundo, la evaluación de la condición:

```
1. Creación de un pointcut (utilización de condiciones):
```

```
VF.GamePointcut game: self game
                sideGame: self sideGame
                events: Slots.SLSpinDone , Slots.SLReSpinDone
                condition: VF.PrizeWonCondition
```

La especificación del pointcut anterior determina que el mismo evaluará sus acciones (definidas en otro lugar) cuando al ocurrir alguno de los eventos SLSpinDone o SLReSpinDone, se cumpla la condición PrizeWonCondition. En algún momento, cuando el pointcut reciba el anuncio de los eventos recién mencionado, se evaluará el pointcut y como consecuencia se ejecutará la condición para determinar si las acciones asociadas al pointcut deben ejecutarse:

```
2. Evaluación de una condición:

VF.Pointcut >> evaluate
     ^self condition evaluateOnGame: self game sideGame: self sideGame
```

El código anterior corresponde a la evaluación de un pointcut, explicada en la sección anterior (Ver sección de Pointcuts5.4), y consiste en la evaluación de la condición que se incluye en el Pointcut. Como se explicará a continuación, la petición de ejecución de la condición tiene en cuenta todos los parámetros necesarios.

En el trabajo desarrollado con los prototipos, las condiciones se especifican sobre elementos del modelo de la funcionalidad volátil o sobre el modelo de la aplciación core (Receivers). Debido a esto, la condición es invocada pasando como parámetro dichos elementos. Concretamente, estos elementos corresponden al objeto que representa el modelo del juego principal y al que representa el modelo del side game. Más allá del trabajo realizado para esta tesis, la implementación de Condiciones no debe estar limitada solamente a elementos del modelo. Por supuesto, estas condiciones pueden ser extendidas para ejecutarse sobre otros elementos, como por ejemplo sobre la interfaz de las funcionalidades volátiles. Las condiciones pueden ser diseñadas para cubrir todas las areas: a saber, MVC de la aplicación core y de la funcionalidad volátil; y permitir la definición de pointcuts más complejos sobre cualquier elemento de la aplicación o de la funcionalidad volátil.

Considerando lo explicado en el párrafo anterior, la forma de invocar la ejecución de una condición de integración, es enviando el mensaje #evaluateOnGame:sideGame:. El método que se ejecuta cuando el mensaje es enviado, recibe el modelo de la aplicación host (game) y el modelo de la funcionalidad volatil (side game) como parámetro. Este mensaje se corresponde con el mensaje #execute de lo que sería una implementación del patrón Command.

5.5.3. Condiciones como Command

Las condiciones pueden ser vistas como instancias del patrón de diseño Command[GHJV94], con ciertas modificaciones. Para poder explicar con mayor claridad cómo se relaciona este patrón con el concepto de IntegrationCondition y su contexto, se explicarán los roles del patrón en el contexto de IntegrationCondition. Esta explicación servirá para poder definir el comportamiento de estos objetos en la arquitectura propuesta. La figura 5.11 muestra la estructura y los participantes del patrón Command, justo con sus roles.

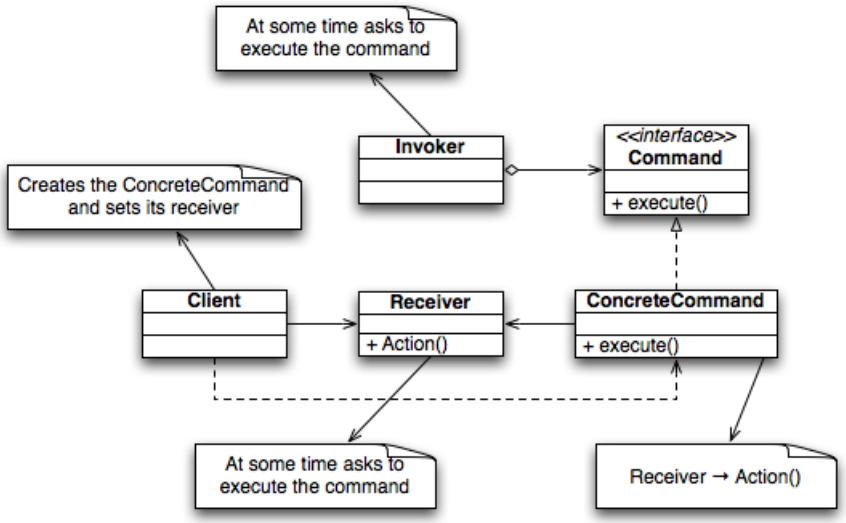

Figura 5.11: Diagrama de la estructura del patrón Command

Los comentarios en el diagrama anterior explican los roles de cada participante en el patrón. Los objetos IntegrationCondition se corresponden con la clase abstracta Command. Cada condición programada es una subclase de IntegrationCondition y en el diagrama se corresponde con la clase ConcreteCommand. Con respecto al Receiver, las condiciones pueden poseer más de un objeto receptor sobre el cuál se evaluará la lógica de la condición, aunque este no es el caso típico de su uso. Algunas condiciones como RandomCondition y NoCondition no tienen Receiver y la mayoría, como PrizeWonCondition, solo tiene un receiver -un componente de la aplicación host en el caso de PrizeWonCondition-. Cuando las condiciones se utilizan dentro de un Pointcut, para determinar la ejecución de sus acciones, dicho Pointcut funciona como Invoker de la condición consultando por su resultado y disparando su ejecución. En general, las condiciones, al igual que los pointcuts pueden ser creadas por updaters que necesitan de especificar puntos específicos en la ejecución del programa donde se iniciará la actualización. Dichos updaters cumplen el rol de Client dentro de la interpretación de condiciones como

commands.

5.5.4. Jerarquía de condiciones y condiciones compuestas

La jerarquía de condiciones permite la composición de condiciones complejas utilizando conectores lógicos. Esta aproximación implementa el patrón de diseño Composite[GHJV94] y su objetivo es poder especificar condiciones de forma más natural en la especificación de Pointcuts. Por ejemplo:

Definición de un poincut con condición compuesta:

```
VF.GamePointcut game: self game
     sideGame: self sideGame
     events: Slots.SLSpinDone , Slots.SLReSpinDone
     condition:(VF.NothingWonCondition and: VF.SideGameEnabledCondition)
```

Este ejemplo muestra cómo se componen dos condiciones utilizando un operador lógico (and). El objetivo de esta composición es poder especificar pointcuts más clara e intuitivamente. En este caso, el pointcut ejecutará sus condiciones cuando el juego principal no otorgue premios y cuando el side game esté habilitado. Cabe remarcar que de no contar con estos objetos, que permiten especificar estas condiciones, rápida y naturalmente, se debería programar la lógica de evaluación del estado tanto del juego principal (para saber si entregó premios) y del side game (para saber si está habilidado). Esto se analizará más adelante.

Una alternativa a la composición de condiciones, es la parametrización de pointcuts con una colección de condiciones. Esta alternativa fue implementada como primera solución y luego fue reemplazada por la composición recién explicada. Por ejemplo:

Primer alternativa: definición de poincut con una colección de condiciones:

```
VF.GamePointcut game: self game
          sideGame: self sideGame
          events: Slots.SLSpinDone , Slots.SLReSpinDone
          conditions: (OrderedCollection
                              with:VF.NothingWonCondition
                              with:VF.SideGameEnabledCondition)
```

Idem anterior, utilizando una simplificación gramatical para la definición de la colección de condiciones:

```
VF.GamePointcut game: self game
          sideGame: self sideGame
          events: Slots.SLSpinDone , Slots.SLReSpinDone
          conditions:(VF.NothingWonCondition, VF.SideGameEnabledCondition)
```

Se mostraron dos alternativas que funcionan de la misma forma, mediante colecciones. La primera utiliza el protocolo de Colleciton para generar la instancia de OrderedCollecion. En el segundo ejemplo, el mensaje (#,) es enviado a la primer condición para generar una colección que incluye ambas condiciones. Ambas alternativas son equivalentes. El problema de esta forma de implementar pointcuts con múltiples condiciones es que no queda clara cuál es la forma de evaluar la colección de condiciones. De alguna manera hay que evaluar todas las condiciones y procesar sus resultados para producir un resultado único. Para esto, se pueden utilizar al menos los operadores lógicos or y and. Esto trae ambiüedades de interpretación y además, aunque es trivial, esta lógica de procesamiento de resultados debe ser programada como parte de de los pointcuts. Es así que, para aumentar el poder de expresión, eliminar ambigüedades y simplificar la forma de definir pointucts: se optó por la alternativa de condiciones compuestas utilizando el patrón Composite, en lugar de colecciones.

La implementación del patrón composite se desarrolló como parte de la jerarquía de IntegrationCondition. La lógica de composición es implementada en la superclase abstracta IntegrationCondition. El proceso de construcción de condiciones compuestas utiliza una clase denominada CompositeCondition (correspondiente a la clase Composite del patrón) que es capaz de integrar dos condiciones bajo un operador lógico. La ejecución de una condición compuesta consiste en la ejecución de las condiciones que la componen y la unión de los resultados utilizando el operador lógico. Debido al tipo de resultados que manejan las condiciones, booleanos, la composición resultó trivial utilizando simples operadores lógicos.

Como resultado del desarrollo de los prototipos mencionados, surgió una jerarquía de condiciones reutilizables en side games similares. Ejemplos de estas condiciones son: RandomCondition, NoCondition, PrizeWonCondition. La posible reutilización de condiciones agrega valor a la arquitectura implementada ya que fácilita la futura implementación de side games mediante un proceso de integración donde ciertos aspectos ya se encuentran resueltos. El diagrama UML de la figura 5.12 muestra la superclase abstracta IntegrationCondition y la clase CompositeCondition, responsable de agrupar condiciones utilizando operadores lógicos. También muestra algunas de las otras subclases implementadas como parte de la jerarquía de condiciones reutilizables del framework de integración de side games desarrollado.

La condición AnyCondition tiene el siguiente significado: Cuando es evaluada siempre da resultado positivo, lo que es equivalente a no evaluar condición alguna (de ahí su nombre). Se utiliza para especificar pointcuts que deben ejecutar su condición cuando ocurre un evento determinado, sin importar condición alguna. Ejemplo:

```
when: self sideGame
    triggers: SG.SideGamePlayDone , CWWarDone
```

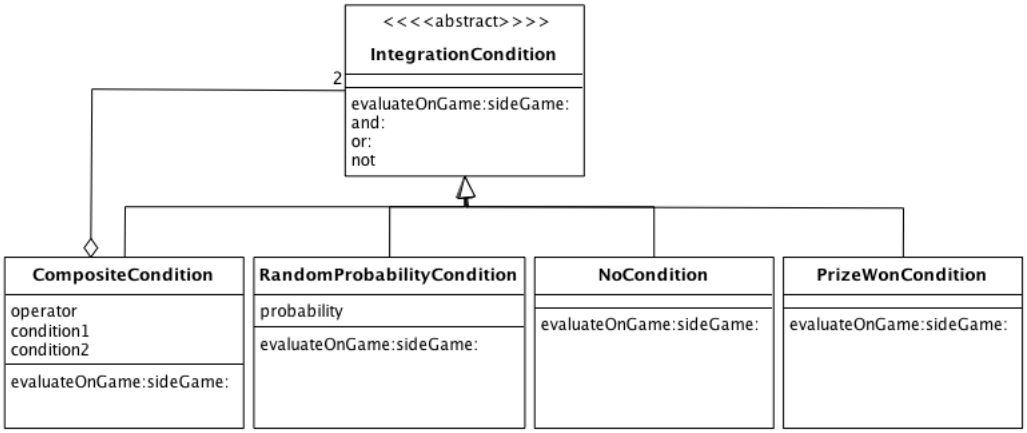

Figura 5.12: Diagrama UML de la jerarquía de Condiciones de Integración

```
underCondition: VF.AnyCondition
do: [:ann | self game subtractCredits: ann sideGame wager]
```

Esta porción de código pertenece al updater de modelo de la aplicación host del side game CardWar (CWGameUpdater). Especifica que cuando ocurre una jugada en el side game (SG.SideGamePlayDone o CWWarDone) se deben substraer créditos del juego principal sin importar la condición.

PrizeWonCondition fue explicada con anterioridad. Se evalúa sobre un juego y su resultado es positivo si el mismo otorgó premios en el último play.

5.5.5. Análisis de ventajas y desventajas.

Estos artefactos elevan el nivel de abstracción y permiten integrar más fácilmente las funcionalidades volátiles. Esto se debe a que se disminuye la complejidad asociada a incluir código de evaluación que quizás sea complejo, reemplazándolo por un objeto significativo y representante de la condición misma. Un ejemplo claro de esto es el siguiente:

```
CWSideGameUpdater >> activationPoints
^Array with: (VF.GamePointcut
            game: self game
            sideGame: self sideGame
            events: Slots.SLSpinDone , Slots.SLReSpinDone
            condition: (VF.RandomCondition probability: 0.3))
```

La condición RandomCondition es configurada con una probabilidad p (0 ¡= p ¡= 1). La idea es que cada vez que se ejecute, su resultado solo dependa de la probabilidad dada. Así, cuando se ejecuta la condición, existe una probabilidad

p de que el resultado sea verdadero. Una forma de programar una condición de tal naturaleza es utilizando números aleatorios, generados por Smalltalk. La idea es generar un número aleatorio entre 0 y 1 y verificar si tal número es menor o igual a la probabilidad especificada. De ser así, el resultado de la condición será verdadero, en el caso contrario será falso. El siguiente código corresponde al método de evaluación de Condiciones implementado:

```
VF.ProbabilityBasedCondition >> evaluateOnGame: aGame sideGame: aSideGame
FastRandom new next <= self probability
```

En caso de no contar con los objetos condición, la programación del pointcut, incluyendo el código de evaluación del evento aleatorio, sería como sigue:

```
CWSideGameUpdater >> activationPoints
^Array with: (VF.GamePointcut
            game: self game
            sideGame: self sideGame
            events: Slots.SLSpinDone , Slots.SLReSpinDone
            condition: [:game :sideGame | FastRandom new next <= 0.3])
```

Este ejemplo muestra cómo la condición encapsula lógica que sería dificil de entender. Este problema se transladaría al entendimiento del pointcut y oscurecería el proceso de integración.

Este mismo problema fue presentado con un ejemplo en el apartado anterior, donde se remarcó la importancia de contar con estos objetos; particularmente para la especificación de condiciones compuestas. En el caso ejemplificado, la concición del pointcut era la conjunción de las condiciones NothingWonCondition y SideGameEnabledCondition y se podía especificar simplemente insertando el operador and: entre ambas condiciones:

```
VF.GamePointcut game: self game
     sideGame: self sideGame
     events: Slots.SLSpinDone , Slots.SLReSpinDone
     condition:(VF.NothingWonCondition and: VF.SideGameEnabledCondition)
```

De no contar con los objetos IntegrationCondition la programación del pointut ejemplificado sería como sigue:

```
VF.GamePointcut game: self game
     sideGame: self sideGame
     events: Slots.SLSpinDone , Slots.SLReSpinDone
     condition:[:game :sideGame | game hasWinningPrizes not &
                                  (side game isEnabled) ]
```

En este caso, el código de las condiciones es fácil de entender debido a que tanto el juego principal como el side game poseen un protocolo que permite acceder fácilmente a los resultados buscados y por lo tanto, el bloque utilizado no es muy complejo. No obstante, la composición de condiciones sugiere un aumento de complejidad en el código de evaluación. Este problema es atacado por la jerarquía implementada, permitiendo la especificación clara y natural mediante la composición y la utilización de subclases de IntegrationCondition con nombres autoexplicativos.

Otra ventaja es la potencial reutilización de condiciones para distintos side games. Se resaltó anteriormente que la reutilización de condiciones agrega valor a la arquitectura implementada ya que aumenta el conjunto de herramientas que permite al desarrollador llevar a cabo una integración menos costosa. Por supuesto que las condiciones deben ser desarrolladas y que algunas de las condiciones no podrán ser reutilizadas. No obstante, en el trabajo realizado, con la construcción de solamente dos prototipos, surgieron condiciones reutilizables que se organizaron como parte de un posible framework para el desarrollo de side games.

Un aspecto negativo de incorporar este mecanismo para especificar condiciones es que es necesario generar una nueva subclase de IntegrationCondition, cada vez que se necesite especificar una condición. Esto es en el caso en que no se pueda generar una condición a partir de la composición de otras condiciones previamente definidas. Por supuesto, para poder contar con este último recurso es necesario haber programado los mecanismos de soporte necesarios para la implementación de la composición.

El segundo aspecto negativo es que la incorporación de este mecanismo aumenta la complejidad y cantidad de código extra que hay que programar para soportar y ofrecer mecanismos de integración de funcionalidad volátil. De no proveer estos objetos IntegrationCondition, el desarrollador del sistema, que genera el soporte para la integración de funcionalidades volátiles, no deberá programar las estructuras y mecanismos necesarios para manejar condiciones de evaluación como se explica en esta aproximación. En esta tesis, el pequeño framework de integración de side games se diseño y programó incoporando estos mecanismos de condiciones. Conociendo la mecánica de los patrones Command y Composite, esto realmente no resultó un desafío mayor y su desarrollo fue más bien trivial.

5.5.6. Conclusiones

Los objetos IntegrationCondition fácilitaron la integración de la funcionalidad volátil a través de la especificación más simple y clara de Pointcuts. Es importante destacar la importancia del aumento de la capacidad de expresión que brindan estos mecanismos, al igual que una mayor legibilidad. Leer código que utiliza estos objetos (con nombres autoexplicativos y que engloban lógica compleja de

evaluación de estado) resulta mucho más natural y brinda mayor fluidez. Además, tal tipo de código es más fácil de comprender que aquél que incluye directamente la lógica de evaluación ya que el código resultante es más compacto y claro.

