UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Un framework para el desarrollo de aplicaciones complejas basadas en Web
Autores: Alejandro Siri y Mariano Montone
Director: Alicia Diaz
Codirector: Alejandro Fernandez
Carrera: Licenciatura en Informática

Resumen

La introducción de la Web como plataforma para el desarrollo de aplicaciones complejas rompió con los moldes y prácticas existentes e introdujo nuevos problemas y complicaciones. Existen muchos Frameworks y Librerías que abarcan estos problemas desde diversos ángulos, y proveen soluciones para algunos ellos. El desarrollador se ve obligado entonces a elegir un conjunto de estos Frameworks y Librerías, intentar solucionar la mayor cantidad de problemas posibles, e integrarlos para que funcionen en conjunto.
Esta tesis describe tales problemas, separándolos en las 3 partes de la arquitectura MVC y luego, basándose en dos conceptos principales, que son la Composicionalidad y la Programación Declarativa, define un Framework que provee una solución a todos estos problemas al mismo tiempo, de manera integrada.
El Framework es luego implementado en PHP, junto con las múltiples librerías que utiliza, y utilizado para resolver un problema del mundo real.

Palabras Claves

Framework, Desarrollo, Web, Ajax, Comet, Componentes, Templates, MVC, MVP, Composicionalidad, Programación Declarativa

Conclusiones

La programación declarativa y la composición fueron ejes fundamentales en el diseño de un Framework integrado y completo.

El Framework diseñado e implementado es usable y performante.

Los problemas del desarrollo web pueden ser solucionados y abstraídos, permitiendo al desarollador enfocarse en el problema real a resolver.

Trabajos Realizados

Investigación de los problemas recurrentes de las aplicaciones web.
Investigación del estado del arte.
Diseño de un Framework para resolver estos problemas.
Implementación del Framework y de librerías complementarias.
Implementación de una aplicación basada en el Framework para una tercera parte.

Trabajos Futuros

Algunas de las mejoras que se pueden hacer sobre el framework son:

- *Agregar "Value Expressions"*
- *Realizar un Testing Automático completo*
- *Permitir Múltiples Bases de Datos*
- *Implementar Integración con Librerías existentes*
- *Implementar un mecanismo de Continuaciones*
- *Diseñar y Desarrollar Plantillas avanzadas*
- *Migrar el código a PHP 5.3*

Fecha de la presentación: Febrero de 2012

Un framework para el desarrollo de aplicaciones complejas basadas en Web

Alejandro Siri Mariano Montone

Índice general

Índice de figuras

Índice de cuadros

La introducción de la Web como plataforma para el desarrollo de aplicaciones complejas rompió con los moldes y prácticas existentes e introdujo nuevos problemas y complicaciones. Existen muchos Frameworks y Librerías que abarcan estos problemas desde diversos ángulos, y proveen soluciones para algunos ellos. El desarrollador se ve obligado entonces a elegir un conjunto de estos Frameworks y Librerías, intentar solucionar la mayor cantidad de problemas posibles, e integrarlos para que funcionen en conjunto. Esta tesis describe tales problemas, y luego define un Framework que provee una solución a todos estos problemas al mismo tiempo, de manera integrada.

1 Introducción

La introducción de la Web como plataforma para el desarrollo de aplicaciones complejas rompió con los moldes y prácticas existentes e introdujo nuevos problemas y complicaciones. El acceso concurrente por parte de varios usuarios y la interacción de forma distribuida en una arquitectura cliente-servidor, sumados al surgimiento de estándares, requerimientos de accesibilidad y el caudal de información, hacen necesaria nuevas formas de resolver viejos y nuevos problemas.

El desarrollador de aplicaciones Web debe lidiar con problemas de persistencia de datos, de interacción y de diseño de interfaces de usuario, todo esto en el contexto de una aplicación destinada a ser accedida por múltiples usuarios mediante una interfaz Web.

Las problemáticas que surgen son múltiples. En el caso de la persistencia de datos, el desarrollador debe leer y guardar los datos que tienen que ser compartidos entre varios usuarios desde un repositorio durable, y transformar los datos persistidos en información manejable por la aplicación, lidiando con problemas de desajuste por impedancia [15]. Debe utilizar transacciones para conservar la consistencia de los datos al mismo tiempo que su uso impone restricciones en el diseño de la interfaz y tiene consecuencias en la navegabilidad de la aplicación. Además, debe ocuparse de mantener la consistencia de los datos en memoria con la base de datos, considerando errores de interacción con el motor de base de datos y el acceso concurrente por parte de múltiples usuarios.

En cuanto a la interacción, debe resolver problemas de navegabilidad propias de una aplicación Web. Debe solucionar problemas de edición de datos de forma transaccional que suelen repercutir en el tratamiento de objetos del modelo. Además, debe encargarse de la interacción con el usuario y de la actualización de la interfaz, manteniendo un grado de consistencia alto de ésta con el estado de la aplicación.

Por último, debe integrar todo con una presentación que utilice tecnologías como AJAX[1] [37] y DHTML[2] [44], incorporar el trabajo hecho por diseñadores gráficos a la aplicación y hacer uso de las tecnologías Web 2.0[31].

[1] Asynchronous JavaScript And XML
[2] Dynamic HTML

Si bien existen soluciones que tratan estas problemáticas, éstas se basan en la utilización de librerías para resolver los problemas específicos y más simples, pero no proveen una solución integrada y consistente. Por ejemplo, en aplicaciones cuyo modelo es un diseño orientado a objetos, las soluciones existentes presentan mapeos relacional-objeto, o soporte para transacciones no adecuadas para un paradigma de objetos, ni para una aplicación de alta interactividad. Nos encontramos con la misma problemática en los motores de plantillas [36, 32]. Muchos de ellos permiten cambios en el flujo de la aplicación desde esta capa, que corresponde a la vista, lo que no es deseable en la arquitectura MVC[3][16]. En el caso de la comunicación cliente-servidor, existen librerías para simplificar la utilización de AJAX pero que no proveen una solución automática. En cuanto a la programación de la interfaz de usuario, en general no se proveen abstracciones adecuadas para la composición de elementos de interacción con el usuario.

En esta tesis se revisará con más detalle cuáles son esos problemas, se analizarán las soluciones existentes y se propondrán soluciones nuevas o alternativas. Ese trabajo culminará en el desarrollo de un framework para desarrollar este tipo de aplicaciones en un ambiente Web que sintetice de forma armónica las distintas soluciones que se hayan encontrado.

En el capítulo 2, se verá en más detalle cuáles son los problemas básicos de una aplicación web, a partir del ejemplo del *eMarket*. A continuación, en el capítulo 3 se verán algunos enfoques que van a motivar muchas de las decisiones que se tomarán para desarrollar el framework. Además, en el capitulo 4 se hará un breve repaso de cómo esos problemas son tratados por los lenguajes/frameworks mas populares de la actualidad, presentando también una forma de solucionar estos problemas utilizando diferentes técnicas; En el capítulo 5, se analizará en detalle el framework creado y se explicará cómo este soluciona los problemas de una forma integrada y consistente. En el capítulo 6 se detallan algunas de las técnicas utilizadas para resolver los problemas, y mejoras que se debieron introducir para poder implementar cada una de las soluciones del capítulo anterior. Por último, en el capítulo 7 se enuncia un caso del mundo real para el cual este framework fue utilizado.

[3] Model-View-Controller

We can't solve problems by using the same kind of thinking we used when we created them.

Albert Einstein

2

Los problemas recurrentes del desarrollo de Aplicaciones Web

2.1 MVC en la Web

Las aplicaciones Web generalmente presentan una arquitectura Model-View-Controller (MVC) [16]. Bajo esta arquitectura, la aplicación presenta un diseño por capas. Una capa correspondiente a la vista, que presenta la funcionalidad relacionada con las cualidades estéticas de la aplicación. Una capa correspondiente al modelo, que contiene la lógica de la aplicación en cuanto al modelo de negocios. Una capa correspondiente al controlador, encargada de responder a las interacciones que hace el usuario sobre la vista, y manipular el modelo según éstas (ver figura 2.1)

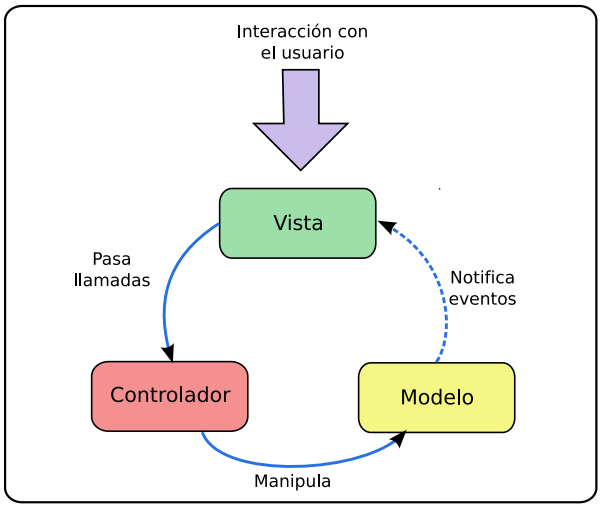

Figura 2.1: Arquitectura Modelo-Vista-Controlador

Esta arquitectura presenta variaciones en el contexto del desarrollo de aplicaciones Web. Esto es así por los requerimientos que la plataforma Web impone. Por el lado del modelo, las bases de datos relacionales se han convertido en el defacto standard a la hora de persistir los datos. Otro tanto ocurre con la Programación Orientada a Objetos; lenguajes y diseños orientados a objetos son utilizados para desarrollar casi cualquier aplicación que corra en la Web. Esto deriva en problemas en cuanto a la persistencia del modelo de aplicación ya que debe hacerse un mapeo entre los objetos del modelo y los datos de la base de datos. A estos problemas se los denomina *problemas de impedancia*. Además, como se verá más adelante, este problema repercute sobre varios aspectos del desarrollo de una aplicación. Por ejemplo, los aspectos de edición de objetos del modelo, el tratamiento de colecciones de datos, la construcción de interfaces transaccionales y demás.

Por el lado del controlador, la aquitectura MVC Web difiere en cuanto a que las aplicaciones no son simplemente accedidas, sino *navegadas*. Esto quiere decir que el usuario espera determinadas cosas respecto de su interacción con la aplicación. Por ejemplo, espera poder utilizar los botones *Atrás* y *Adelante* del navegador Web para poder visitar los sitios previos, tal como si se tratase simplemente de documentos interconectados. Además, puede querer guardar la dirección de un documento (bookmark) para poder acceder al lugar en que se encuentra en este momento, más adelante. Esto supone varios problemas para el desarrollador de aplicaciones Web; éste no debe encargarse solamente de la interacción con el usuario, sino que debe permitir también navegar la aplicación y responder según el usuario espera en los casos que tenga sentido.