Concluyendo con este análisis, si se tienen en cuenta los beneficios del uso de estos elementos: mayor legibilidad, mayor poder de expresión y capacidad de reutilización; contra sus aspectos negativos: mayor complejidad en el framework de integración de funcionalidad volátil, necesidad de subclasificar; se puede decir que el balance general resultó positivo a favor de la utilización de estos objetos.

5.6. Instalación de funcionalidad volátil.

Las secciones anteriores detallaron los componentes de la capa intermedia, necesarios para implementar comunicación entre la funcionalidad volátil y la aplicación core. En esta sección se explica todo lo referente a la instalación y configuracion de los elementos de la capa intermedia. También se explica el proceso inverso: remover la funcionalidad volátil.

5.6.1. Introducción

El proceso de instalación de funcionalidad volátil es complejo y resulta de vital importancia para poder integrar la funcionalidad con la aplicación core. En el mismo se efectúan las uniones necesarias entre la aplicación core, la capa intermedia y la funcionalidad volátil para establecer los mecanismos de comunicación y de actualización entre las partes. Finalmente, el proceso de instalación debe tener su contraparte, para poder contar con una aplicación core funcional en el caso de remover la funcionalidad volátil.

La instalación de funcionalidad volátil consiste en la instanciación de todos los componentes necesarios. Por una parte es necesario instanciar los componentes volátiles: modelo, controllers y vista de la funcionalidad volátil; y por otra parte se necesita instanciar los componentes de la capa intermedia: updaters, condiciones y demás. La instanciación debe ser seguida por la configuración necesaria para que los componentes mencionados puedan interactuar entre si y con la aplicación core. En ocasiones, el proceso de instalación necesita que ciertas modificaciones se incorporen a la aplicación core, para que la misma se comporte como es esperado por las funcionalidades volátiles. Por ejemplo, ciertos eventos deben ser anunciados por la aplicación core. Para lograr esto sin modificar el código existen ciertas técnicas que serán explicadas en el apéndice A.

Primero se explicará el proceso de instalación y los artefactos utilizados en el mismo. Luego se explicará como se remueve la funcionalidad volátil de la aplicación core. Finalmente, como en las demás secciones, se incluye un análisis de lo explicado y las conclusiones.

5.6.2. Proceso de instalación

Para explicar el proceso de instalación se comenzará por el contexto donde el mismo fue implementado, para poder luego extrapolar los resutlados a un proceso más general. En el trabajo desarrollado, la aplicación core consiste en un sistema casino que maneja juegos online. El usuario puede elegir entre distintos juegos del casino para realizar sus apuestas. Solo un juego fue implementado como parte del casino, el juego generala previamente introducido. La incorporación de juegos secundarios sugiere la posibilidad de que el usuario elija conjuntos de juego principal-side games para utilizar. Así, si el sitio provee un juego x, al incorporar los side games a y b, el usuario potencialmente podría elegir entre los "paquetes"x (juego solo), x-a (juego con side game a), x-b(juego con side game b), x-a-b (juego con ambos side games). Entonces el proceso de instalar side games debe tener como resultado la posibilidad de elegir el juego o paquete de juego-sidegame(s) a utilizar.

El sistema core está compuesto por los juegos principales (uno solo en este caso) y la aplicación casino que permite ofrecer tales juegos. El proceso de instalar la funcionalidad volátil en este contexto está conformado por una serie de pasos:

1. Primero es necesario determinar cuales serán los conjuntos de juego principal y side games que serán ofrecidos. Este proceso podría realizarse en forma automática combinando los juegos detectados en el sistema core, junto con los side games detectados en la imagen Smalltalk. No obstante esto se dejó como decisión para un usuario administrador, encargado de generar tales combinaciones. Para realizar tal tarea se construyó una aplicación que permite elegir juegos core y configurarlos con side games a elección, generando nuevos paquetes que son puestos a disposición de los jugadores. En el apéndice A se explica con detalles esta aplicación (Packager) y las decisiones de diseño asociadas a su construcción.

2. Luego de definir las posibles combinaciones, es necesario generar las agregaciones de juego y side games correspondientes. Este paso consiste en generar las instancias necesarias de los modelos de la funcionalidad volátil y agregarlas a instancias de componentes de la aplicación core (vista como un todo, el sistema casino con los juegos implementados). Los componentes a los cuales se agregará la funcionalidad volátil en este caso son las instancias del juego principal que serán ofrecidas. Con las instancias de funcionalidad volátil y juego principal se genera un paquete que los reune (Ver apéndice A, sección de aplicación Packager).

3. El siguiente paso consiste en generar los componentes de la vista necesarios para introducir la funcionalidad volátil como parte del contenido presentado por el sitio. Concretamente esto consiste en generar componentes Seaside

que contengan el juego principal en conjunto con los side games. Esto consiste finalmente en la generación de componentes seaside que agreguen los componentes seaside volátiles con los componentes seaside del juego principal.

4. Luego es necesario brindar acceso a los componentes de la vista generados en el paso anterior. Esto consiste en brindar al usuario la posibilidad de elegir los paquetes de juego-side games generados anteriormente.

5. Finalmente es necesario instanciar todos los elementos de la capa intermedia necesarios para que la comunicación entre la funcionalidad volátil y la aplicación core se lleve a cabo sin problemas. Este paso consiste en la instanciación de updaters, pointcuts y condiciones, configurando todos los observers necesarios.

Los pasos recién explicados son específicos para la instalación de los side games en los prototipos desarrollados. No obstante, como se mencionó arriba, estos pasos no deberían diferir en la utilización de la arquitectura en otros contextos. De esta manera, se definen a continuación, de forma general, los pasos a seguir para instalar funcionalidades volátiles utilizando la aproximación explicada:

1. **Definición de los puntos donde se incorporará la funcionalidad volátil**. En el caso de la implementación explicada, la funcionalidad volátil (side games) se integra junto con un componente de la aplicación core casino (juego principal generala). Estos puntos deben ser definidos en la etapa de diseño de la funcionaldiad volátil ver [RNM+06, RGDU08] para mayor información.

2. **Instanciación de los elementos de los modelos**. Necesarios a incluir como parte de la integración. En el caso de las aplicaciones desarrolladas, estos elementos son agregados en una especie de paquete contenedor, pero esto quizás en otros contextos no se aplique.

3. **Instanciación de los componentes de la vista**. Este paso consiste en generar los componentes de la vista necesarios para mostrar el contenido de la funcionalidad volátil. Esto se da solo en casos donde la funcionalidad volátil posee vista por supuesto.

4. **Introducción de puntos de acceso a los componentes volátiles**. La funcionalidad volátil debe ser accedida de alguna forma, quizás se incluya directamente dentro de los componetes de la aplicación core, quizás se ofrezcan links que disparen su uso. Sin importar la forma de integrar la funcionalidad, si la misma cuenta con una vista, de alguna forma se deberá integrar con la vista de la aplicación core.

5. **Instanciación de elementos integradores**. La capa de integración contiene los elementos que sirven para lograr mantener comunicados y actualizados los componentes volátiles con la aplicación core. Es de vital importancia este paso para contar con una integración funcional de los módulos volátiles.

Las posibles combinaciones de juego y side games debieron crearse programáticamente. Para solucionar esto se implementó una herramienta que permite al administrador del sitio o super-usuario, generar paquetes a elección, configurando el juego principal con los side games disponibles. De esta forma se generan paquetes en el momento de instalación de la funcionalidad volátil para que el usuario final de la aplicación pueda elegirlos para jugar. La figura 5.13 muestra algunas capturas de pantalla de la aplicación de instaladora, denominada "Packager". Las capturan muestran cómo se puede configurar una instancia de juego principal con distintos side games y cómo se puede instalar o desinstalar la funcionalidad volátil en su totalidad.

Los detalles del modelo implementado para esta aplicación se pueden encontrar en el apéndice B.5 de implementación.

El proceso explicado arriba en primer lugar es muy particular de los prototipos implementados. No obstante, al utilizar la arquitectura para otro tipo de aplicaciones, el proceso explicado sigue siendo válido y la aproximación utilizada no debería alejarse demasiado de la serie de pasos explicada en segundo lugar. Siempre será necesario interceptar el código que genera el contenido que habrá que mostrar, para incluir entonces la funcionalidad volátil. Las construcciones necesarias para lograr esto (por ejemplo, los paquetes de juegos) como también las técnicas de modificación del código (por ejemplo, los method wrappers) pueden variar en las distintas tecnologías e implementaciones. Tecnicas de AOP pueden utilizarse para alterar el comportamiento del proceso de generación del contenido. La forma de agregar la funcionalidad volátil junto con la aplicación core es bien específica de los dominios en cuestión.

5.6.3. Proceso de desinstalación

El proceso de desinstalación consiste en varios pasos que serán explicados a continuación:

1. Es necesario remover los mecanismos de modificación de comportamiento de la aplicacion core. En el caso de los prototipos desarrollados, solo es necesario desinstalar los method wrappers necesarios. Es de vital importancia realizar este proceso de manera adecuada ya que los efectos secundarios de dejar method wrappers fantasmas en la aplicación pueden ser muy difíciles de encontrar. Los problemas asociados a esto se pueden encontrar en la sección de problemas de method wrappers en el apéndice A.6.

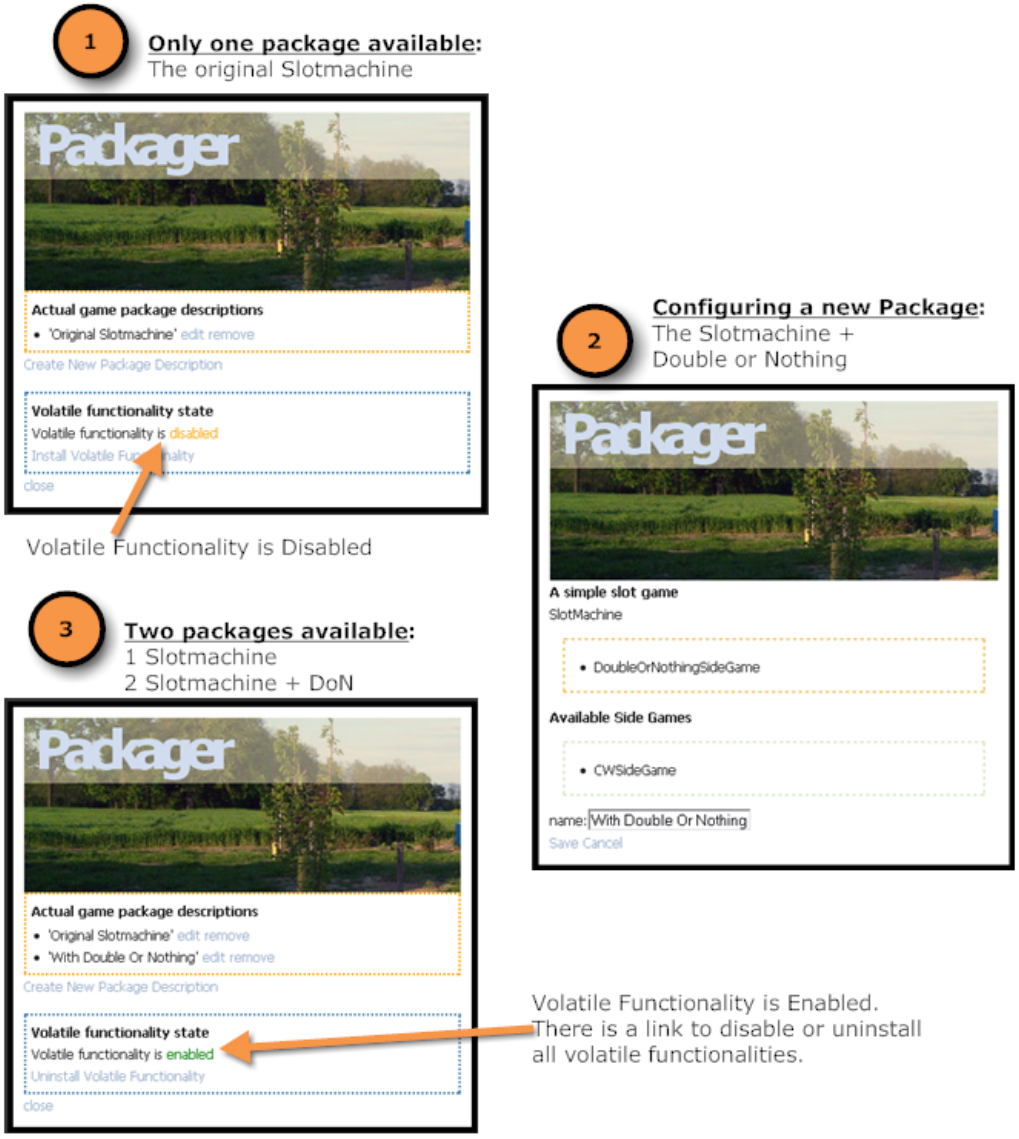

Figura 5.13: Screenshots de la aplicación instaladora

2. También es necesario eliminar las dependencias instaladas mediante observers entre los elementos de la capa intermedia, aplicación core y funcionalidades volátiles.

3. Por último se deben eliminar las instanacias innecesarias correspondientes al modelo y vista de las funcionalidades volátiles, como también los elementos de la capa intermedia generados.

Este proceso se logró automatizar a través de la aplicación Packager (ver la sección B.5), que permite instalar y desinstalar side games dinámicamente sobre los juegos que se encuentran disponibles en el sistema.

5.6.4. Conclusiones

El proceso de instalación de la funcionalidad volátil es de vital importancia para la correcta integración de la misma y no es algo sencillo. El patrón de conexión de la funcionalidad puede ayudar a diseñar el proceso de instalación, pero la implementación dependerá del contexto donde se desarrolle. No obstante las potenciales diferencias, el proceso de instalación puede seguir unos pasos granulares definidos genéricamente. Finalmente, la correcta implementación del proceso de instalación puede devenir en la automatización del mismo, como ocurrió a través de la aplicación Packager.

Capítulo 6

Trabajo Relacionado

6.1. Introducción

En esta sección se analizará la bibliografía leida en relación a este trabajo de tesis. Algunos de los trabajos se relacionan directamente con el foco del desarrollo y otros tienen una naturaleza diferente pero su evaluación resulta provechosa. Otros trabajos relacionados no fueron estudiados por cuestiones de tiempo y debido al alcance de esta tesis de grado. Alguno de estos trabajos son: [BKKZ05, CVWH07, GG01, HOT06, KV08, LB05, MAW06, NZF04]

6.2. Svahnberg et. al. A Taxonomy of Variability Realization Techniques

En el paper *A Taxonomy of Variability Realization Techniques* se estudia el problema de la extensión de software, desde el punto de vista de las etapas de un proceso de desarrollo controlado e industrializado, con respecto a familias de productos. Los autores analizan el desarrollo de líneas de productos de software y cómo las diferencias entre productos de la misma línea pueden ser manejadas utilizando el concepto de variabilidad. Argumentan que la forma adecuada de desarrollar líneas de productos o familias de productos, es mediante la explotación de las similitudes y la implementación de las diferencias como *variabilidad* entre los productos.

Segun los autores, *Software Variability* es la habilidad de un sistema de software o artefacto de ser eficientemente extendido, cambiado, personalizado o configurado para su utilización en un contexto particular. Los productos dentro de una línea deben ser diseñados de manera tal que las configuraciones particulares puedan ser facilmente incorporadas mediante componentes dedicados, a través

de una estructura de soporte o arquitectura pensada con tales objetivos. El esfuerzo del trabajo está enfocado en discutir los factores necesarios que deben considerarse para seleccionar métodos apropiados o técnicas para implementar variabilidad. Finalmente el paper se enfoca en aspectos de implemencatión de técnicas de variabilidad que son presentadas en extensión.

El trabajo realizado en esta tesis consiste en el diseño, implementación y estudio de una de las posibles estrategias o técnicas para enfrentar el problema analizado por los autores: la extensibilidad a través del concepto *Software Variability*. Poder copar con funcionalidad volátil de manera eficiente para lograr dicha extensibilidad, es un caso particular de software variability. Esto es así debido a que la funcionalidad volátil puede ser considerada como un componente variable, además temporal, del software o de una línea de productos de software.

Una arquitectura para el soporte de la introducción de funcionalidad volátil puede ser considerada como una técnica de realización de software variability. También se explica, en el paper en cuestión, que una fuente de problemas mayor en el uso de técnicas que permitan implementar variabilidad, es la falta de conocimiento sobre las mismas (overview, pros y contras de cada técnica). Esta tesis trata de aumentar el conjunto de técnicas sobre variabilidad disponibles dando una descripción y un análisis de la técnica propuesta (incorporación de funcionalidad volátil mediante el uso de una arquitectura).

A continuación se presentan brevemente similitudes y diferencias con el trabajo estudiado:

- Al igual que el estudio en cuestión, en esta tesis se considera que la incorporación de nueva funcionalidad, *feature* en el caso del paper, tiene un impacto en otras partes del sistema. Ciertamente, se mostró en esta tesis cómo la incorporación de funcionalidad volátil como un nuevo feature puede afectar desde el comportamiento de los objetos del modelo de la aplicación hasta la forma en que la vista de la misma es presentada al usuario (ver sección 2.2.2).