En cuanto a la vista de la aplicación, la Web ha dado origen a nuevas (y en algunos casos, mejores) formas de expresión visual. Tanto es así, que en la mayoría de las aplicaciones la parte de la presentación es llevada a cabo por diseñadores gráficos. Nuevamente, esto propone problemas al desarrollador, tanto de interacción con profesionales de diseño como de integración de los trabajos realizados por éstos. En particular, debe integrar interfaces con la forma de documentos XML. Esto ha dado origen a lo que se conoce como *motores de plantillas*, los cuales ayudan a mantener separadas las capas de la aplicación y a integrar los diseños.

Estas variaciones aparecen resumidos en la figura 2.2.

En los siguientes apartados se analizan estos problemas con un poco más de detalle y se ve cómo son resueltos por los frameworks más populares de la actualidad.

2.2 El Ejemplo: eMarket

A lo largo de la tesis, se utilizará un ejemplo muy conocido para la mayoría de los desarrolladores de aplicaciones web: Un sitio de compras online. Se verá que

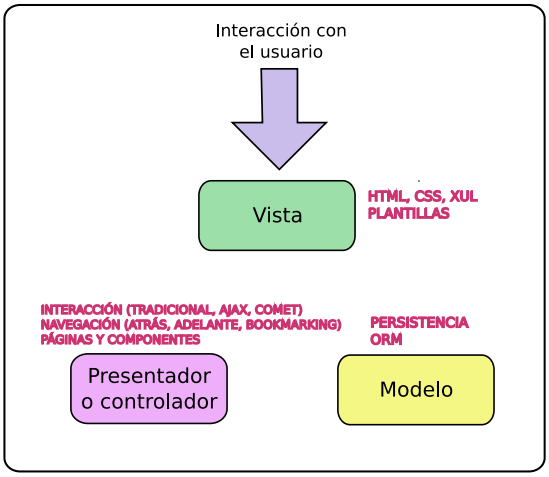

Figura 2.2: El MVC en la Web

en este caso, los problemas que se plantean son relativamente comunes, y pueden suceder en muchas otras aplicaciones.

Si bien no se incluirá una especificación formal completa de la funcionalidad del mismo, se detallan las siguientes características:

- El sitio tiene productos, con distintos parámetros sobre los que se puede buscar (categoría del producto, rango de precios, autor). Cada producto conoce su stock actual.
- Un carrito de compras al que el usuario le puede agregar elementos, para luego comprarlos.
- Los usuarios luego hacen un checkout en un `wizard`, donde ingresan toda la información de pago necesaria. En este checkout se manejan los detalles del pago (método de pago, cantidad de cuotas, etc) y los detalles del envío (dirección, fecha de entrega, si es para regalo o no).
- Un invitado al sitio, es decir, un usuario aún no registrado o logueado, puede cargar cosas a su carrito, y luego, dentro del proceso de checkout, registrarse y completar el proceso, o también puede registrarse luego de haber agregado cosas al carrito, antes de hacer el checkout.
- El carrito de los usuarios registrados se salva entre sesiones. Si el usuario, luego de haber cargado cosas a su carrito, se loguea al mismo desde otra ubicación, o pasada una cantidad de tiempo considerable, seguirá teniendo su carrito en el estado que lo había dejado.

El diagrama UML de la aplicación aparece en la figura 2.3

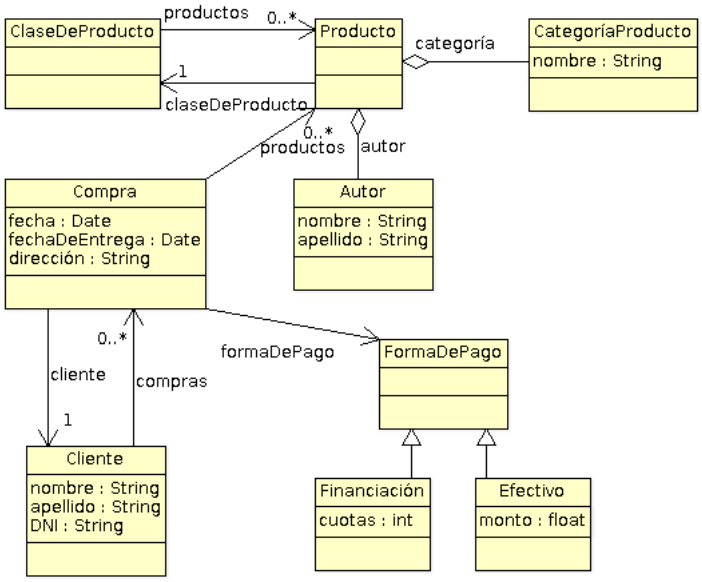

Figura 2.3: Diagrama UML del ejemplo de eMarket

2.3 Los problemas

En esta sección se detallan los problemas mencionados en la introducción de este capítulo, utilizando el ejemplo de *eMarket*como motivación.

2.3.1. Modelo

Mapeo objeto-relacional La mayoría de los sistemas web necesita que los datos utilizados por el mismo perduren en el tiempo, en una base de datos. Luego, estos datos son accedidos y modificados (dentro del modelo) por las distintas funcionalidades del sitio. Como se especifica anteriormente, el modelo es en objetos y la base de datos es relacional. Esto hace necesario una conversión de un esquema al otro, y de los datos en ellos. Las diferencias entre los 2 esquemas son sutiles pero importantes. En el modelo relacional, se pueden especificar relaciones entre tablas, pero es mas complicado expresar la herencia de las clases, obligando a decidir por una estrategia de adaptación del modelo. En el modelo de objetos no existe una manera de definir un atributo de un objeto mediante el cual se lo va a identificar unívocamente en el sistema, impidiendo que exista otro igual. Es muy importante poder mantener una coherencia entre las dos representaciones, y aprovechar al máximo las ventajas de cada una.