- El concepto de componente que manejan los autores equivale al objetivo ideal que la incorporación de la arquitectura presentada propone sobre la funcionalidad volátil. Se espera de la funcionalidad volátil pueda ser reutilizada, al estilo plugin en distintos sistemas hosts. Los autores del paper definen el concepto de componente como una unidad de composición que puede ser desplegada independientemente pero que está sujeta a la composición con otras terceras partes. Bajo esta interpretación, un side game o funcionalidad volátil (componente) tiene cierta independencia ya que puede ser incorporado en distintos juegos, pero está sujeto a dicha incorporación ya que sólo no puede funcionar, o siquiera existir.

- Los autores definen como Variation Points a los lugares del diseño e implementación -o código- que permiten a un feature ser variable. Son los puntos del programa que permitiran introducir nueva funcionalidad o features. En esta tesis, estos puntos son definidos como puntos de integración y son caracterizados en parte por los atributos Intent y Extent de la funcionalidad volátil. Los variation points, o puntos de integración, son implementados utilizando Pointcuts, Updaters y Condiciones en esta tesis.

Finalmente y al igual que en esta tesis, el propósito del trabajo estudiado no es proveer flexibilidad ilimitada, sino proveer la flexibilidad necesaria para afrontar las necesidades presentes y futuras de un sistema, minimizando el impacto de la incorporación de cambios y disminuyendo los costos de mantenimiento.

6.3. Aspect-Oriented Programming is Quantification and Obliviousness

Durante el desarrollo de los prototipos de side games de la arquitectura presentada, particularmente a partir de la implementación del componente Pointcut (ver sección 5.4); fueron surgiendo incógnitas acerca de la relación entre la funcionalidad volátil y conceptos de AOP. ¿Constituye la funcionalidad volátil un concern, programable mediante técnicas AOP? ¿Es posible realizar las aserciones necesarias para poder finalmente programar funcionalidad volátil con técnicas AOP? El trabajo de Filman et. al. *AOP is Quantification and Obliviousness* [FF00] analiza los sistemas AOP a partir de ciertas propiedades para poder definir una heurística de decisión sobre si un sistema es AOP o no. En dicho trabajo los autores presentan un estudio de diversas propiedades de sistemas AOP que sirve para analizar la arquitectura desarrollada desde la perspectiva de AOP. A continuación se presentarán algunos puntos importantes del trabajo de Filman et. al. y las relaciones encontradas con el trabajo desarrollado en esta tesis.

6.3.1. EBPS

En un sistema EBPS (Event Based Publish Subscribe)[FF00] los concerns pueden ser realizados por medio de suscripciones a los eventos de interés de tal concern. Se explicó anteriormente cómo los updaters (ver sección 5.3) utilizan un sistema de eventos(ver sección 5.2) para determinar el momento en que se ejecutarán las acciones pertinentes a la integración de funcionalidad volátil. Los mismo ocurre con los pointcuts (ver sección 5.4) que sirven para especificar puntos exactos en la ejecución del programa de manera declarativa. El comportamiento de la funcionalidad volátil (el concern) y de los mecanismos de integración es implementado a partir de la suscripción a los eventos de interés que ocurren en la aplicación host. Entonces, queda claro que la arquitectura planteada, a través

de la utilización de eventos para definir reglas de integración y comportamientos específicos de la funcionalidad volátil, puede ser catalogada como un sistema EBPS.

Indican los autores que en esta clase de sistemas se pierden las propiedades de Unidad y Localidad del programa. Esto es estudiado en el apéndice de Method Wrappers (ver apéndice A). Al ser un sistema EBPS, la localidad se pierde debido a que el comportamiento disparado a partir de la aparición de eventos se dispersa en los updaters y la funcionalidad volátil.

Según los autores, "Si el estilo de programación de la aplicación es utilizar eventos como la interfaz entre componentes, entonces EBPS es un mecanismo Black-box AOP. Contrario a esto, si se espera que el programador disperse la generación de eventos para los propósitos perseguidos, a través del programa, no es obvlivious y por lo tanto no es AOP "[FF00]. Los side games se podrían acoplar a juegos que levanten una serie de eventos. En este caso nos encontramos con un sistema de las características primeras, Black box. En el caso de querer integrar un side game con un juego que no anuncie los eventos requeridos, será necesario acudir a una solución de las mencionadas en el apéndice A.

6.3.2. Concepción de AOP

Luego de presentar un análisis de algunos atributos de sistemas AOP, los autores presentan su concepción del concepto de AOP:

> *"AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantfications hold over programs written by oblivious programmers."*

En relación a este párrafo y cómo el mísmo puede interpretarse de acuerdo a la arquitectura presentada, se explican dos partes importantes: la frase *"To make quantified statements about the behavior of programs "* se refiere a poder especificar un punto exacto en la ejecución del programa. A partir de esos puntos se podrá incluir nueva funcionalidad (programar el concern, en el caso de AOP). En el caso de la arquitecura presentada, este proceso se realiza sobre la aplicación host a través de eventos y condiciones que permiten al desarrollador especificar concretamente momentos en el comportamiento del programa. Con respecto a "and to have these quantifications hold over programs written by oblivious programmers "se puede remarcar que es deseable que los side games puedan ser utilizados en distintas aplicaciones host. Utilizando esta interpretación se puede caracterizar la arquitectura presentada como un sistema AOP en el que las funcionalidades volátiles son consideradas concerns sobre la aplicación host. Con respecto a esto, los autores proponen como objetivo de poder decir:

"This code realizes this concern. Execute it whenever these circumstances hold "

o

"In programs P, whenever condition C arises, perform action A " [FF00]

De esta manera, como uno de los objetivos de la arquitectura presentada, es deseable poder realizar aserciones del estilo: *"Cuando en el juego base, o aplicación core, se cumple la condición C, ejecutar la acción A."* Por ejemplo *"Cuando la slotmachine hace spin y el juego entregó premios, activar el side game doble o nada."*

6.3.3. Sobre Dynamic Quantification

La forma de integrar funcionalidad volátil provista por la arquitectura propuesta es mediante lo que los autores definen como Dynamic Quantification. Dynamic Quantification consiste en atar el comportamiento del aspecto a algo que ocurre en tiempo de ejecución [FF00]. En la arquitectura implementada, a través del sistema Publish-Subscribe, se asocia el comportamiento de un side game a una condición del contexto donde se encuentra el side game y el juego principal. Para que los side games se suscriban a los eventos del juego, y la dynamic quantification recién mencionada se encuentre en ejecución, es necesario aumentar el código de ciertas partes del juego para que se instancien los objetos necesarios. Este aumento fue logrado mediante Method Wrappers, una técnica clear box[FF00], explicada en el apéndice A, que permite aumentar el código del programa.

Para concluir el análisis de la relación con el trabajo de Filman et. al. se introduce un texto del paper en cuestion. En el mismo los autores justifican que los sistemas AOP mejor construidos son los que permiten al programador desconocer los detalles del sistema host (oblivousness) para integrar concerns: "Just program like you always do, and we'll be able to add the aspects later. "[FF00]. Sería ideal poder decir: *Programa el juego de la forma en que originalmente lo haces. Nosotros podremos añadir los side games luego.* El objetivo de la arquitectura presentada queda expresado en esa frase.

6.4. Rules engine.

Un rule engine, según [Fow09], es una serie de condiciones con una acción asociada. Las reglas pueden ser escritas en cualquier orden y el sistema las evalúa en cualquier orden. El sistema elige las reglas donde se cumple la condición especificada y evalúa las acciones correspondientes. Usualmente la característica

principal de un rule engine es que los usuarios finales, "business people", pueden especificar las reglas ellos mismos, construyendolas sin la ayuda del programador. Un rule engine, como es explicado posee justamente un motor que permite evaluar las condiciones y ejecutar las acciones. No obstante esto, en el análisis que se presenta a continuación, me refiriré a rule engine a menudo, como el concepto de conjunto de reglas especificadas en un mismo lugar.

El sistema de eventos presentado como parte de la arquitectura no fue pensado como un rule engine, pero sin dudas tiene muchas similitudes. El sistema funciona utilizando eventos y evaluando acciones cuando los eventos ocurren. La ocurrencia de eventos implica el momento en el que el sistema evaluará las acciones de las reglas. Los eventos ocurren y de acuerdo a cómo fue programada la regla, se evaluarán condiciones para finalmente ejecutar las acciones.

El sistema de eventos se parece aún más a un rule engine cuando se utilizan Pointcuts, Conditions y advices. Estos objetos son una forma de especificar eventos, condiciones, y acciones en el juego. Las condiciones se evaluarán cuando ocurra algún evento especificado en el pointcut. Cuando las condiciones son satisfactorias se evalúan las acciones, especificadas en los Advices. Con este esquema, claramente el sistema utilizado se puede interpretar como un rule engine. La mayor diferencia es que el experto en dominio no escribe las reglas, el programador lo hace. Otra diferencia reside en el lugar donde se especifican las reglas. Estas reglas se especifican en lugares dispersos a través de los distintos updaters de los side games. Un rule engine quizás necesita que las reglas se escriban en un lugar común.

El sistema de eventos también soporta un uso de mecanismos standard (Announcements), dejando de lado las reificaciones (Pointcut, Condition, Advice). En esta forma de uso, el programador escribe las reglas especificando las condiciones y las acciones como un bloque de código que se evaluará como consecuencia de un evento. Esto puede ser interpretado como un rule engine donde las reglas se escriben de una forma determinada usando no un DSL sino Smalltalk. Si bien este sistema puede ser utilizado de distintas formas, por lo que es versatil, resultó conveniente aglomerar las reglas escritas en un lugar común. Esto facilitó el mantenimiento de las mismas, ya que de otra forma se encuentran dispersas y son difíciles de corregir. Como consecuencia directa de este aglomeramiento, el conjunto de eventos y acciones se parece mucho a un rule engine.

Queda claro entonces, que no resulta forzado pensar en el sistema de eventos implementado como un rules engine. El papel de conjunto de reglas aglomeradas en un sitio es interpretado por: Pointcuts + Conditions + Advices o bien, por un conjunto de Eventos + Bloques de condiciones y Acciones. Mientras que la parte *engine*, de rules engine, es ejecutada por Smalltalk y el framework de eventos Announcements.

6.5. Active Rules for Runtime Adaptivity Management

En este paper [DMM+07], Daniel et. al. estudian el tema de características adaptativas (adaptive features) de aplicaciones web. Tales características consisten en elementos de software que pueden que tienen la capacidad de ser adaptados en un sistema. A través de una aproximación basada en un sistema de reglas ECA (Event-Condition-Action), proponen manejar la evolución de elementos adaptativos de una aplicación. Debido a su naturaleza temporal y a que puede ser incorporada y removida a menudo, la funcionalidad volátil puede ser considerada como una característica adaptativa de un sistema. Es así, que el trabajo desarrollado en el paper en cuestión será analizado considerando a la funcionalidad volátil como un *adaptive feature*. Resultan notorias las similitudes entre ambos trabajos. Esto se debe a que ambos utilizan un sistema ECA para especificar de una u otra manera las reglas de integración (de adaptive features o de funcionalidad volátil).

Los autores soportan la idea de un sistema de reglas ECA a través de la creación de un lenguaje basado en XML , llamado "ECA-Web ", que sirve para especificar las reglas de integración de adaptive features. Mediante tal lenguaje permiten la especificación de comportamiento adaptativo. Además de un lenguaje los autores presentan un arquitectura que funciona como soporte para las reglas generadas con el lenguaje. En la misma, introducen un engine desacoplado que les permite capturar eventos para activar las acciones correspondientes de los "Adaptive features". Los autores argumentan que al poder especificar las reglas de adaptatividad de forma desacoplada y ejecutarlas autónomamente mediante el engine desarrollado, se provee una forma de diseñar y administrar los adaptive features de forma independiente (ortogonal al diseño del sistema host). El diseño por separado de las funcionalidades volátiles y la aplicación host fue uno de los ejes primordiales de la arquitectura presentada en esta tesis(ver sección 4.2). De acuerdo a lo recién explicado, es claro que los objetivos de la arquitectura propuesta y del lenguaje+arquitectura presentados en el paper en cuestión, no están muy alejados.

6.5.1. Comparación de los sistemas ECA utilizados

Considerando a la funcionalidad volátil como un adaptive feature, en esta tesis, se soporta la adaptatividad mediante una arquitectura basada en un sistema ECA. En esta tesis no se proveyó un lenguaje para la definición de reglas. En la aproximación implementada, las reglas de inegración son especificadas por el desarrollador utilizando mecanismos provistos por la arquitectura como Updters, Pointcuts y demás. El lenguaje ECA-Web permite definir reglas de interacción mediante la especificación de, entre otras cosas, Eventos, Condiciones y Acciones. A continuación se explicarán sucintamente algunas similitudes y diferencias entre el sistema ECA propuesto en el paper en cuestión y el utilizado en la arquitectura

desarrollada en esta tesis:

Eventos (Igual).

Mediante los eventos, se especifica cómo una regla es disparada en respuesta a acciones del usuario o cambios en el modelo. Lo mismo ocurre en la arquitectura propuesta, los eventos constituyen la forma de evaluar Pointcuts o ejecutar acciones de Updaters, a partir de cambios del modelo o acciones del usuario.

Condiciones (Similar).

Los autores presentan el concepto de condición, que permite evaluar el estado de los datos de la aplicación para decidir si la acción correspondiente a una regla debe ser ejecutada o no. Lo mismo ocurre con los componentes Condición de la arquitectura propuesta. Los mismos permiten consultar no solo el estado de la aplicación host sino también de la funcionalidad volátil. De esta forma se puede decidir si se ejecutarán o no las acciones de actualización.

Acciones.

Mediante el concepto de Acción, el lenguaje ECA-Web permite especificar el comportamiento de adaptación de la aplicación en respuesta a un evento disparado y a una condición que evalua a "true". Este concepto equivale al concepto de Advice de un pointcut o al bloque de código de actualización de un Updater en la arquitectura provista.

6.5.2. Comparación de las arquitecturas

Además del lenguaje mencionado en los párrafos anteriores, los autores presentan una arquitectura que utiliza tal lenguaje para generar reglas de integración y poder procesarlas. La arquitectura está pensada para manejar las reglas generadas con el lenguaje. Debido a que ambas arquitecturas están basadas en un sistema ECA se notan similitudes (algunas ya expuestas). A continuación se explican algunas similitudes encontradas en ambas arquitecturas:

Event Managers.

Como parte de la arquitectura expuesta, los autores presentan el concepto de Event Manager. Un Event manager permite capturar eventos y activar la ejecución de las reglas para finalmente ejecutar las acciones de actualización. EventManager es la superclase de una taxonomía que divide a los EventManagers de acuerdo a la naturaleza de los eventos, el tipo de evento (Data Event, Web Event, External Event, etc.). En la arquitectura propuesta en esta tesis no se presenta el concepto de EventManager explicitamente. En la misma, los eventos

son capturados por mecanismos de observers instalados como parte de Updaters o Pointcuts.

Action Enactors.

De acuerdo a la aproximación presentada en el paper hasta aquí analizado, las acciones corresponden a modificaciones a la aplicación web host de los adaptive features. Las adaptaciones (acciones) pueden ser alteración de contenidos de página, reestructuración de la estructura de hypertexto, modificación de la logica del modelo y demás.

En la arquitectura presentada en esta tesis, el concepto de Advice de un Pointcut corresponde a las acciones presentadas en el trabajo aquí analizado. Lo mismo ocurre con la lógica de actualización programada en updaters mediante bloques de código. Anteriormente se explicó que las acciones de actualización pueden ser programadas sin soporte extra proveniente de la arquitectura (Ver sección 5.4.5). Lo mismo remarcan los autores del paper, sin embargo también proveen un artefacto que permite encapsular las acciones a ejecutar: ActionEnactor.

La ejecución de aciones de adaptatividad es realizada a través de la utilización de ActionEnactors. ActionEnactor es la superclase de una taxonomía generada a partir del lugar de la aplicación donde se ejecutarán las acciones (Web, External, Data). En la arquitectura presentada en esta tesis, Los lobjetos encargados de realizar las acciones de actualización (equivalentes a las acciones de adaptatividad bajo el enfoque seguido hasta ahora) son los Updaters. Los Updaters fueron clasificados en dos taxonomías como se explicó en la sección 5.3.3 de acuerdo al lugar de impacto en el modelo MVC o de acuerdo al impacto en la funcionalidad volátil o la aplicación host. Como los adaptivity features pertenecen a la aplicación host, son elementos de la misma, en el trabajo evaluado esta última clasificación no resulta necesaria. La clasificación expuesta en tal trabajo se compara con la propuesta en la arquitectura, siendo WebActionEnactor equivalente a ComponentUpdater y ExternalActionEnactor/DataActionEnactor equivalentes a ModelUpdater.

Updaters = ActionEnactor + EventManager.

Los EventManagers son capaces de recibir eventos para disparar la ejecución de acciones, al igual que los Updaters. Los Updaters son los encargados de la lógica de actualización, al igual que los ActionEnactors. Entonces, a partir de lo expuesto aquí (proveniente de los párrafos anteriores) se puede observar que el concepto de Updater, como es concebido en esta tesis, corresponde a una conjunción entre el concepto EventManager y ActionEnactor.

RuleEvaluator.

El concepto de pointcut de esta tesis se corresponde con el concepto de Rule Evaluator. Este objeto está encargado de evaluar comndiciones para determinar si las acciones se deben ejecutar o no dependiendo del estado de la aplicaicón.

Proceso de instalación = Deploying ECA Rules.

Los autores también proponen un proceso de instalación en el cuál se configuran los Event Mangers y los Action Enactors a partir de los eventos disponibles en la aplicación. Este proceso consiste en utilizar las reglas definidas en el lenguaje ECA-Web para generar todos los elementos necesarios para la integración de los adaptive features. Claramente se corresponde con el proceso de instalación de funcionaldiad volátil explicado en la sección 5.6.2. En el mismo se generan los elementos necesarios de la capa de integración para que la funcionalidad volátil funcione de manera adecuada.

6.5.3. Conclusiones

Finalmente los autores presentan un pequeño análisis de cómo realizar modificaciones en la interfaz gráfica de una RIA. La forma de introducir cambios en los contenidos visuales, generados dinámicamente por la aplicación, no dista de las técnicas explicadas en este trabajo. Al utilizar técnicas perteneciente a la familia de tecnologías Ajax, será necesario de alguna manera alterar el documento html producido.

Se puede concluir, a partir de este acotado análisis del trabajo estudiado, que el mismo y la tesis desarrollada comparten muchas similitudes. Si bien el problema concreto de cada trabajo tiene un nombre distinto, adaptive features por un lado y volatile functionality por otro, ambos tratan de atacar un problema mayor de los sistemas de software web actuales: la evolución. Este problema, como se mencionó ya varias veces, no es un problema nuevo y por lo tanto no resulta extraño que dos caminos de estudios separados desemboquen en soluciones similares. Ambos trabajos proponen, a partir de un estudio centrado en aspectos de implementación, una arquitectura para atacar los aspectos cambiantes de un sistema. La arquitectura provista por Daniel et. al. es más general. Está diseñada de manera tal que el lenguaje ECA-Web sea soportado y las reglas producidas en el mismo puedan ser instanciadas y manejadas en el sistema, para lograr la integración de los adaptive features. La arquitectura provista en esta tesis está más orientada a un tipo de feature particular, la funcionalidad volátil, y se trata de considerar al mismo como un potencial módulo reutilizable, de caracter temporal. La arquitectura última se centra en la comunicación entre un sistema host y un módulo ajeno, a integrar con el mismo, mientras que la porpuesta por

Daniel et. al. considera a los adaptive features como parte del sistema a modificar, por lo que no repara en cuestiones de reutilización y comunicación modular.

6.6. Evolution of Web Applications with Aspect-Oriented Design Patterns

Los autores del paper *Evolution of Web Applications with Aspect-Oriented Design Patterns* [BVD07], argumentan que la utilización de AOP permite manejar cambios en un sistema con el objetivo de controlar su evolución. En dicho trabajo se centra el foco en el estudio de patrones y técnicas de AOP, para lograr la modularización y posterior reutilización de los cambios mencionados.

El estudio analizado, comienza con la explicación del concepto de cambio, como algo inevitable y como parte de la evolución del software. Se mencionan los altos costos de manejar cambios y tambien el impacto que los mismos pueden tener en el sistema que los sufre. Según los autores, los cambios son alteraciones en el comportamiento base de una aplicación. En algunas ocasiones es necesario revertir algunos cambios, por lo que sería ideal si tales cambios fuesen expresados de manera "pluggable ". Es inevitable comparar este concepto con la concepción de funcionalidad volátil en esta tesis. Se mencionó ya varias veces (Ver sección 2.3) el caracter temporal de la funcionalidad volátil y la importancia de la modularidad y reutiliación de la misma. Bebjak et. al. argumentan que la implementación modular de cambios, para su integración con un conjunto acotado de aplicaciones, puede realizarse considerando a los cambios como aspectos.

En este paper utilizan como base aplicaciones desarrolladas a partir del patrón de diseño MVC, al igual que en esta tesis (Ver sección 4.2.1). La justificación es que la estructuración provista por dicho patrón, es adecuada para utilizar AOP.

El foco principal del estudio analizado es la utilización de patrones de AOP para lograr la implementación de cambios. Nuevamente, insisten en la utilización de dichos patrones recalando en la potencial reutilización de los cambios. Uno de los patrones presentados analiza el caso en que se deben efectuar acciones luego de que un evento haya ocurrido. El patrón presentado consiste en la implementación del comportamiento, a ejecutar como consecuencia del evento, como código adicionál que se inserta en el flujo normal del programa, luego de que un método predefinido sea llamado. Estp se corresponde con el análisis de Method Wrappers de esta tesis.

Los autores realizan una clasificación de los posibles cambios que una aplicación web puede sufrir. Se presenta un tipo de cambio particular concerniente a la relación que una aplicación puede mantener con otras aplicaciones. La comunicación con aplicaciones de terceros es presentada como un problema estudiado, siendo los cambios en tales relaciones algo necesario de manejar. Lo que vale resaltar de esta parte del estudio es la realización de dos tipos de comunicaciones o

integraciones entre aplicaciones. Los autores introducen el concepto de interación de una vía (one way integration) que consiste en una aplicación siendo integrada con otras. En tal tipo de integración, la información solo fluye en un sentido, de la aplicación a integrar a sus pares. El segundo tipo de integración, necesita no solo que la aplicación integrada se comunique con sus pares, sino que los mismos se comuniquen con la aplicación. Estos conceptos tienen relación con uno de los ejes de la arquitectura presentada en esta tesis. La comunicación de dos vías, o comunicación bidireccional se planteó en la sección 4.2.3 como uno de los ejes necesarios para la integración completa de la funcionaldiad volátil.

Bajo el nombre *cambios* o *funcionalidad volátil* ambos trabajos atacan el mismo problema: La evolución de aplicaciones web o cómo integrar nueva funcionalidad, temporalmente, para luego removerla sin afectar el sistema base. Finalmente, no es sorpresa que se solucionen en los dos trabajos muchas cosas de la misma manera. Para concluir con este pequeño análisis se remarca la importancia que el trabajo da al concepto de cambio como elemento modular reutilizable, incluso *pluggeable*.

6.7. Rossi et. al. 2006-2009

En esta sección se analizarán los trabajos principales sobre los cuales se basó parte de esta tesis.

6.7.1. Model-Based Design of Volatile Functionality in Web Applications.

Este paper es uno de los pilares sobre los cuáles se fundó el trabajo de esta tesis. En este trabajo y en el paper *"Designing Volatile Functionality in E-Commerce Web Applications"* [RNMS06] Rossi et. al. presentan el concepto de funcionalidad volátil. En ambos trabajos el foco está centrado en el diseño conceptual y navegacional en el contexto de OOHDM [Ros96]. El trabajo propuesto consiste en una extensión de OOHDM y un framework que soporta la incorporación de funcionalidad volátil en el modelo de diseño.

Rossi et. al. discuten los problemas de incorporar funcionalidad volátil editando el diseño, a medida en que esta surge, e implementando directamente sin tener en cuenta los procesos previos usuales. Estos problemas fueron presentados en la sección 2.4. Como una alternativa a las estrategias recién mencionadas, proponen una aproximación que se basa en el diseño y modelado de la funcionalidad volátil en forma separada, utilizando OOHDM.

Los autores proponen una estrategia para el manejo de la funcionalidad volátil basada en el desarrollo a partir de modelos (model-driven approach.) y en los conceptos separation of concerns e inversion of control. El proceso de diseño de la funcionalidad volátil se desarrolla de acuerdo a las etapas especificadas en OOHDM.

La aproximación consiste en considerar, y por ende diseñar, hasta las más básicas de las funcionalidades volátiles como ciudadanos de primera clase (por ejemplo, clases). Al mismo tiempo la funcionalidad volátil debe estar completamente desacoplada de las clases de la aplicación core. Las clases core y volátiles deben ser integradas de manera consistente en tiempo de ejecución de manera tal que luego puedan ser fácilmente removidas. Para lograr la integración, se propone utilizar una especificación de integración y un motor (engine) de integración. Algunos puntos fuertes de la aproximación propuesta son:

- Los servicios volátiles y la aplicación core son desacoplados introduciendo una capa de modelado específica para la funcionalidad volátil (Volatility Layer), que comprende los modelos conceptual, navegacional y de interfaz para cada concern volátil.
- La nueva lógica, perteneciente a la capa de funcionalidad volátil, es modelada en el modelo conceptual de dicha funcionalidad. Tal lógica o funcionalidad es considerada como una combinación de Commands y Decorators.
- Se utiliza IOC para lograr obliviousness: En lugar de que las clases del modelo conceptual de la aplicación core conozcan a los nuevos features, se invierte la relación de conocimiento. Las nuevas clases conocen las clases base, sobre las cuales son construidas. Por lo tanto los modelos core no son modificados y no tienen conocimiento acerca de las adiciones.
- El modelo navegacional core es agnóstico de las clases navegacionales de la funcionalidad volátil. Es decir, no hay links u otras referencias desde la capa core a la capa volátil.
- Se utiliza una especificación de integración separada de la funcionalidad volátil que especifíca la forma en que la funcionalidad es acoplada con la aplicación core. A dicha especificación se la conoce como “Affinity”. Se define mediante esta especificación, las instancias de nodos navegacionales que son afectadas por la funcionalidad volátil y la manera en que el modelo navegacional es extendido.
- Se diseña e implementan las interfaces correspondientes a cada funcionalidad de manera separada. Nuevamente los elementos de la capa core son ajenos a los nuevos elementos de la capa volátil.

Los autores proponen diseñar y modelar todo tipo de funcionalidad volátil utilizando técnicas de ingeniería de software. La estrategia propuesta se basa en mantener el diseño e implementación de las funcionalidades volátiles separados de la aplicación core. Para lograr esto proponen diseñar los servicios volátiles en una capa separada del diseño de los componentes core. La funcionalidad volátil es

modelada en la capa de funcionalidad volátil utilizando combinaciones del patrón Command y Decorator. Los servicios volátiles pueden ser interpretados como commands debido a que cada servicio encuadra en una clase comportamiento de la aplicación. También pueden ser decorators al añadir nueva funcionalidad o propiedades a la aplicación de manera no intrusiva. Finalmente, además de las capas mencionadas, se provee una capa de integración destinada a unir las capas core y de funcionalidad volátil. Al hacer esto, según los autores, se aumenta la reusabilidad de los elementos core y volátiles.

La funcionalidad volátil es integrada a la aplicación utilizando especificaciones de integración. Las mismas son desacopladas de los servicios volátiles y de los componentes core, aumentando la propiedad de obliviousness. La especificación se logra a partir de un lenguaje de consultas de OOHDM e indica cómo serán aumentados los modelos navegacionales con la funcionalidad volátil. El conjunto de nodos afectados por un servicio volátil es denominado Affinity de la funcionalidad. Al mantener la especificación de affinities, o de la integración de la funcionalidad volátil, desacoplada de la propia funcionalidad, se logra la posibilidad de utilizar distintos esquemas de integración. Además, esto permite la evolución independiente de la funcionalidad volátil y de la aplicación host y también de las reglas de integración.

Proponen una arquitectura que extiende Cazon[RNM+06] y Struts. La arquitectura presentada en el paper posee tres capas principales, componentes core, servicios volátiles y capa de integración. A través de extensiones al framework Struts, logran alterar el flujo normal de procesamiento de requerimientos http para alterar la aplicación integrando la funcionalidad volátil. Como se mencionó antes, la comunicación provista entre los módulos volátiles y la aplicación core se basa en mecanismos de commands y decorators, por lo que el flujo de información es desde los servicios volátiles hacia los módulos core. Finalmente proveen un sistema de construcción de lógica de integración a partir de las especificaciones plasmadas en xml.

Esta tesis se basó en los conceptos planteados en el paper en cuestión y como consecuencia gran parte de los líneamientos de este trabajo es compartida. La filosofía de mantener la funcionalidad volátil desacoplada, considerándola parte de un proceso completo que comprende modelado e implementación es heredadada de este paper. De la misma manera, en ambos trabajos se hace incapié en la reutilización de la funcionalidad volátil, aunque en esta tesis este es uno de los aspectos cruciales que determina la arquitectura resultante. Algunas diferencias relevantes surgen a partir del foco de cada trabajo. Esta tesis se centra en cuestiones de implementación que fueron pensadas para lograr funcionalidad volátil integrada en RIAs manteniendo un marco de comunicación complejo. En el paper de Rossi et. al. el esfuerzo se centra en conceptos de modelado, diseño conceptual y navegacional. La arquitectura propuesta por los autores no tiene en cuenta

las características de comunicación entre componentes planteadas en este trabajo. Las especificaciones de integración propuestas por los autores solo abarcan los elementos que serán afectados, pero no tienen en cuenta las condiciones o el punto en la ejecución de la aplicación en el que las mismas deben ser evaluadas.

Finalmente, el mensaje que promueven los autores es mantener los elementos core agnósticos de la existencia de la funcionalidad volátil, sin importar cómo el proceso de integración es finalmente llevado a cabo. Con respecto a esto, el mensaje de esta tesis introduce una nueva parte: El objetivo es poder desarrollar elementos volátiles, manteniento obliviousness en ambas partes, core y volátil para poder aumentar la retulización de los elementos volátiles.

6.7.2. Modeling and Deploying Volatile Functionalities in Web Applications

Este trabajo es la continuación del trabajo analizado anteriormente. Introducen mejoras en el estudio de la funcionalidad volátil y en el approach propuesto para su administración. A lo explicado anteriormente, añaden un análisis de los atributos de la funcionalidad volátil y definen un posible ciclo de vida de la misma de acuerdo a uno de sus atributos: volatitlity pattern. Los autores además presentan mejoras en el framework Cazon, que permiten una automatización de los procesos de integración de la funcionalidad.

En este trabajo se presenta un análisis de los atributos de la funcionalidad volátil. Se presentan los atributos Intent, Extent y Volatility Pattern, explicados en la sección 2.2.2. Los autores presentan un lenguaje basado en eventos para especificar la activación de la funcionalidad volátil de acuerdo al atributo Volatility Pattern. A partir de este atributo, los autores explican las posibles alternativas que determinan el ciclo de vida de una funcionalidad volátil. Se explicó en la sección 2.2.2 que en esta tesis se diferencian los procesos de conexión y de activación, en relación al atributo Volatility Pattern. Así, se introdujo en esta tesis el atributo Connexion Pattern. De la misma manera, se presentó una modificación el ciclo de vida que tiene en cuenta el nuevo atributo.

En la aproximación propuesta en esta tesis, se diferencian los procesos de activación/deactivación con los procesos de acople/desacople (o instalación/desinstalación, integración/desintegración). El proceso de instalación consiste en incorporar la funcionalidad volátil a la aplicación core y es equivalente al proceso de activación del trabajo de Rossi et. al. El proceso de activación de mi trabajo consiste en hacer disponible o activa la funcionalidad en la aplicación ejecutandose. La aproximación de Rossi hace énfasis en la automatización del proceso de instalación mientras que mi aproximación se enfatiza en poder especificar las condiciones necesarias para los procesos de activación y desactivación. Mediante el desarrollo de un sistema de condiciones y eventos que se aplica a las funcionalidades volátiles y a la aplicación core, se puede activar y desactivar las funcionalidades

volátiles (como también realizar otro tipo de acciones, como configuración). Mi aproximación se basa en esa idea sin hacer enfasis en la automatización de la activación y desactivacióny permite comunicar las partes completamente, en ambas direcciones. Se promueve el diseño e implementación de f.v. como plugin para su reutilización.

Los autores proponen un DSL para la especificación de reglas de activación que permitan llevar a cabo lo descrito por el Volatility Pattern. Mediante el uso de dicho lenguaje promueven la especificación de reglas basadas en eventos de la aplicación o reglas de caracter temporal. Así se puede especificar cuándo la aplicación puede ser modificada. En relación a esto, el trabajo desarrollado en esta tesis, no promueve un DSL pero permite especificar con detalle, a través del uso de Eventos y Condiciones el momento en que la funcionalidad volátil es activada y los puntos en los cuales la misma es actualizada.

6.7.3. Oblivious Integration of Volatile Functionality in Web Application Interfaces

Los autores de este paper [GDRU09] presentan una continuación a la aproximación para lidiar con funcionalidad volátil presentada en [RNM+06]. Basándose en las nociones explicadas con anterioridad sobre su aproximación, los autores mantienen el foco del trabajo en cómo extender tales ideas para solucionar el problema de polución de código y diseño a nivel de presentación. El trabajo en cuestión se centra en la composición de interfaces web haciendo incapié en mantener el concepto de "obliviousness". El objetivo es que los componentes core se mantengan ajenos (oblivious) de la existencia e integración de los componentes volátiles. Proponen una aproximación donde ambas interfaces, core y volátil, sean ajenas entre si y puedan ser compuestas de forma coherente y consistente. Argumentan que es posible crear interfaces específicas para cada concern (funcionalidad volátil y aplicación core, por ejemplo) que sean compuestas mediante especificaciones de integración.

La aproximación expuesta en el trabajo en cuestión, hace incapié en el diseño e implementación por serparado de las interfaces web, para la posterior composición de las mismas. La composición de interfaces es lograda a partir de la especificación de reglas de integración llamadas *affinities*. Cada affinity se especifica utilizando un lenguaje de queries definido como parte de OOHDM [SR98]. Luego de la especificación, las reglas son interpretadas por un engine que puede ejecutar tales reglas, almacenadas en archivos XML, y aumentar el contenido de los nodos con los atributos, links o lo que sea necesario. Los autores remarcan que al contar con tales formas de especificar la integración y con un engine preparado para procesar las especificaciones, solo es necesario modificar las especificaciones para lograr alterar la manera en que se integra la funcionalidad en la vista. Por ejemplo, se puede cambiar su posición o incluso la página en la que se muestra.