Implementación de métodos que modifican el modelo Por lo general existen 2 tipos de métodos en el modelo: Los que realizan consultas sobre el estado, y los que lo modifican. Estos cambios deberían ser durables, es decir, estar vigentes en cada corrida de la aplicación (así por ejemplo se puede hacer un cambio a un domicilio de entrega y el sistema lo tendrá registrado para la próxima vez que se inicie sesión). Entonces por un lado, es necesario que la base de datos sea actualizada con los cambios realizados en el modelo, pero por otro lado, simplifica mucho la labor del programador no tener que intercalar dentro de los métodos del modelo, llamados a la base de datos para que realice la persistencia, sino que estos cambios deberían ser observados por un ente externo y luego llevados a la base.

Consulta del modelo Navegando en el sitio, un usuario puede intentar acceder, entre otras cosas, a información sobre su autor favorito. Sería interesante aquí mostrar información estadística o extraer información de los datos que ya se encuentran almacenados en el modelo, por ejemplo, cuales son los productos más vendidos de este autor. Si se utiliza un modelo relacional, la opción más directa es el uso de SQL para averiguar esta información, pero ya que el modelo es orientado a objetos es interesante poder aprovechar las ventajas que este provee. Las consultas entonces deben de alguna manera traducir lo que el usuario busca a una consulta relacional, para luego recuperar nuevamente objetos del modelo.

Integridad de los datos en memoria Si un feliz comprador en el sitio terminó de elegir cada uno de los ítems que irían en su carrito, y luego de elegir las cantidades de cada uno, y realizar todos los pasos del checkout (que implican modificar datos de dirección de envío, e ingresar datos de la tarjeta de crédito), al realizar el último guardado en la base de datos, ocurre una excepción (por ejemplo, que alguno de los productos no tiene stock suficiente y no se puede cumplir con el requerimiento), es necesario manejar todo cambio en la base de datos mediante una transacción, debido que ante esta excepción es necesario hacer un rollback. Además es importante que el usuario tenga la posibilidad de corregir el error sin perder todas sus modificaciones hasta el momento (por ejemplo, reduciendo la cantidad de ítems elegidos de ese producto).

Modificación de objetos desactualizados Una situación muy habitual en un sistema multi-usuario, es que más de un usuario intente hacer modificaciones en simultaneo sobre un objeto. Esto haría que el objeto pierda integridad. Más aún, es probable que las modificaciones sean sobre un conjunto de objetos, que podrían llegar a ser superpuestos. En este caso, sería interesante que, para evitar inconsistencias, solo uno de los 2 usuarios pueda llegar a hacer sus modificaciones (el primero de los 2 que haya terminado, por ejemplo) y que el segundo, al intentar hacer el guardado, sea advertido de las inconsistencias y sea ofrecido una o más maneras de resolverlas (siempre que sea posible).

Observabilidad de los cambios Uno de los problemas más comunes y al mismo tiempo no conocido y por lo tanto no tenido en cuenta en general, es del de la *observabilidad de los cambios*. Para entender a qué se refiere, es posible considerar el siguiente ejemplo. Se quiere implementar un componente de interfaz para la selección de productos de un autor. Se presentan dos listas, una con los productos ya seleccionados, la otra con los objetos aun no seleccionados. Hay además dos botones que permiten cambiar elementos de una lista y moverlos a la otra, y por último un botón de Cancelar y otro de Aceptar.

Las listas son fácilmente populables inicialmente mediante dos consultas simples a la base de datos, pero luego de cada interacción con el usuario, (luego de cambiar un elemento de un listado al otro) el estado de la base de datos difiere con el de la memoria, por lo tanto una consulta directa a la base de datos no mostrará los listados actualizados, requiriendo un manejo avanzado de la información que se muestra en el listado.

Modificación del esquema del modelo En una aplicación viva, durante su utilización es muy probable que surjan cambios a implementar que impliquen una modificación al modelo. Por ejemplo, luego de unos meses de utilización, se decide que sería útil agregar a los productos la fecha de publicación. Obviamente, los datos ya almacenados no deben perderse, se quiere simplemente agregar un atributo en la clase Producto. También es necesario que las pantallas de administración de la clase Producto tenga el nuevo campo necesario para editar el atributo. Lo ideal es poder modificar la estructura del modelo, y que los datos automáticamente pasen a respetar la nueva estructura, sin pérdida de datos ni trabajo manual de migración.

Administración de objetos Para el sitio web, teniendo múltiples productos a la venta, es necesario poder administrar los productos. Este es uno de los problemas a resolver más comunes y más repetidos en todos los sitios, el de la administración de objetos del modelo. Poder crear, modificar y eliminar objetos, listarlos, buscarlos y filtrarlos (lo llamado ABM - Alta Baja Modificación, o CRUD - Create Read Update Delete) son algunas de las tareas mas comunes y repetitivas en un sitio web. Estos componen una buena parte del comportamiento de los sitios, y si bien a veces los ABM que se pueden generar con las herraminetas de un framework luego deben ser modificados extensamente para poder tener toda la funcionalidad extra que se les quiera dar, son muy útiles para dar una primer funcionalidad básica al sitio.

2.3.2. Controlador

Navegabilidad La Web surge como documentos interconectados mediante enlaces. Más tarde, evolucionó como medio para desarrollar el front end de aplicaciones complejas, que poco tienen que ver con la interconexión de hipertextos.