Como la integración de interfaces sucede en tiempo de ejecución, explican los autores, es posible configurar la aplicación en runtime. Lo mismo ocurre en la aproximación utilizada en esta tesis. En la sección 4.5 de esta tesis, se mencionó una herramienta construida para la configuración dinámica de aplicaciónes host, agregando y quitando funcionaldiad volátil en runtime. Esto es posible debido a que tanto la lógica de integración de modelos como la integración de interfaces se hace en tiempo de ejecución.

Con respecto a los aspectos de implementación de la integración de interfaces web, este estudio uitiliza una extensión a un framework basado en Struts, Cazon. Las tecnologías utilizadas son Java, JSP, XML para especificar los contenidos de las vistas y XSLT para finalmente especificar las formas de componer los contenidos. Las tecnologías utilizadas en esta tesis, como se mencionó anteriormente, son Smalltalk, Seaside, DOM, Javascript y CSS. DOM [DOM] permite acceder y actualizar contenido, estructura y estilo de la de interfaz dinámicamente. Las técnicas utilizadas ya fueron explicadas.

La integración efectuada a nivel de interfcaces en el desarrollo efectuado en esta tesis, es consecuencia directa del modelo de producción de interfaces de Seaside y de las decisiones de diseño consecuentes de la arquitectura propuesta. Al desarrollar por separado los modelos y Componentes Seaside de la aplicación core y de los módulos volátiles, la integración se logra mediante la posterior composición de los mismos en tiempo de ejecución. Tal composición requiere de un marco que permita introducir ambos componentes en el mismo proceso de renderización html. Luego de integrar en un mismo documento html los componentes core y volatile se puede dar forma o estilo utilizando CSS. Básicamente se componen los componentes seaside de la aplicación core y de la funcionalidad volátil para conseguir una estructura html que será desplegada y estilizada luego utilizando CSS. La integración final entonces debe ocurrir en las hojas de estilo, por lo que las mismas no pueden ser independientes. En la implementación alcanzada, fue necesario alterar las hojas de estilo para poder dar forma y posición correcta a los nuevos elementos incluidos como consecuencia de la funcionalidad volátil. Por supuesto el concepto de obliviousness se puede estresar solo hasta un punto. En el caso de esta tesis, dicho punto fueron las hojas de estilo.

Finalmente, para concluir el estudio y la comparación con este trabajo, se puede remarcar que hay similitudes inherentes a la naturaleza de esta tesis. Esta tesis es un trabajo derivado del trabajo de [RNM+06, RGDU08], al igual que el paper estudiado en este apartado. No obstante, los focos de los trabajos son distintos, siendo el de este paper: la composición de interfaces para aplicaciones web. Con respecto a esto (composición de interfaces) en esta tesis se estudió el problema desde el punto de vista del desarrollo de aplicaciones RIA.

6.8. Conclusión trabajo relacionado

AOP está intrínsecamente relacionado con el estudio presentado aquí debido a que la funcionalidad volátil, generalmente, puede aparecer de hecho como crosscutting concerns [BVD07, GDRU09, RNM+06]. Es por eso que varios de los trabajos presentados son de este area de estudios. Resulta interesante encontrar que otros estudios recalen en las mismas necesidades, como comunicación de dos vías o necesidad de manejar la funcionalidad volátil de manera modular para su posterior reutilización. Siendo este último punto una constante en varios de los trabajos, este hecho no es algo extraño ya que su objetivo final es la disminución de costos. La mayor parte del trabajo relacionado consiste en el desarrollado por Rossi y la segunda parte está comprendida por el trabajo de AOP para resolver concerns volátiles o resolver el cambio en el software. Todos los trabajos encaran el mismo problema de fondo: como lidiar con la evolución inminente del software de manera exitosa

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Introducción

En este documento se presentó el problema del manejo de funcionalidad volátil en aplicaciones web RIA en Smalltalk, utilizando Ajax mediante Seaside. Se introdujeron los conceptos teóricos relacionados con el término "Funcionalidad Volátil", como sus atributos (Extent, Intent, Volatility Pattern) y las incumbencias prácticas inherentes a la incorporación y eliminación de este tipo de funcionalidades en sistemas web. Luego se presentó el contexto en el que se desarrolló el estudio, los juegos de casino web, introduciendo el concepto de side game como funcionalidad volátil y los prototipos que se utilizaron para evaluar el problema. Una vez explicado el problema y el contexto donde se lo analizó, se introdujo una arquitectura para dar soporte al manejo de funcionalidad volátil y se dio un panorama general de la misma. Se presentó la descripción de cada componente de la arquitectura y un análisis exhaustivo de de sus ventajas, desventajas y rol en la arquitectura. Se estudió el trabajo relacionado especificando similitudes y diferencias en los puntos de vista teóricos y prácticos de las distintas aproximaciones. A continuación se presenta un pequeño análisis del trabajo desarrollado, de las conclusiones y de los resultados obtenidos.

7.2. Conclusiones

El trabajo de grado comenzó con el diseño e implementación de un prototipo de juego de casino web en Smalltalk. Debido a las actividades desarrolladas en relación a la empresa ID-Interactive, dicho prototipo fue construyéndose de acuerdo a las características encontradas en juegos reales de máquinas de casino. La complejidad del prototipo, en relación a la persistencia de eventos y capacidad de recuperación frente a caídas, hizo que el mismo se convierta en una aplicación

completa. La misma serviría para la incorporación de funcionalidad volátil y su posterior análisis como parte de este trabajo.

Una vez obtenida una versión funcional de la aplicación central, el juego de slotmachine, se prosiguió a implementar juegos secundarios que eventualmente serían analizados como funcionalidad volátil. El trabajo desarrollado en los prototipos, fue basado en una problemática real. Dicha problemática proviene del contexto de un proyecto de investigación y desarrollo sobre el campo de los juegos de casino. En dicho proyecto se necesitaba implementar versiones web de slotamachines y side games como elementos acoplables. Todas las decisiones tomadas en la construcción de las aplicaciones tuvieron bases en tales premisas y en la consideración de side games como posible funcionalidad volátil. Es por esto que los prototipos tienen una complejidad asociada considerable.

Las primeras implementaciones de los prototipo no fueron incluidas como elementos de funcionalidad volátil debido a la complejidad asociada. Eventualmente, mediante el continuo refactoring y una actividad muy fuerte de testing, se fueron convirtiendo los prototipos de juegos secundarios en instancias de funcionalidad volátil. La capacidad de acoplar y desacoplar los juegos secundarios de la aplicación host fue implementada paulatinamente. Al progresar, se construyó también un pequeño framework para la construcción de side games como plugins. Finalmente, los prototipos estuvieron listos para ser analizados. De esta forma, evaluando el panorama general obtenido, se fue elaborando una posible arquitectura.

El delineamiento de la arquitectura presentada fue paulatino y acompañado con modificaciones y refactorings sucesivos sobre los prototipos construidos. Con cada paso de la arquitectura los prototipos debieron ser modificados. Se logró obtener una implementación de la arquitectura, a partir de las aplicaciones desarrolladas en un principio y mantenidas mediante su refactoring continuo. Es necesario mencionar que el soporte brindado por una batería de tests muy completa, fue esencial para poder refactorizar una y otra vez las aplicaciones. Mediante el sucesivo refactoring se logró obtener una versión aceptable de la arquitectura, sobre la cuál se escribió este documento.

Se desarrolló un trabajo completo que concluyó en una arquitectura que permite manejar funcionalidad volátil en Aplicaciones Web Smalltalk con Seaside. Los puntos fuertes de la arquitectura presentada son el manejo de comunicación bidireccional y la implementación por separado de los distintos componentes. La arquitectura permite el diseño e implementación por separado de los componentes volátiles y la aplicación core. Junto con la comunicacion bidireccional y la integración transparente, la arquitectura posee buenos fundamentos que permiten el manejo de funcionalidad volátil.

Se implementó una herramienta que permite la instalación dinámica de la funcionalidad volátil, de acuerdo a ciertas reglas diseñadas e implementadas en

el sistema. La herramienta permite al operador instalar side games en el juego principal de manera fácil y en tiempo de ejecución, mientras el sitio sigue operando. La herramienta fue desarrollada gracias a la utilización de esta arquitectura y permite la incorporación de funcionalidades volátiles sin alterar el código de la aplicación principal, permitiendo restaurar el comportamiento de la misma en cualquier momento, en tiempo de ejecución.

Una vez concluido el proceso de desarrollo se inició el análisis de lo implementado y la construcción de este documento. Se estudió el problema de funcionalidad volátil en profundidad. Se incluyó un análisis de los elementos teóricos relacionados con el concepto. El trabajo de Rossi et. al. proveyó las bases para dicho análisis. Se estudió y se explicó el contexto sobre el cuál se realizaron las pruebas. El contexto constituye no solo los juegos de casino sino también el entorno donde se los programó: Seaside y Smalltalk. Nuevamente, las necesidades planteadas debido al proyecto en el que este trabajo se desarrolló, aportaron una cuota de realismo y complejidad considerable al desarrollo. Como resultado del desarrollo, la arquitectura propuesta fue analizada y se presentó en extensión en este documento.

Se explicaron exhaustivamente los objetivos y las bases sobre las cuáles se construyó la arquitectura presentada. Se detalló cada componente de la arquitectura y se presentó un análisis de su rol, ventajas y desventajas. Esto constituye un estudio completo de la solución propuesta.

Como complemento de los aspectos teóricos, del contexto y del estudio mismo de la arquitectura, se presentó un análisis extensivo del trabajo relacionado. Por supuesto, como se mencionó en dicha sección, resulta imposible análizar toda la bibliografía relacionada. No obstante, no solo se estudió el trabajo tomado como base, de Rossi et. al., sino también trabajos menos relacionados, del mundo de AOP y otros.

Finalmente, este trabajo constituye un gran esfuerzo, cuyo desarrollo fue a conciencia y extenso; generando como resultado un conjunto de mecanismos complejos para el manejo de la problemática. Se presentó un extenso análisis del tema de estudio, su contexto, el trabajo relacionado y la solución propuesta.

7.2.1. Conclusiones sobre la arquitectura presentada

Se obtuvo una arquitectura que permite desarrollar aplicaciones con soporte de funcionalidad volátil. Se hace énfasis en la capacidad de integrar funcionalidades volátiles completamente integradas con la aplicación host (Two Way Integration [BVD07]). Junto con el diseño por separado de los componentes volátiles de la aplicación core, fue uno de los ejes más importantes y tenidos en cuenta de la arquitectura.

La arquitectura presenta componentes que elevan el nivel de abstracción del programador permitiendo una integración de la funcionalidad volátil más natural.

Las condiciones, al igual que los Eventos y Pointcuts, son artefactos que permiten integrar más facilmente las funcionalidades. El programador puede epecificar situaciones complejas utilizando elementos de alto nivel de abstracción. La utilización de Condiciones facilita la reutilización de elementos de integración de funcionalidad. Por ejemplo, un conjunto de condiciones sobre una aplicación host puede constituir una librería que puede ser compartida por el proceso de integración de distintas funcionalidades volátiles. Junto con los eventos y Pointcuts es posible especificar momentos en la ejecución del programa para establecer políticas de actualización o activación, utilizadas por updaters. Es importante destacar la importancia del aumento de la capacidad de expresión que brindan estos mecanismos, al igual que una mayor legibilidad. Leer código que utiliza estos objetos (con nombres autoexplicativos y que engloban lógica compleja de evaluación de estado) resulta mucho más natural y brinda mayor fluidez.

7.2.2. Aporte al trabajo de Rossi et. al.

Estando basada en el trabajo de Rossi et.al. es importante mencionar cuáles fueron los aportes a dicho trabajo.

Nuevo atributo: Connexion Pattern

Como se mencionó el en capítulo 2.2.2, en [RGDU08] se utilizan los conceptos de activación e instalación de la funcionalidad volátil de manera intercambiable. No se hace distinción entre los procesos de conexión y desconexión de la funcionalidad con los procesos de activación y deactivación de la misma. A partir del trabajo desarrollado en esta tesis, se encontró que la funcionalidad volátil debe ser instalada en el sistema primero, para poder luego cumplir con su patrón de volatilidad, es decir: ser activada o desactivada. Como consecuencia, aquí se planteó como necesario identificar este proceso y diferenciarlo del proceso de activación de la funcionalidad. Esto generó la necesidad de incorporar un nuevo atributo a la funcionalidad volátil: el atributo Connexion Pattern.

Por ejemplo, un side game es instalado como funcionalidad volátil sobre un juego host y luego, cuando ocurre cierto evento del juego host, el side game es ofrecido al usuario (es activado). En este ejemplo se puede diferenciar el estado de conexión o acople de la funcionalidad volátil (Connexion Pattern) con el estado de disponibilidad o activación (Volatility Patter). Una funcionalidad volátil puede estar instalada en un sistema pero todavía no estar activa, esperando que ocurra algo en el sistema host para activarse. La funcionalidad volátil puede estar desconectada, esperando a que alguna regla de negocio se cumpla o ocurra algun evento de caracter temporal. Una vez que la funcionalidad se acopla al sistema, puede ocurrir que la misma esté en estado pasivo, esperando algún evento de la aplicación host. Así, cuando la funcionalidad volátil ya está acoplada al sistema,

pueden ocurrir los cambios de estado entre activo y pasivo como se explicó anteriormente. Finalmente, en algún momento la funcionalidad se desacoplará de la aplicación host, dejándola como antes.

Las diferencias entre los atributos son sutiles, pero se hacen más visibles al considerar los procesos inherentes a cada uno. Es importante poder diferenciar claramente estos procesos para lograr un mejor diseño, más claro, que permita una intergación correcta de las funcionaldiades volátiles. Justamente, considerar estos atributos por separado genera mayor claridad en la especificación de la mecánica de integración de las funcionalidades volátiles.

Evaluación práctica en un contexto nuevo: Smalltalk + Seaside

El trabajo de Rossi et.al. se centra en aspectos conceptuales basados en modelos. No obstante también estudian aspectos prácticos de la incorporación de funcionalidad volátil. Tal estudio se centra en Java y Struts, utilizando tecnologías auxiliares como XML y XSLT para definir atributos y componer vistas. El trabajo presentado en esta tesis constituye un aporte al estudio de la funcionalidad volátil en un entorno diferente al estudiado por Rossi et. al. Si bien Smalltalk es uno de los lenguajes pioneros en programación orientada a objetos, el framework Seaside está en ebullición. Es por esto que un análisis actualizado del tratamiento de la funcionalidad volátil en este entorno resulta un aporte interesante al estudio de Rossi et. al.

Alternativas para la arquitectura de Rossi: Comunicación de dos vías.

En el trabajo de Rossi et. al. logran comunicación en un solo sentido mediante la utilización de los patterns Decorator y Command. En este trabajo se propuso un esquema de comunicación basado en el pattern Observer [GHJV94]. Cuando un evento ocurre en alguna de las partes (core o volátil), los cambios son anunciados y mediante observers los objetos pertinentes son notificados. De esta forma se logra comunicación bidireccional y desacople entre la aplicación host y las funcionalidades volátiles. Particularmente, las funcionalidades volátiles pueden ser notificadas de los cambios que acontecen en la aplicación core. Esto constituye un aporte desde el punto de vista implementativo para el estudio en cuestión.

7.3. Trabajo Futuro

En esta sección se describe el trabajo que por cuestiones de tiempo y de alcance de esta tesis no fue abarcado en este estudio.

7.3.1. Integración de acciones de bases de datos

La aplicación host desarrollada, sobre la cuál se incorporaron los prototipos de funcionalidad volátil, presenta una fuerte actividad de base de datos. Tal intendsidad proviene de la necesidad de persistir todos los eventos del juego principal, para poder proveer recuperación automática ante caidas. La incorporación de funcionalidad volátil, que potencialmente puede alterar el flujo de la aplicación host, presenta un problema que en esta tesis no fue atacado completamente: el impacto de la funcionalidad volátil en la capa de persistencia de la aplicación host.

En el prototipo desarrollado para la aplicación host se proveyó un sistema de persistencia que almacena todos los cambios que ocurren en el juego, en el momento adecuado. El objetivo de tal sistema es poder contar con un registro de las actividades del juego para poder brindar al jugador la posibilidad de recuperar sesiones de manera automática. De tal manera no se pierde consistencia en los créditos de la cuenta del jugador. Esto es un requisito que proviene de la naturaleza de los prototipos estudiados: juegos de casino. La incorporación de los juegos secundarios como elementos de funcionalidad volátil y su participación en el ciclo de jugadas del jugador, genera cambios en la actividad del conjunto del juego principal y de los juegos secundario como un todo. Los juegos secundarios intervienen en la secuencia de eventos generada por el juego principal y modifican la cuenta del jugador a medida en que otorgan premios y cobran créditos. Así, para que el juego mantenga su característica de consistencia con respecto a las caídas y la actividad en las cuentas de los usuarios; es necesario proveer una solución al problema de la interferencia en los eventos del juego principal.

Este problema no fue atacado en esta tesis. La dificultad consiste en asignar un espacio en el diseño de la base de datos para almacenar los eventos, no previstos, generados por los juegos secundarios (la funcionalidad volátil). Así, la incorporación de funcionalidad volátil introduce un problema de diseño de base de datos. Además de los problemas de diseño de modelos de bases de datos, existen cuestiones referentes a aspectos más concretos, como la modificación de las secuencias de eventos generada por el juego.

Es necesario que los eventos generados por la funcionaldiad volátil, estén coordinados con los eventos generados por la aplicación host, para mantener la consistencia mencionada anteriormente. La incorporación de nuevos eventos puede alterar la lógica de recuperación del estado del juego principal. Sería necesario entonces analizar la forma en que el juego principal se comportará al interactuar junto con un juego secundario.

Supongamos que un jugador utiliza el juego principal, por lo que se persisten los eventos de spin y los premios otorgados. A partir de dicho spin, en el que el jugador gana α créditos, se activa el side game Double or Nothing. El jugador decide jugar al side game y al hacerlo gana un premio, α x 2 créditos. Como el

jugador gana el side game, se vacían los premios de la slotmachine y se aumentan sus créditos en α x 2 créditos. Ahora bien, pueden ocurrir distintos escenarios:

- El jugador puede realizar logout, por lo que se persistirá la slot con los créditos nuevos (α x 2). En tal caso, en la base de datos no existirá un evento que justifique tal incremento en los créditos. Este caso es un inconveniente menor pero muestra inconsistencias.

- El jugador puede abandonar la slot navegando a otro sitio y luego volver a ingresar. En este caso el sistema tratará de recuperar la slot perdida por el jugador. Lo que ocurrirá será que se recuperará la slot y no se verán los créditos ganados por el side game, se visualizará el juego justo como estaba antes de jugar el side game. Esto no es deseable.

Hay más escenarios posibles en este ejemplo, pero los mencionados sirven para afirmar que es necesario encontrar una solución.

En el caso de juegos de casino reales, que funcionan sobre máquinas dedicadas, este problema no es menor. En tal contexto, existen restricciones que van más allá de mantener la consistencia con los datos del jugador (sus créditos). Existen restricciones impuestas por tiempos de respuesta, reacciones esperadas en servidores dedicados que escuchan por los eventos del juego y demás. En este tipo de juegos se logró incorporar juegos secundarios simulando sus jugadas como si fuesen jugadas del juego principal. De esta forma no fue necesario alterar el esquema de eventos ni la lógica que lo administra. Esta podría ser una solución aplicable a los casos de estudio de esta tesis. Los juegos secundarios no generarían información en la base de datos directamente, sino que la misma sería generada por el juego principal, con sus mecanísmos originales. No obstante esta posible solución para el caso particular estudiado, el problema es más general y sigue pendiente: como incorporar funcionalidad volátil que altera procesos internos de la aplicación host.

7.3.2. Composición avanzada de interfaces gráficas

En este trabajo se realizó la composición de GUIs de la aplicación host y de las funcionalidades volátiles utilizando mecanismos provistos por seaside. Mediante decoraciones de componentes, se pudo componer fácilmente la estructura final del conjunto: aplicación host + side games. Smalltalk permite la composición utilizando mecanismos orientados a objetos incluso a nivel de generación de contenido HTML, que determina la estructura de la vista final. Se logró implementar por separado las vistas de la aplicación host y de las funcionalidades volátiles para luego unirlas en un esquema de composición simple. La composición consistió simplemente en la generación de distintos contenedores HTML (DIV) que separan los contenidos del juego principal de los juegos secundarios.

Finalmente la última etapa de la composición de las vistas consistió en modificar las hojas de estilo (CSS) que dan la apariencia final de la aplicación. En el trabajo relacionado, esto fue realizado manualmente. El juego principal tiene una hoja de estilo asociada que determina su presentación. Para incorporar side games, fue necesario alterar dicha hoja de estilo para poder incluir los atributos de los juegos secundarios. Fue necesario incorporar atributos que no entren en conflicto con los atributos del estilo del juego principal. Este fue el punto donde la composición de funcionalidades volátiles con la aplicación host dejó de mantener la propiedad de obliviousness. Como trabajo pendiente queda el estudio de cómo generar hojas de estilo automáticamente, mediante la combinación de los estilos del la aplicación host y de los componentes volátiles incorporados.

7.3.3. DSL para definición de Volatility Pattern y Connexion Pattern via XML.

Mediante la definición del atributo connexion pattern utilizando un lenguaje DSL diseñado a tal efecto, se podría describir el punto de instalación y automatizar el proceso. En [RGDU08] los autores presentan un DSL que permite definir el atributo Volatility Pattern. Con el DSL propuesto se puede definir los puntos de activación, una vez instalada la funcionalidad. Mediante el lenguaje se pueden hacer referencia a puntos en el tiempo o en la ejecución del programa principal. De la misma manera en que se puede definir el atributo Volatility Pattern se podría extender el DSL para poder automatizar el proceso de conexión de la funcionalidad.

En esta tesis no se proveyó una forma de especificar los atributos Volatility Pattern y Connexion Pattern de manera que permita su automatización. Una forma útil sería implementar el DSL mediante XML, como lo hicieron los autores de [DMM+07].

7.3.4. Estudio de interacción entre funcionalidades volátiles.

Como parte de los prototipos desarrollados se incluyeron dos side games: Double or Nothing y Card Wars. Se logró integrar dichos side games con el juego principal para que ambos funcionen de manera completa. Lo que no se consideró es cómo afecta la existencia del side game Doble o Nada al side game Card Wars. Este problema es un caso particular de una cuestión más general: Cómo afecta la existencia de una nueva funcionalidad volátil a otra.

Si dos side games dependen de un premio y uno de ellos lo consume, es necesario deactivar el segundo?. Qué pasa si no hay un mecanismo en el juego principal que avise al segundo juego que el premio fue consumido por un tercero?. Todas estas cuestiones no fueron analizadas en profundidad. Como trabajo pendiente sería necesario evaluar la posibilidad de incluir atributos de interacción entre funciona-

lidades volátiles, que indiquen o describan la manera en que otras funcionalidades afectan su comportamiento. Definir una manera de especificar los aspectos de la aplicación host sobre los cuales la funcionalidad volátil está interesada, y que la misma sea notificada cuando alguno de ellos cambia; sería otra posibilidad. Para esto sería necesario evaluar nuevamente, la necesidad de un nuevo atributo para la parte teórica. Con respecto a la parte práctica, sería necesario incorporar mecanismos en la capa intermedia que ataquen estos problemas.

Bibliografía

[AT98] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. *Aspect-Oriented Programming workshop*, 1998. [cited at p. 20]

[BKKZ05] H. Baumeister, A. Knapp, N. Koch, and G. Zhang. Modelling adaptivity with aspects. In *Proceedings of the 5th International Conference on Web Engineering (ICWE'05)*, volume Lecture Notes in Computer Science. Springer Verlag, 2005. [cited at p. 97]

[Bro87] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20:10–19, April 1987. [cited at p. 7]

[BVD07] Michal Bebjak, Valentino Vranic, and Peter Dolog. Evolution of web applications with aspect-oriented design patterns. volume Vol-267, Como, Italy, jul 2007. CEUR Workshop Proceedings, CEUR Workshop Proceedings. [cited at p. 54, 107, 114, 117, 135]

[BvDTvE04] Magiel Bruntink, Arie van Deursen, Tom Tourwé, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. *Software Maintenance, IEEE International Conference on*, 0:200–209, 2004. [cited at p. 134]

[Byk05] Vassili Bykov. Introducing announcements. Web publication, Cincom, 2005. Available at http://www.cincomsmalltalk.com/userblogs/vbykov/blogView?entry=3310034894. [cited at p. 54]

[CBL07] Dave Crane, Bear Bibeault, and Tom Locke. *Prototype and scriptaculous in action*, chapter 1, pages 3–25. Manning Publications Co., Greenwich, CT, USA, 2007. [cited at p. 26]

[CVWH07] Sven Casteleyn, William Van Woensel, and Geert-Jan Houben. A semantics-based aspect-oriented approach to adaptation in web engineering. In *HT '07: Proceedings of the eighteenth conference on Hypertext and hypermedia*, pages 189–198, New York, NY, USA, 2007. ACM. [cited at p. 97]

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Softw.*, 24(5):56–63, 2007. [cited at p. 27]

[DMM+07] Florian Daniel, Maristella Matera, Alessandro Morandi, Matteo Mortari, and Giuseppe Pozzi. Active rules for runtime adaptivity management. In *Proceedings of the 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering AEWSE'07*, Como, Italy, July 19 2007. [cited at p. 103, 122]

[DOM] Document object model. [cited at p. 113]

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems. (OOPSLA 2000)*, pages 21–35. Addison-Wesley, October 2000. [cited at p. 7, 81, 99, 100, 101, 140]

[Fow04] Martin. Fowler. Inversion of control containers and the dependency injection pattern. Technical report, January 2004. Available online at http://www.martinfowler.com/articles/injection.html. [cited at p. 45]

[Fow09] Martin Fowler. Rules engine. Web publication, 2009. Available online at http://martinfowler.com/bliki/RulesEngine.html. [cited at p. 59, 101]

[FY96] Brian Foote and Joseph Yoder. Evolution, architecture, and metamorphosis. In *Pattern Languages of Program Design 2*, Department of Computer Science. University of Illinois. Urbana-Champaign, 1996. University of Illinois, Addison-Wesley. [cited at p. 7, 8]

[FY97] Brian Foote and Joseph Yoder. Big ball of mud. In N.Harrison, B.Foote, , and H.Rohnert (Eds.), editors, *Pattern Languages of Program Design*, pages 653–692. Addison-Wesley, 1997. [cited at p. 7, 8, 41]

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications. Web publication, AdaptivePath.com, Feb 2005. Available online at http://www.adaptivepath.com/ideas/essays/archives/000385.php. [cited at p. 56]

[GCRFPL02] Lina García-Cabrera, María José Rodríguez-Fórtiz, and José Parets-Llorca. Evolving hypermedia systems: a layered software architecture. *Journal of Software Maintenance*, 14(5):389–405, 2002. [cited at p. 23]

[GDRU09] Jerónimo Ginzburg, Damiano Distante, Gustavo Rossi, and Matías Urbieta. Oblivious integration of volatile functionality in web application interfaces. *Journal of Web Engineering*, 8(1):25–47, March 2009. [cited at p. 11, 21, 22, 40, 42, 112, 114]

[GG01] Martin Gaedke and Guntram Graf. Development and evolution of web-applications using the webcomposition process model. In *Web Engineering, Software Engineering and Web Application Development*, pages 58–76, London, UK, 2001. Springer-Verlag. [cited at p. 97]

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Professional, illustrated edition edition, November 1994. [cited at p. 45, 53, 54, 85, 86, 119, 145]

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. Available online at http://portal.acm.org/citation.cfm?id=273. [cited at p. 83]

[Hir01] Robert Hirschfeld. Aspects - aop with squeak. *Workshop on Advanced Separation of Concerns in Object-Oriented Systems. (OOPSLA 2001)*, Oct 2001. [cited at p. 140]

[Hir02] Robert Hirschfeld. Advice activation in aspects. In University of Bonn, editor, *Proceedings of the 2nd Workshop of Aspect-Oriented Software Development by the GI SIG 2.1.9 (AOSD-GI)*. University of Bonn, February 2002. [cited at p. 133]

[HL95] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, 1995. Available online at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5223. [cited at p. 82]

[HOT06] William Harrison, Harold Ossher, and Peri Tarr. General composition of software artifacts. pages 194–210, 2006. [cited at p. 97]

[JBR98] R. Johnson J. Brant, B. Foote and D. Roberts. Wrappers to the rescue. In . Springer-Verlag, editor, *Proceedings ECOOP'98, LNCS 1445.*, page 396–417, 1998. [cited at p. 133, 140]

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Heidelberg, editor, *ECOOP 2001 - Object-Oriented Programming: 15th European Conference*, pages 327–253, Budapest, Hungary, June 2001. Springer Verlag. Available online at http://www.springerlink.com/content/mc9xermkrav48ff1. [cited at p. 47, 74, 133]

[Kni00] Alan Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174, New York, NY, USA, 2000. ACM. [cited at p. 25]

[KV08] G. P. Kulk and C. Verhoef. Quantifying requirements volatility effects. *Science of Computer Programming*, 72(3):136–175, 2008. [cited at p. 97]

[LB05] Annabella Loconsole and Jurgen Borstler. An industrial case study on requirements volatility measures. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 249–256, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 97]

[LE02] David B. Lowe and John Eklund. Client needs and the design process in web projects. 2002. [cited at p. 7, 8]

[LHs08] David Lowe and Brian Henderson-sellers. Characterising web systems: Characterising web systems: Merging information and functional architectures. In *S. Nansi (Ed.), Architectural Issues of Web-Enabled Electronic Business, Idea Group Publishing.*, 2008. [cited at p. 7, 8]

[MAW06] A. Moreira, J. Araujo, and J. Whittle. Modelling volatile concerns as aspects. In *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE 2006)*, pages 544–558, Luxemburg, June 2006. [cited at p. 97]

[NZF04] N.Ñurmuliani, Didar Zowghi, and Sue Fowell. Analysis of requirements volatility during software development life cycle. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference*, page 28, Washington, DC, USA, 2004. IEEE Computer Society. [cited at p. 97]

[Ree79] Trygve M. H. Reenskaug. Models - views - controllers. Web Article, 1979. Available online http://heim.ifi.uio.no/trygver/1979/mvc- 2/1979-12- MVC.pdf. [cited at p. 41]

[Ree03] Trygve M. H. Reenskaug. The model-view-controller (mvc) – its past and present. Presentation, University of Oslo, Oslo, Norway, August 2003. http://folk.uio.no/trygver/2003/javazone-jaoo. [cited at p. 41]

[RGDU08] Gustavo Rossi, Jeronimo Ginzburg, Damiano Distante, and Matias Urbieta. Modeling and deploying volatile functionalities in web applications. 2008. [cited at p. 14, 16, 17, 18, 19, 21, 22, 36, 37, 38, 39, 40, 42, 43, 45, 93, 113, 118, 122, 134]

[RNM+06] Gustavo Rossi, Andres Nieto, Luciano Mengoni, Nahuel Lofeudo, Liliana Ñuno Silva, and Damiano Distante. Model-based design of volatile functionality in web applications. In *LA-WEB '06: Proceedings of the Fourth Latin American Web Congress*, pages 179–188, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 7, 8, 11, 14, 17, 22, 36, 37, 38, 39, 40, 42, 43, 45, 50, 53, 93, 110, 112, 113, 114]

[RNMS06] Gustavo Rossi, Andres Nieto, Luciano Mengoni, and Liliana Silva. Designing volatile functionality in e-commerce web applications. *E-Commerce and Web Technologies*, pages 92–101, 2006. [cited at p. 108]

[Ros96] Gustavo Rossi. *An Object-Oriented Method for Designing Hypermedia Applications.* PhD thesis, Departamento de Informática, PUC-Rio, Rio, Brazil, July 1996. In Portugese. [cited at p. 15, 37, 108]

[SA08] Victor T. Sarinho and Antonio L. Apolinario. A feature model proposal for computer games design. In *Proceedings of SBGames 2008: Computing Track*, Belo Horizonte, Brazil, November 2008. [cited at p. 14]

[SR98] Daniel Schwabe and Gustavo Rossi. An object oriented approach to web-based applications design. *TAPOS*, 4(4):207–225, 1998. [cited at p. 15, 112]

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005. [cited at p. 14]

[W.D74] Edsger W.Dijkstra. Ewd 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1974. [cited at p. 82]

[Wik09a] Wikipedia. Double or nothing — wikipedia, the free encyclopedia, 2009. [Online; accessed 13-May-2009]. [cited at p. 30]

[Wik09b] Wikipedia. Generala — wikipedia, la enciclopedia libre, 2009. Descargado 1-agosto-2009. [cited at p. 30]

[Wik09c] Wikipedia. Prototype javascript framework — wikipedia, the free encyclopedia, 2009. [Online; accessed 20-July-2009]. [cited at p. 26]

[Wik09d] Wikipedia. Slotmachine — wikipedia, the free encyclopedia, 2009. [Online; accessed 31-July-2009]. [cited at p. 25, 28]

[Wik09e] Wikipedia. Time to market — wikipedia, the free encyclopedia, 2009. Online accessed 16-March-2009. [cited at p. 7, 21]

[Wik09f] Wikipedia. War (card game) — wikipedia, the free encyclopedia, 2009. [Online; accessed 23-July-2009]. [cited at p. 30]

[YJ02] Joseph W. Yoder and Ralph E. Johnson. The adaptive object-model architectural style. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, 2002. Kluwer, B.V. [cited at p. 14]

Appendices

Apéndice A

Apéndice: Method Wrappers

A.1. Introducción

La introducción de funcionalidad volátil como un nuevo concern puede requerir cambios en más de un lugar del programa base. Es probable que estos lugares se encuentren dispersos en el código del programa, generando "tangled code". El código "enredado"(tangled code) es extremadamente dificil de mantener debido a que un pequeño cambio en la funcionalidad requiere que haya que "desenredar"el código para volver a armarlo después[KHH+01]. El caracter temporal de la funcionalidad volátil implica la inminente extracción de dicha funcionalidad del programa. Entonces, el proceso de desenredar y volver a armar el código tendrá que hacerse al menos dos veces y en varios lugares (debido a la dispersión en el código de la solución, scattered code). La posibilidad de incorporar luego más funcionalidad volátil hace que el problema se torne aún mucho más complejo. Entonces, es necesario dar con algúna solución que permita disminuir estos problemas.

Method Wrappers es un mecanismo para introducir nuevo comportamiento que es ejecutado antes y/o después, y quizás en lugar de, un método existente [JBR98]. Es una herramienta que puede ser utilizada para implementar mecanismos de AOP. AspectS para Squeak utiliza method wrappers para implementar los distintos tipos de advices y realizar el proceso de weaving[Hir02]. También son usualmente utilizados para programar herramientas de análisis que necesitan aumentar código de algunos métodos. En este trabajo se utilizó method wrappers para programar cierto comportamiento necesario para instalar funcionalidad volátil, alterando el programa base sin modificar su código original.