Así, el desarrollo de aplicaciones bajo la plataforma Web debe tener en cuenta problemas muy particulares relacionados con la navegación mediante páginas, marcadores, etc.

Ahora bien, el modelo navegacional de las aplicaciones Web tradicionales desarrolladas bajo la metáfora de páginas resulta ser limitado en cuanto a la forma de composición de la interfaz. En particular, ésta impone la restricción de que cada posible estado de la aplicación deba poder ser nombrado; esto es, bajo la metáfora de páginas, cada estado distinto de la vista de la aplicación debe llevar aparejada una URL a partir de la cual pueda recuperarse ese mismo estado de la aplicación más tarde, por ejemplo, con propósitos de bookmarking.

Composición de interfaces Existen dos enfoques para la resolución de la navegabilidad: por un lado, el diseño del flujo de la aplicación mediante páginas, y por otro, el flujo por medio de la modificación de un árbol de componentes.

El control de flujo centrado en páginas es el equivalente de una máquina de estados. Según el estado en que se encuentre la aplicación se codifica explícitamente el proximo estado (o página) en que la aplicación debe continuar.

Otro enfoque es construir la aplicación mediante la modificación de un árbol de componentes. De esta manera se eliminan las instrucciones al estilo GOTO del flujo de control y se gana en composicionalidad y claridad en el código. También, al mantener cada componente del árbol su propio estado, e interactuar con el cliente de manera independiente, se mejora de gran manera la posibilidad de reusar componentes. Sin embargo, ésto introduce problemas al momento de acceder a partes de la aplicación a partir de bookmarks; esto es así, porque el estado actual de la aplicación en general ya no es determinado simplemente por aquellos parámetros en el request, sino que se hace de una forma mucho más compleja. De todas formas existen formas de resolver el problema, al menos parcialmente; la modificación de un árbol de componentes como especificación de flujo de control ciertamente representa una ventaja en cuanto al enfoque anterior.

Actualización de vistas El problema de la actualización de las vistas está directamente relacionado con el problema de la programación imperativa. A saber, el desarrollador queda a cargo de orquestar el orden exacto en el cual todos los eventos ocurren, aunque en general le falte la omnisciencia y la clarividencia[1] requerida para hacerlo de forma perfecta. El resultado es tanto complejidad como fragilidad en las soluciones. En consecuencia, una solución habitual es generar la vista completamente ante cada interacción del cliente, lo cual simplifica la tarea de saber qué cosas fueron modificadas, pero en el caso de que el desarrollador quiera una interacción más avanzada que no permita refrescar la página entera, queda en manos de él mismo realizarla.

[1] Capacidad de ver cosas que no pueden ser percibidas por los sentidos normales

Casos de Uso y Transacciones Además de la adminstración de objetos, existen muchos otros lugares dentro de una aplicación donde se realizan ediciones de datos que impactan en el modelo. Estas modificaciones por lo general son transaccionales, es decir, luego de una serie de interacciones con el cliente, deben persistir todas las modificaciones o descartarlas. Durante la duración de esta transacción larga, se necesita que el usuario que se encuentra realizando los cambios los pueda ver, pero que ningún otro usuario los vea. Por ejemplo, si se encuentra dentro del proceso del checkout, es importante que el carrito no pierda los productos que tiene mientras el cliente se encuentra actualizando su dirección de envío, pero el producto no debe decrementar su stock ni la orden de compra ser generada hasta que el cliente no haya terminado de realizar el proceso completo.

2.3.3. Vista

El MVC hace una clara distinción entre aquello debería ser parte de la vista y lo que corresponde al controlador. Como ya se mencionó, el controlador en una aplicación Web está relacionado con la navegabilidad y el control de flujo de la aplicación, además de proveer la funcionalidad necesaria para la actualización de las vistas y la modificación del modelo a partir de los cambios que ésta sufra. Por lo tanto, queda para la capa de la vista lidiar sólo con los problemas de presentación. En particular, en un ambiente Web, esto significa generar el HTML y las hojas de estilo necesarias para dar una buena estética a la aplicación.

Ahora bien, estos problemas de presentación pueden ser divididos en cuatro concerns:

1. Lógica de la aplicación: Todo lo relacionado con el manejo de datos y las interacciones del usuario.
2. Tipo de vista: Si va a ser HTML, XML, XUL, etc.
3. Diseño de la vista: Cómo se va a ver la aplicación, colores, posición de los datos en la pantalla, etc.
4. Forma de interacción: Si van a haber recargas en cada request, si se va a usar AJAX, COMET (ver 6.8).

El tercer ítem de esta lista es un trabajo para el cual el desarrollador no es apto, o al menos no fue capacitado para él; lo más común es dejar la tarea a un diseñador gráfico. De ahí que el problema fundamental no es cómo realizar la presentación de la aplicación, sino cómo integrar el trabajo hecho por un diseñador gráfico con el resto de los concerns.

Generación de vistas La generación de vistas está fuertemente asociada dos aspectos del controlador: el control de flujo y las formas de interacción que éste propone. Según el control de flujo, las formas de generar las vistas varían según este esté orientada a páginas o a componentes. Además, se ven dos tendencias separadas, la generación programativa como las plantillas o "skins" por página. En el caso de la generación programativa, el controlador de la aplicación genera el XML necesario mediante estructuras tradicionales del lenguaje de programación. En el caso de las plantillas, estas son elegidas por un motor en base al estado de la aplicación. Existe también un enfoque híbrido, en el cual las plantillas son elegidas por un motor, pero luego dentro de las mismas se ejecuta código que termina de generar la parte faltante de las mismas, extrayendo información del modelo y del controlador.

Reutilización de vistas De la misma manera que el código de la aplicación puede ser reusable, lo mismo sucede con las vistas, y los patrones de uso suelen ser similares. Por ejemplo en el caso de los formularios, que suelen ser bastante similares en la mayor parte de la aplicación, o en el caso de los listados en los que las posibilidades de reuso es aun mas evidente. La pantalla de listado de productos es diferente de la pantalla del listado de clientes, pero ambas compartirán una forma muy similar, con controles para el filtrado, el ordenamiento, posiblemente una tabulación de los resultados, y los controles para la paginación por ejemplo. Si el esquema de generación de vistas que se utilizan no permite un reuso simple, fuerza a copiar y pegar, trayendo consigo los conocidos problemas de esta técnica. El enfoque de los "generadores" de vista puede parecer muy conveniente al principio, pero fácilmente se puede ver que es una forma eficiente de realizar copy-paste múltiples veces.

Integración con el diseño Luego de tener la aplicación ya desarrollada, implementar el estilo provisto por un diseñador puede llegar a ser un desafío importante. Es importante en este punto que la funcionalidad de la aplicación no se vea afectada por la aplicación del diseño. Cuanto mayor sea la separación de la presentación del funcionamiento del controlador detrás, mas simple será este trabajo. Además, cuanto más se parezca el lenguaje HTML del diseñador al esquema de templates, más simple será la programación luego.

Formas de interacción Otro de los aspectos pertinentes es la forma en que la aplicación interactúa con el exterior, en este caso con el navegador de Internet. Existen varias formas de hacerlo, a saber:

- Una forma *sincrónica*, en la cual cada requerimiento se hace a partir de una acción del usuario. Cada requerimiento supone una generación completa de la página Web.

- Una forma *asincrónica*, la cual cada requerimiento puede no ser disparado necesariamente por una acción del usuario, sino que puede ocurrir como respuesta a algún evento; por ejemplo, a cuestiones que tienen que ver con el tiempo. Además, esta forma de interacción permite evitar una generación completa de la página Web; es posible actualizar sólo aquellas partes de la página que se desee. Esta forma de interacción se denomina AJAX.
- Las dos formas anteriores ocurren como consecuencia de un evento del lado del cliente. Una tercera forma de interacción es una en la cual el disparo de una actualización proviene esta vez del lado del servidor. Esta forma de interacción se conoce como COMET.

Las últimas dos formas agregan cada una más potencia a la aplicación, pero ambas traen aparejadas una complejidad que también va en aumento.

3

Enfoques Metodológicos

A continuación se describen algunos conceptos e ideas utilizados como principios sobre los cuales fue construido el framework.

3.1 Acerca de la composicionalidad

Muchos de los recientes avances en el desarrollo de software tienen que ver con la composicionalidad.

El ejemplo más claro de composición es la composición de funciones matemáticas $f \cdot g$. Esta operación es composicional ya que su aplicación produce una nueva función, la cual puede volverse a componer. Ésto no significa que dos funciones puedan componerse arbitrariamente; es necesario que exista compatibilidad semántica y de tipos entre ellas.

Más específicamente, en el ámbito de la informática:

"Composability means that you can put two bits of code together and important correctness properties will be preserved automatically. This does not mean, of course, that the composition is automatically correct in a wider sense, but it does mean that you don't introduce new bugs merely by sticking two things together.[1]

Un ejemplo menos obvio, aunque no menos importante en el contexto de la informática, es el de recolección de basura[42]. A simple vista podría parecer una forma más conveniente para el manejo de memoria; pero esta perspectiva oculta una característica más determinante: la recolección de basura hace que el manejo de memoria sea composicional.

Consideremos dos módulos X e Y y supongamos la ausencia de un recolector de basura. El módulo X crea un objeto y sirve una referencia a ese objeto al módulo Y. En algún punto en el futuro, X liberará la memoria de ese objeto (manualmente, por supuesto). Sin embargo, *sólo es seguro hacerlo una vez que*

[1] http://paulspontifications.blogspot.com/2007/09/composability-and-productivity.html"

el módulo Y haya terminado de utilizarlo. Entonces, si los módulos X e Y fueron desarrollados de forma independiente, es bastante probable que el módulo Y posea la referencia por más tiempo que lo esperado por el módulo X. Resumiendo, la administración de memoria manual no es composicional ya que la composición de los módulos X e Y puede introducir punteros inválidos que no estaban presentes ni en X en Y por separado.

Si hay un recolector de basura disponible, ésta situación no resulta problemática: el recolector liberará la memoria del objeto sólo cuando *ambos módulos* hayan terminado de utilizarlo. De esta forma, el módulo X puede olvidarse del objeto a su propio tiempo sin necesitar saber nada acerca del módulo Y.

Es importante notar que muchos de los últimos focos de investigación y avances en la disciplina de la informática tienen que ver con hacer el software cada vez más composicional. Quizás el ejemplo más claro sea las recientes investigaciones hechas en el campo de la memoria transaccional[27], cuya motivación subyacente principal es hacer de la programación concurrente una actividad composicional.

Otros ejemplos comunes de esta misma tendencia son por ejemplo las estructuras de datos persistentes [18] (estructuras de datos funcionales). Como la recolección de basura, estas hacen la administración de memoria composicional al no sobreescribir versiones anteriores de la memoria. Esto previene al programador de modificar la memoria incorrectamente, es decir, de forma no composicional; memoria que puede estar siendo accedida desde otras partes del programa. Un control de concurrencia optimístico y multiversión (lock-free) [24] también tiende a la composicionalidad, especialmente al evitar el uso de locks. Incluso el uso tan en boga de continuaciones para el desarrollo de aplicaciones Web[19] son un ejemplo de ésto. Las continuaciones permiten solucionar el problema de navegabilidad de aplicaciones Web composicionalmente.