A.2. Method wrappers en esta tesis

Para evitar los problemas mencionados anteriormente (tangled and scattered code) es necesario evitar la edición intrusiva del código del programa base [RGDU08]. Sería necesario realizar modificaciones en el programa base en distintos lugares para introducir la nueva funcionalidad. En este trabajo se relevaron dichos lugares y luego se implementaron los cambios necesarios. No se utilizó method wrappers desde el comienzo, sino que, en un principio, se implementaron los cambios de forma directa, codificando de forma tradicional sobre el programa base. Luego de que los cambios efectuados de forma tradicional funcionen, se migró el comportamiento necesario a method wrappers.

La manera de utilizar method wrappers es subclasificar la clase MethodWrapper indicando cuál será el comportamiento antes o después del método a decorar. Dentro del method wrapper se tiene acceso al receptor del mensaje original y a los argumentos del método. Esto posibilita implementar funcionalidad como si fuese un método normal, sin limitaciones. Luego, es necesario instanciar el method wrapper, indicar cuál es su comportamiento y finalmente instalarlo en un método de una clase.

Solo fue necesario crear dos subclases de MethodWrapper para poder migrar el código fuera del programa base. Estas clases fueron instanciadas y tales instancias se utilizaron en métodos de una clase llamada VFInstaller. El objetivo de esta clase, es proveer un protocolo que permita instalar(desinstalar) la funcionalidad volátil. Además, en métodos de instancia de VFInstaller se reunieron todas las instanciaciones de MethodWrapper. De esta forma se logró tener en una sola clase, acceso a todo el código necesario para poder instalar la funcionalidad volatil. Esta aproximación permitió eliminar los problemas asociados a la dispersión de código (scattered and tangled code). Al aglomerar las instanciaciones de los method wrappers en un mismo lugar se obtienen los siguientes beneficios:

1. Se identifica el código que corresponde a la funcionalidad rápidamente. Identificar tal código automáticamente mejora grandemente el mantenimiento y la posibilidad de evolucionar de la aplicación. Permite al desarrollador encontrar los lugares en el código que deben ser cambiados cuando cambia la funcionalidad, y hace que tales cambios consuman menos tiempo y sean menos propensos a errores [BvDTvE04]

2. Agiliza el proceso de debugguing. Esto es consecuencia también del primer item.

3. Facilita el proceso de entendimiento ya que el código es no se encuentra disperso, logrando mayor cohesión.

Las desventajas de este enfoque son las inherentes a utilizar method wrappers, que se describiran abajo en un análisis más detallado del mecanismo en general.

A.3. Utilizando method wrappers para solucionar problemas de compatibilidad.

Los method wrappers se utilizaron para implementar comportamiento necesario sin alterar el código de la aplicación core. También pueden utilizarse para hacer compatibles componentes de aplicaciones base con componentes volátiles. El sistema de eventos utilizado para comunicar las partes core con volatile requiere que cuando ocurre un evento se envíe la notificación pertinente. En [BVD07] se indica que esto puede ser implementado mediante AOP en un after advice. En el sistema implementado se puede serguir la misma idea y utilizar method wrappers para efectuar el anuncio del evento. Supongamos que necesitamos acoplar un nuevo side game a un juego, que requiera que cierto evento se anuncie se para poder funcionar (evento PrizePaid). PrizePaid es un evento que indicará que se pagaron los créditos. Supongamos también, que el juego no anuncia dicho evento. Al ser el juego host un juego de slots, en algún momento efectivamente se pagarán los premios otorgados, supongamos que esto ocurre en un método llamado #afterPlayPay. Entonces, para poder acoplar el nuevo side game podemos seguir distintas alternativas:

1. Incluir el anuncio del PrizePaid en el método #afterPlayPay de forma intrusiva. Con esta alternativa clásica, se altera el código de la aplicación principal para incluir funcionalidad volátil.

2. Incluir el anuncio en un method wrapper del método #afterPlayDoSomethhing. El nuevo side game se puede acoplar utilizando el código original de la parte core, por lo que al momento de tener que desacoplar el side game, solo será necesario desinstalar el method wrapper, sin modificar código.

El segundo item cumple con el objetivo de este trabajo, integrar nueva funcionalidad temporal de manera transparente, minimizando los impactos en la aplicación original. Además, obtenemos la posibilidad de instalar y desinstalar la nueva funcionalidad de forma dinámica, en tiempo de ejecución, utilizando herramientas programadas para tal caso, que pueden ser utilizadas por un usuario del sistema.

A.4. Subclases de method wrappers implementadas.

Fue necesario implementar dos subclases de MethodWrapper para alterar el código de la parte core. Esto se debe a que el comportamiento original que provee MethodWrapper no fue suficiente. Recordemos que el comportamiento de MethodWrapper es ejecutar código antes o después de la ejecución del método decorado. Se puede optar por ejecutar código solamente antes o solamente des-

pués. Con la explicación de cada subclase se verá porqué fue necesaria la creación de tal subclase.

A.4.1. ReturnBlockMethodWrapper

En dos ocasiones, que se detallaran adelante, fue necesario ejecutar código después del método original. En tales ocasiones se necesitó utilizar el resultado del método original para finalmente devolver otro resultado. El MethodWrapper permite la ejecución de código posterior, pero no permite utilizar el resultado del método original como parámetro, ni devolver otro resultado. Es por esto que fue necesario crear esta subclase. La clase ReturnBlockMethodWrapper está encargada de ejecutar código que tome como parámetro el resultado del método original, para luego retornar el resultado final. Se instancia utilizando un BlockClosure como parámetro de su constructor. Tal instancia de BlockClosure debe tener solo un argumento. El método original es invocado desde el method wrapper cuando éste se activa y el resultado se almacena en una variable temporal. Luego, se invoca el bloque con el que fue configurado el method wrapper, con el resultado del método original como parámetro. Así, el method wrapper ejecuta el bloque con el que fue creado y devuelve el resultado de dicha ejecución.

A.4.2. MethodInterceptor

En una tercera ocasión fue necesario interceptar el método original para reemplazar completamente su comportamiento. El MethodWrapper no provee funcionalidad para realizar esto, por lo que fue necesario crear la subclase MethodInterceptor. El MethodInterceptor recibe un bloque de 2 argumentos en su constructor. Dichos argumentos contendrán al receptor del mensaje y a la colección de argumentos. Cuando el MethodInterceptor se activa, se ejecuta el bloque con el receptor y argumentos del método original como parámetros y se devuelve el resultado. Notar que no se ejecuta nunca el método original.

A.5. Instanciación de method wrappers

A continuación se explican los lugares donde se utilizaron method wrappers. El objetivo es entender mejor cómo funciona la instalación de funcionalidad volátil, sin alteración de código, en la aplicación desarrollada.

A.5.1. Comportamiento original del programa:

El programa original posee tres métodos que son clave para que el jugador pueda elegir el juego que usará. El primer método es #allGamesClassesFor: de la clase OCCasino. OCCasino representa al casino y una de sus responsabilidades es

conocer cuáles son los juegos disponibles para cada jugador. Dicha responsabilidad está implementada en el método #allGamesClassesFor:, donde se devuelven las clases de los juegos permitidos para el jugador.

OCGamePicker es un componente Seaside que está encargado de renderizar la lista de juegos del casino y proveer al usuario un mecanismo de elección. OCGamePicker se comunica con el casino pidiendo la lista de juegos a través del método mencionado #allGamesClassesFor:. Una vez renderizada la lista de juegos, cuando el usuario final del sistema elige uno, se ejecuta el método #pickGameOfClass: que recibe la clase del juego elegido y genera una instancia del mismo.

A partir de la instanca del juego elegido se generará un componente Seaside listo para renderizarse. Dicho componente es generado en el método #componentFor: de la clase OCComponentFactory y recibe como parámetro la instancia del juego mencionada antes. Este es el paso final para poder renderizar el juego. Luego de la ejecución de este método el usuario puede utilizar el juego elegido.

El código original de los métodos mencionados es el siguiente:

```
OCCasino>>allGamesClassesFor: aUser
    ^OrderedCollection with: Slots.SlotMachine

OCGamePicker>>pickGameOfClass: aGameClass
    | game |
    game := self session casino
                getAGameOfClass: aGameClass
                for: self session user.
    self session announce: (OCGameWasPicked game: game)

OCGameComponentFactory>>componentFor: aGame
    ^Slots.SLSlotmachineComponent new
```

A.5.2. Alteraciones necesarias

El objetivo de los method wrappers descritos a continuación es alterar los métodos anteriores para que en lugar de renderizarse el unicamente el juego, se muestren también los side games activados. Para esto, #allGamesClassesFor: ya no debe devolver una lista de clases sino una lista de paquetes de clases. Estos paquetes de clases, instancias de GamePackageDescription, contendrán la clase del juego junto con las clases de los side games acoplados. Luego, #pickGameOfClass: tendrá que utilizar uno de los paquetes mencionados (y no una clase de un juego) para crear una instancia de GamePackage (y no una instancia de un juego). GamePackage contendrá una instancia del juego elegido y una instancia de cada side game acoplado al juego elegido. Finalmente el método #componentFor: de OCComponentFactory deberá generar el componente correspondiente al juego

original y decorarlo utilizando los side games. Sobre el proceso de decoración nos detendremos más adelante.

El código original de los métodos se mantiene intacto. A continuación se incluye el código de los method wrappers necesarios.

```
1. ReturnBlockMethodWrapper
            on: \#allGamesClassesFor:
            inClass: OnlineCasino.OCCasino
            block:
                [:classes |
                classes inject: OrderedCollection new
                    into:
                        [:allPackages :each |
                        allPackages addAll: (SG.PackageDescriptionsContainer
                                                allPackagesFor: each).
                        allPackages]]
```

Este es el primer method wrapper necesario. La instalación del este wrapper, sobre el método #allGameClassesFor: tiene como objetivo que la lista devuelta no se una lista de clases de juego, sino una lista de paquetes de clases que incluyan la clase del juego y las clases de todos los side games asociados al juego.

```
2. MethodInterceptor
    on: \#pickGameOfClass:
    inClass: OnlineCasino.OCGamePicker
    block:
                [:receiver :arguments |
                | aGamePackageDescription game gamePackage |
                aGamePackageDescription := arguments first.
                game := receiver session casino
                        getAGameOfClass: aGamePackageDescription gameClass
                        for: receiver session user.
                gamePackage := aGamePackageDescription
                        createGamePackageFrom: game.
                receiver session
                        announce: (OnlineCasino.OCGameWasPicked game:
                                                gamePackage)]
```

Una vez instalado el primer wrapper, la lista que devuelve el método #allGameClassesFor: ya no está comprendida por clases de juegos, sino por instancias de GamePackageDescription. Por lo tanto, #pickGameOfClass:, que antes era invocado con una clase de juego como argumento, ahora es invocado con una

instancia de GamePackageDescription como parámetro. Por esto resulta necesario alterar el método #pickGameOfClass: para que trabaje con la instancia mencionada en lugar de una clase de juego. Se instaló un segundo wrapper en el método #pickGameOfClass de la clase OCGamePicker. Se utilizó un method wrapper denominado MethodInterceptor, que ejecuta un código alternativo en lugar del método decorado. Mediante el uso de este wrapper se consiguió cambiar el comportamiento de #pickGameOfClass: para que se tome una instancia de GamePackageDescription en lugar de una clase de juego.

```
3. ReturnBlockMethodWrapper
    on: \#componentFor:
    inClass: OnlineCasino.OCGameComponentFactory class
    block:
        [:component :receiver :arguments |
        | aGamePackage sideGameComponent |
        aGamePackage := arguments first.
        aGamePackage sideGames do:
                [:each |
                sideGameComponent :=
                each componentFactory createComponentUsing: component
                                                    sideGame: each.
                component addDecoration: (SG.SGSlotDecoration from:
                                                     sideGameComponent)].
        component]
```

Finalmente, es necesario generar el componente del juego y decorarlo con los side games. Para esto se utilizó un ReturnBlockMethodWrapper. Primeramente el method wrapper invoca al método original, por lo que se obtiene el componente correspondiente al juego elegido. Con dicho componente se puede proceder a generar las decoraciones necesarias. Sobre la decoración de componentes de juegos con componentes de side games se explicará más adelante.

A.6. Análisis de method wrappers.

A continuación se evaluaran las ventajas y desventajas de la utilización de esta técnica.

Los method wrappers pueden son beneficiosos porque:

1. Permiten mantener limpio el código de la aplicación base ya que el código externo se mantiene fuera.

2. Se puede instalar y desinstalar la funcionalidad volátil en runtime sin cambios en el código base.

3. Como vimos anteriormente, permite aglomerar los cambios en un lugar común, disminuyendo los problemas que tener código disperso pueden traer, a saber: posibles efectos secundarios, dificultad en debugging, dificultad de entendimiento, dificultad de mantenimiento.

No obstante lo anterior, es necesario tener cuidado al utilizarlos. El mal uso de method wrappers puede llevar a obtener programas que sean complejos y difíciles de entender[JBR98]. El código de los métodos ordinarios sigue un flujo imperativo, que posee cierta localidad y por lo tanto puede ser comprendido con mayor o menor esfuerzo. Al instalar un method wrapper sobre un método, las acciones de tal método ya no pueden ser comprendidas simplemente leyendo su código y el de sus colaboradores. Se pierde la secuencialidad, localidad y estilo unitario[FF00]. Es necesario entonces saber cuáles son los wrappers activos para poder comprender el comportamiento del método decorado.

El código que instancia un method wrapper está escrito en algún lugar, entonces, es posible saber a qué método está decorando. Para saber cuál es el comportamiento de un método es necesario saber si existen method wrappers instalados sobre dicho método y solo se puede saber con certeza en tiempo de ejecución. Por lo tanto, el comportamiento de un método puede variar a través de ejecuciones dependiendo de la instalación del method wrapper. Con todo esto, aún es posible seguir el flujo de programa, pero claramente, es más dificil. La situación se puede tornar aún más difícil si múltiples method wrappers interactúan a la vez. El flujo de programa puede tornarse imposible de seguir. En cierta forma, los method wrappers tienen la misma desventaja con respecto a la capacidad de entendimiento del programa que el mecanísmo de eventos implementado en este trabajo. Como se vio anteriormente, en el sistema de eventos utilizado, las reglas y sus acciones asociadas pueden hacer que el seguimiento y comprensión del flujo de programa sea muy dificil.

Los method wrappers modifican el comportamiento en tiempo de ejecución. Para saber si un método está siendo modificado es necesario conocer las instancias de MethodWrapper vivas en la imagen de smalltalk que están instaladas sobre tal método. En [Hir01] se indica que sería útil una herramienta de extensión del browser de smalltalk que permita acceder a los métodos alterados por un method wrapper, como también acceder a todos los method wrappers que alteran un método particular, o todos los method wrappers instalados en un sistema. Esto sería útil porque al permitir saber rápidamente cuáles son los method wrappers activos se facilitaría la comprensión y disminuirían los problemas asociados.

A continuación se delinean las posibles características de una herramienta como la mencionada en el párrafo anterior: Para saber qué method wrappers están instalados sobre una clase es necesario almacenar las instancias de estos method wrappers en algún lugar global, accesible en tiempo de desarrollo. Con estas instancias, se puede invertir la situación, desinstalar (instalar) el method

wrapper, y también se puede saber qué clases están siendo alteradas. Las instancias de method wrappers pueden ser accedidas fácilmente utilizando mecanismos básicos. Se podría desarrollar una extensión al browser de smalltalk que permita saber cuáles son los method wrappers instalados sobre un método. La extensión podría permitir instalar, desintalar e inclusive, eliminar las instancias de method wrappers deseadas.

Method Wrappers es bastante peligroso. Por ejemplo, si dentro del method wrapper se invoca al método decorado, ocurre una recursión infinita. La instalación y desinstalación de method wrappers es algo muy delicado. Las instancias de method wrappers pueden mantenerse vivas si no son tratadas con cuidado al momento de instalarlas y desinstalarlas. Cuando quedan instaladas sin que éste sea el cometido, hay efectos secundarios que son imposibles de debuggear. Aunque este problema se puede salvar usando el método de emergencia de clase en MethodWrapper #uninstallAllWrappers, es algo que tiene que ser considerado con mucho cuidado.

Method Wrappers no se pueden debuggear si no se incluyen breakpoints. Si se aumenta el comportamiento de un método, #metodoX, mediante un method wrapper, el debugger no pasa por el código del method wrapper a no ser que se haya incluido un breakpoint en el mismo. En el proceso de debugging, se invocará el método original y este se ejecutará y retornará un resultado sin que el debugger entre en el código del method wrapper. Así el método original se comportará de manera extraña y si el programador no es conciente de que hay method wrappers activos, los resultados pueden ser muy frustrantes. Nuevamente, para poder realizar debugging en method wrappers es necesario incluir breakpoints para que el debugger pueda pasar por el código en cuestión.

Con lo recién mencionado, dificultades de debugging, comprensión de código y posibles efectos secundarios, la tarea de mantenimiento se dificulta considerablemente al incorporar method wrappers. Paradojicamente la utilización method wrappers, como dijimos anteriormente, permite mejorar el proceso de mantenimiento al aglomerar los cambios en un mismo lugar. Entonces, la mantenibilidad del sistema se ve beneficiada por un lado y perjudicada por otro. Finalmente, los method wrappers constituyen un mecanismo poderoso que permite agregar comportamiento dinámicamente sin alterar el código de un programa. Pero al utilizarlos el programador tiene la responsabilidad de mantener el mayor de los cuidados para evitar los aspectos negativos.