El framework desarrollado hace aportes al desarrollo composicional en el contexto de desarrollo de aplicaciones Web.

3.2 Programación declarativa

La *programación imperativa* es un paradigma de programación que describe las computaciones en término de sentencias que cambian el estado del programa.

Contrariamente a ésto, un estilo de programación *declarativo* es aquel en el cual se describen los resultados deseados del programa sin enunciar de forma explícita los comandos o pasos necesarios para alcanzarlos[35].

El flujo de datos es una arquitectura según la cual el cambio del valor de una variable fuerza que otras variables dependientes de ésta sean calculadas nuevamente.

El ejemplo más claro de este estilo de programación son las hojas de cálculo. El usuario ingresa una fórmula en una celda. El valor de la fórmula depende del

valor de otras celdas. Cuando el valor de una de estas celdas cambia, el valor de la celda en el cual se ingresó la formula es calculado nuevamente.

La programación por flujo de datos hace implícita una cantidad significativa de computaciones que es necesario expresar explícitamente en otros paradigmas de programación.

Por último, se verá un ejemplo más, esta vez en el contexto de la programación de interfaces gráficas. En un editor de textos, el estado del menu "Cortar" generalmente está activo sólo cuando el usuario ha seleccionado una porción de texto. En una aplicación normal, alcanzar este comportamiento requeriría identificar todos los métodos y acciones del usuario que pueden llegar a modificar la región de texto que esta actualmente seleccionada, y agregar en todos ellos la funcionalidad necesaria para activar y desactivar el menú de forma tal de mantenerlo en un estado consistente (Figura 3.1). Si se dispone en cambio de la posibilidad de programar bajo un paradigma de flujo de datos, sólo es necesario indicar que el estado del menú depende de la longitud de la región de texto seleccionada: si la longitud es 0, entonces el estado del menu es "desactivado", caso contrario es "activado" (Figura 3.2). De ahora en más es posible trabajar sobre el resto de la aplicación ignorando el estado del menu; éste será recalculado automáticamente cada vez que la longitud de la región de texto seleccionada varíe. Además, todo aquello relacionado con el menú es colocado cerca de su definición y no esparcido a través de toda la aplicación.

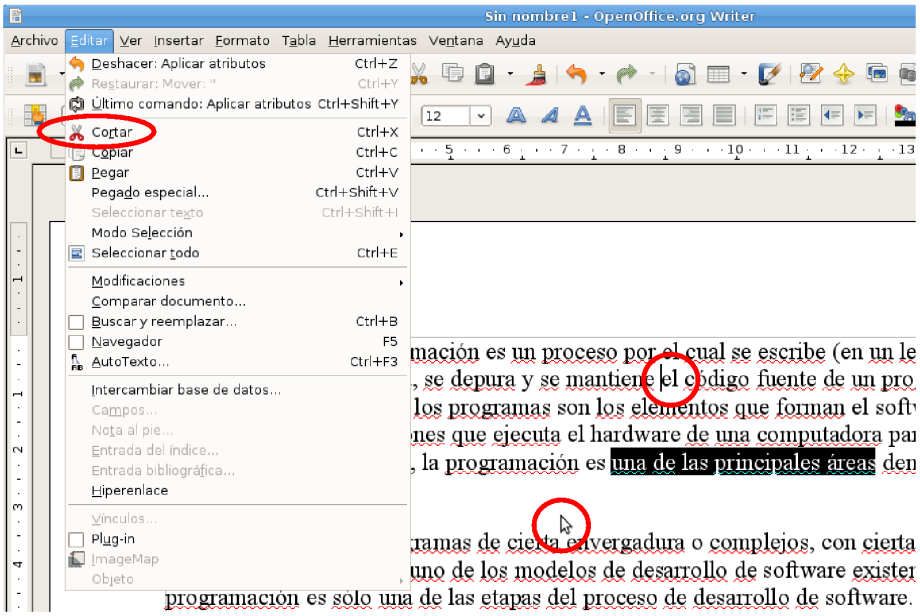

Figura 3.1: Estado del menú "Cortar" bajo el paradigma imperativo

En la próxima sección se verán más en detalle todos estos conceptos.