Apéndice B

Apéndice: Documentación sobre prototipos desarrollados

B.1. Introducción

En esta sección se presentan los modelos de los prototipos desarrollados y algunas de sus características relevantes. Se incluyen los diseños UML de los prototipos más importantes, para brindar un panorama de la complejidad que cada uno trajo aparejada. Una explicación detallada de las decisiones de diseño y del funcionamiento de cada aplicació desarrollada escapa al alcance de esta tesis. Es por esto que esta información está condensada en este apéndice.

B.2. Aplicación Casino

La aplicación casino consiste en un sitio web que permite al jugador registrar una cuenta e ingresar al casino virtual con la misma. Mantiene asociada a cada jugador una cuenta donde se almacenan créditos que pueden ser utilizados para jugar en los distintos paquetes de juego-side games ofrecidos. En la figura B.1 se puede ver el diagrama UML de clases y paquétes de la aplicación Casino.

En el casino online se registran las slotmachines utilizadas por los usuarios. De esta forma se inicia el proceso de seguimiento de eventos mediante persistencia. cada actividad realizada por los juegos es registrada en el casino, que actúa como Façade frente a los juegos y provee un protocolo para poder realizar el registro de los distintos eventos.

Como se puede observar en el modelo de la aplicación casino, se modelaron distintos roles que determinan conjuntos de acciones posibles dependiendo de los sectores de la aplicación. La jerarquía Employee contiene los roles principales que

se desarrollan en el casino virtual. Cada Employee juega un papel importante en el funcionamiento de la aplicación como un todo.

El rol Engineer define las tareas a realizar para la creación, mantenimiento y recuperación de Slotmachines. De esta forma se controla el proceso de creación de juegos y recuperación frente a caidas. El Engineer interactúa con la base de datos para poder reconstruir estados de slotmachines según corresponda.

El Accountant es el encargado de crear y mantener las cuentas de créditos de los usuarios del casino. Cuando un usuario es registrado en el casino, el Accountant inicia una cuenta que será mantenida y sobre la cuál se debitarán y depositarán créditos. Sobre la cuenta del jugador se realizarán las transacciones correspondientes al cash in y cash out de cada juego. Cuando un juego necesita dinero para ejecutarse, el jugador debe realizar la operación de cash in. Con esta transacción, créditos de la cuenta manejada por el Accountant se transfieren hacia la Slotmachine solicitante. De esta manera, el Accountant maneja las cuentas de los jugadores en el casino, mientras que cada juego posee una cuenta particular.

El rol UserManager auna las tareas de registro y adminsitración de usuarios. Cuando un usuario del casino crea una cuenta en el sitio, el UserManager es el encargado de realizar dicha tarea. Lo mismo ocurre cuando el usuario ingresa al casino o se va del mismo (login y logout). Estas tareas traen como consecuencia el registro de las actividades en la base de datos. Todo esto es realizado por el UserManager.

La clase GlorpDescriptorSystem contiene la definición de los mapeos Objeto-Relacional utilizada por Glorp. En esta clase se define cómo será traducido desde el mundo de Objetos al mundo de Base de datos relacionales, cada propiedad, cada clase y cada relación.

B.3. Juego Slotmachine

La aplicación principal slotmachine posee todas las características de un juego de slots completo. Se puede ver en su modelo UML en la figura B.2 cada componente de una Slotmachine tradicional.

La aplicación consiste en un juego de slots que simula el juego de dados generala. El jugador puede elegir arrojar todos sus dados mediante la acción spin. El spin consiste en girar todos los reels de la slotmachine y simula la tirada de los cinco dados. Mediante el feature respin se simula la elección de dados para realizar las tiradas selectivas. Respin consiste en hacer girar solo algunos reels de la máquina. Por lo tanto, como parte del juego generala, se puede hacer respin de solo algunos reels simulando la tirada selectiva luego de una tirada normal.

El juego otorga premios al jugador de acuerdo a la secuencia de símbolos obtenida y a las líneas elegidas para realizar la apuesta. Las líneas son configurables mediante los botones destinados a tal efecto. Los premios son presentados al

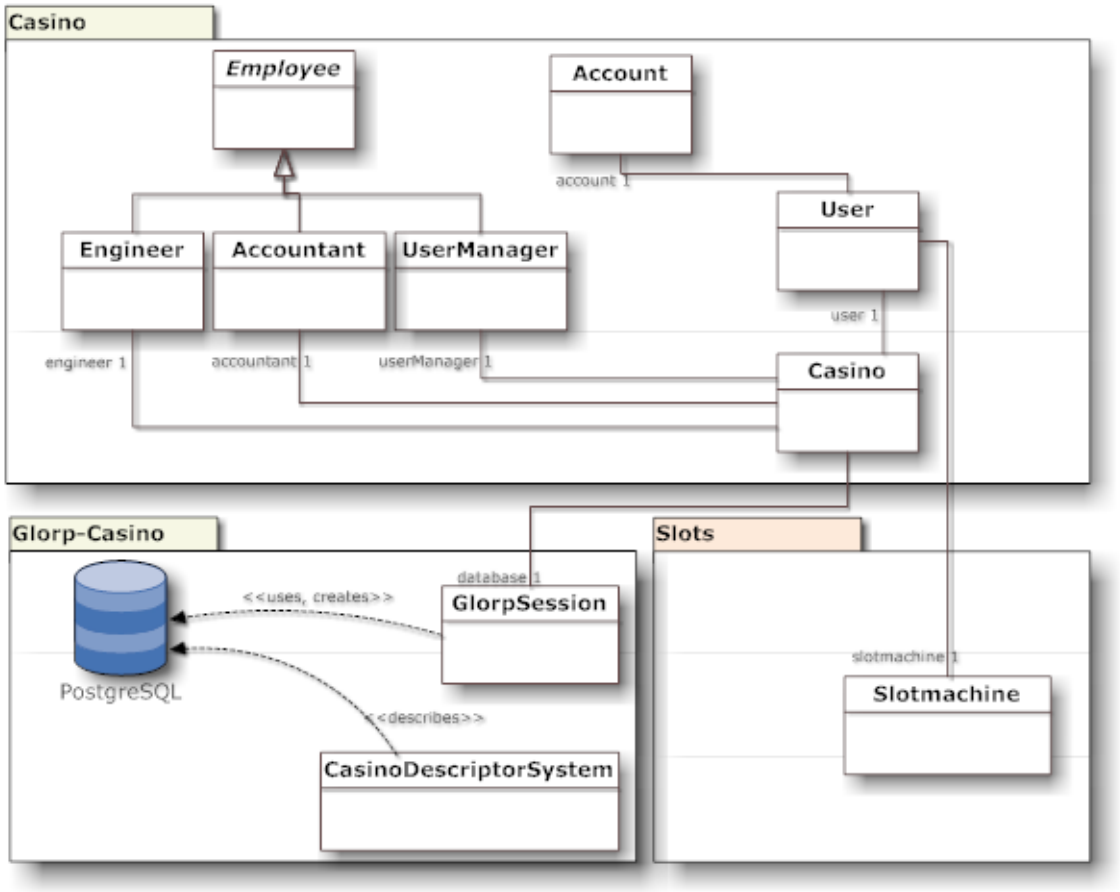

Figura B.1: Diagrama UML de Clases de la aplicación Casino.

jugador cuando la animación correspondiente a la acción de jugada termina. El juego permite la transferencia de créditos desde y hacia la cuenta del casino que posee el jugador. Los créditos son necesarios para la ejecución del juego.

B.4. Side Games Framework

Se presenta el diagrama UML de los distintos paquetes que componen el pequeño framework de side games en la figura B.3.

El mismo posee mucha funcionalidad implementada en la parte de controller y vista. Mediante dicha funcionalidad se implementan los mecanismos comunes a los side games implementados. El framework provee mecanismos para crear componentes visuales de Seaside que renderizan los contenidos de los side games. La clase SideGameComponent es la clase padre de la taxonomía dedicada a tal efecto.

Los SideGameComponents son creados por un Factory[GHJV94], denominado

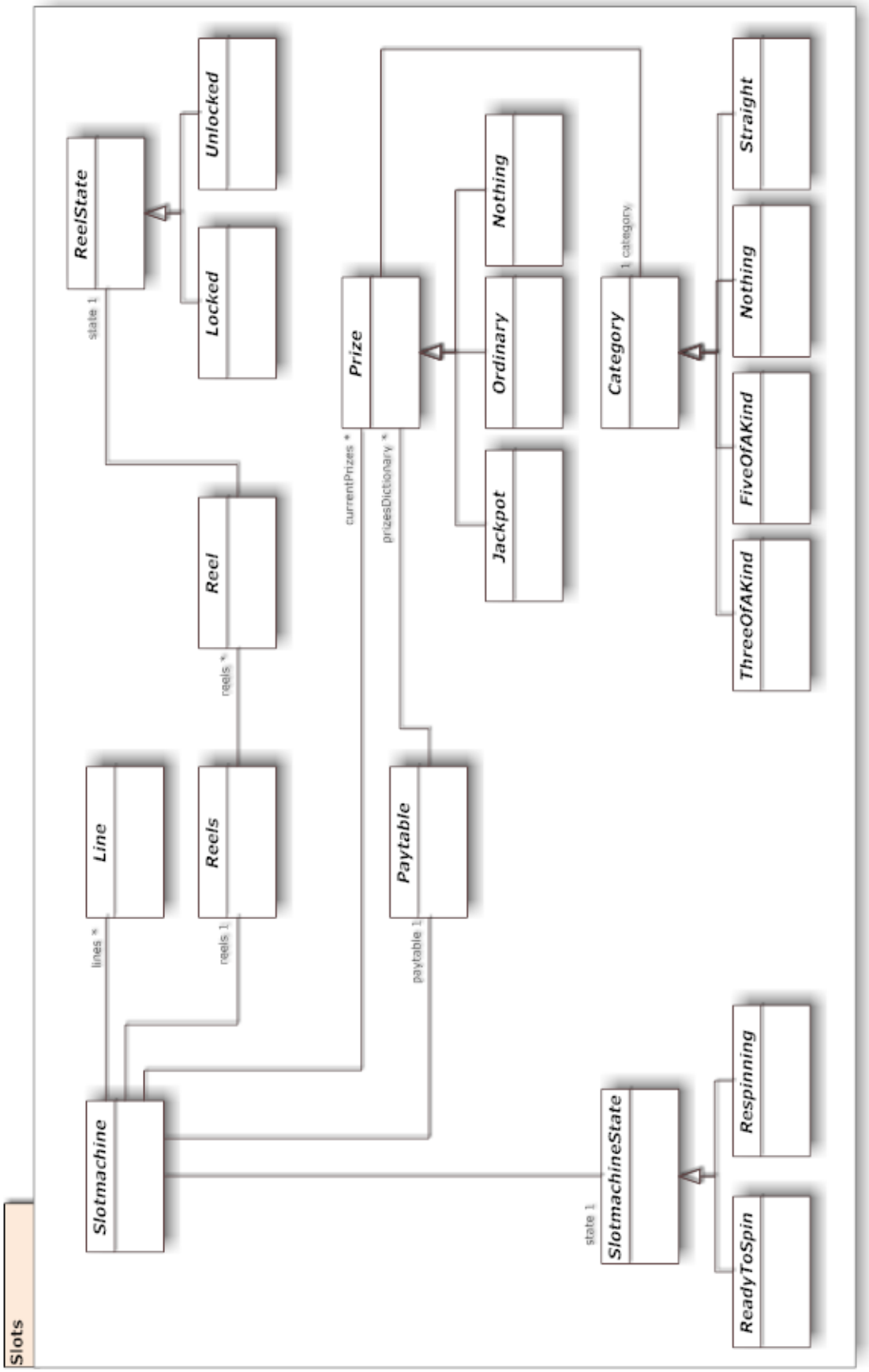

Figura B.2: Diagrama UML de Clases de la aplicación Slotmachine.

SideGameComponentFactory, de forma automática como parte del proceso de instalación de la funcionalidad volátil. Cada subclase de SideGame especifica cuál es el factory utilizado para determinar el proceso de creación de su componente visual asociado. En el SideGameComponentFactory concreto, especificado para cada side game se define el proceso de creación de los componentes visuales del side game. En dicho proceso de creación se especifica cuáles serán los updaters a utilizar para los distintos componentes del side game, ver la sección 5.6.2 para más información.

B.5. Aplicación Packager

La aplicación Packager permite la instalación y desinstalación de la funcionalidad volátil. Consiste en una herramienta que permite al desarrollador elegir la aplicación core a configurar. En el caso de los prototipos desarrollados solo se implementó un juego principal como aplicación core. Una vez elegida la aplicación a configurar, la herramienta busca en la imagen por funcionalidades volátiles disponibles para integrar. Se presenta al usuario una lista con todos los side games disponibles para realizar distintas configuraciones con el juego principal. El usuario puede armar paquetes que incluyen al juego principal solo, con uno o más side games. Una vez configurados los paquetes deseados se puede instalar toda la funcionalidad volátil o removerla del sistema.

Los cambios efectuados mediante la utilización de esta aplicación se ven reflejados en la aplicación casino, que provee la lista de paquetes disponibles a los jugadores. Cuando la funcionalidad volátil es removida, los jugadores solo tienen disponible el juego principal. Mediante la creación de paquetes y la instalación de los mismos, los jugadores pueden elegir combinaciones de juego principal con side games.

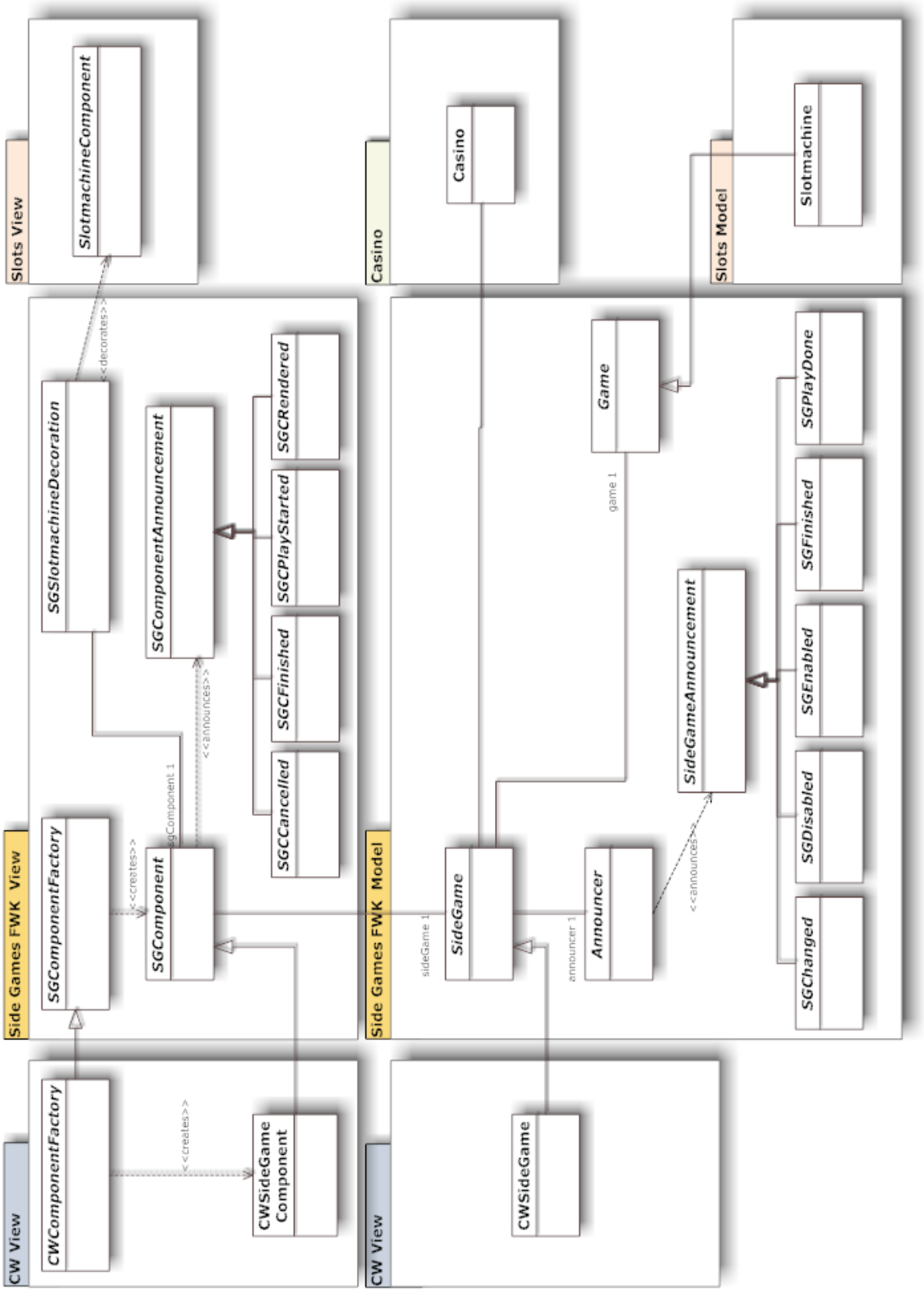

Figura B.3: Diagrama UML del framework de Side Games

Índice de figuras

Índice de tablas

Índice alfabético