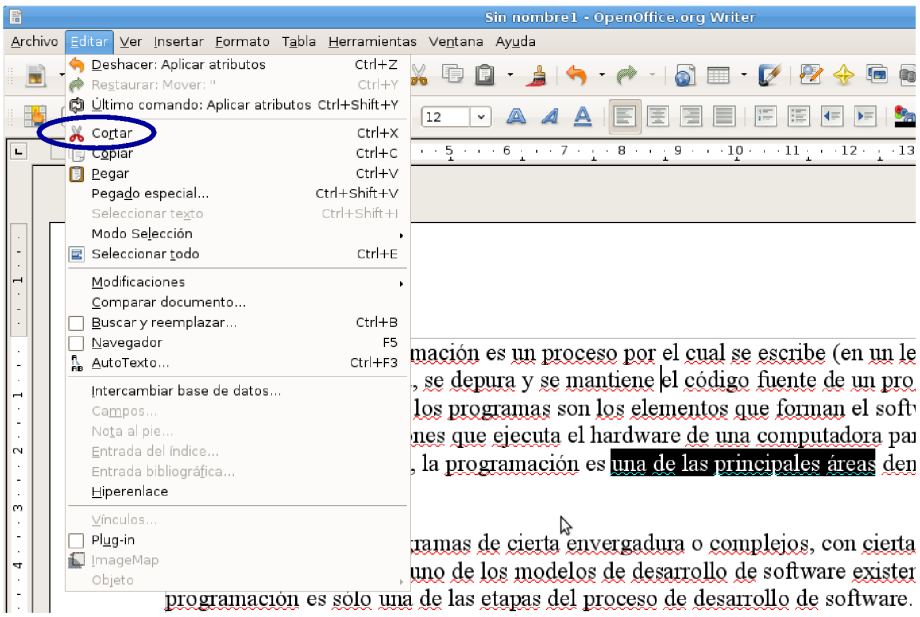

Figura 3.2: Estado del menú “Cortar” bajo el paradigma declarativo

3.2.1. Actualización imperativa y declarativa

En este apartado se hace una comparación de los estilos imperativo y declarativo para la actualización de los estados de una aplicación y se enuncian las ventajas y desventajas en cada caso.

Se toma como ejemplo el grafo de dependencias entre objetos que contienen estados de la figura 3.3. Las flechas caladas indican una dependencia de estados entre los objetos involucrados; quiere decir que cada vez que el estado del objeto hacia el cual apunta la flecha cambia, el estado del otro objeto también lo hace.

En el caso de una actualización de estilo imperativa, el usuario debe desarrollar el código que realice la actualización en cada nodo que cambia de estado. Por ejemplo, en la figura 3.4 se indican con rojo las partes que el usuario debe desarrollar para llevar a cabo la actualización. Las flechas en rojo significan que el usuario es el que decide cuáles son los objetos a actualizar, además el orden, los momentos y el número de veces que se realizan las actualizaciones. Las aureolas coloradas significan que ese código es colocado en cada objeto que provoca un cambio de estado en otros objetos; aquellos objetos que no lo hacen no la poseen, como puede verse en esa misma figura. Las desventajas de esta forma de actualización surge de su propia construcción. Primero, los nodos que se ven afectados son determinados en cada caso por el desarrollador; no se hace una deducción a partir de las dependencias. Como se ve más adelante, esto repercute en la composicionalidad del enfoque. Segundo, el orden, los momentos

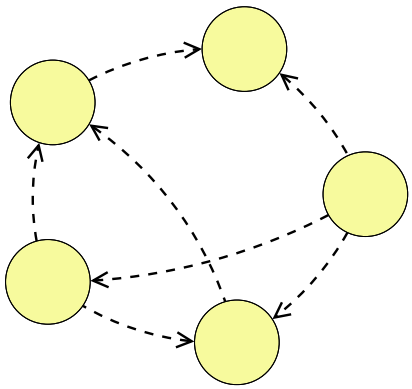

Figura 3.3: Dependencias entre objetos con estados

y el número de veces que se realizan las actualización es indicado manualmente por el desarrollador en cada caso, lo que hace de la actualización de estados un verdadero problema a medida que la aplicación crece y el número de estados a tener en cuenta aumenta.

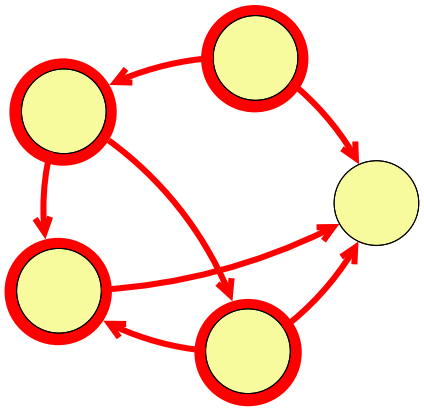

Figura 3.4: Actualización imperativa

Alternativamente, la actualización declarativa de estados desacopla la especificación de dependencias entre los objetos con estados del código que efectivamente realiza las actualizaciones. Esto acarrea una serie de ventajas con respecto a la versión imperativa. Por un lado, el desarrollador sólo es responsable de determinar las dependencias entre los objetos; el orden, los momentos y el número de veces que se realizan las actualizaciones es llevado a cabo por un algoritmo destinado a tal fin. Esto tiene consecuencias en la capacidad del software de conservar la forma ante un incremento del número de estados a tener en cuenta y en la composicionalidad de la solución. En la figura 3.5 puede verse un ejemplo de este enfoque. Una vez más, las aureolas rojas indican las partes que el usuario

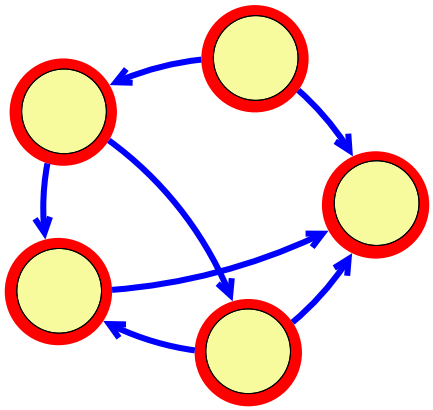

Figura 3.5: Actualización declarativa

debe desarrollar. A diferencia que en el caso anterior, éstas sólo indican que el usuario debe determinar las dependencias entre los distintos objetos. Además, los objetos que serán actualizados y la manera de llevar esto a cabo no es dejado como tarea al desarrollador, sino que se realiza de forma automática. Esto se ve reflejado en el color azul de las flechas que representan la actualización.

La composicionalidad de esta nueva solución no es menos simple de ver. A continuación se considera la inclusión de un nuevo objeto con estado a nuestro sistema (Figura 3.6).

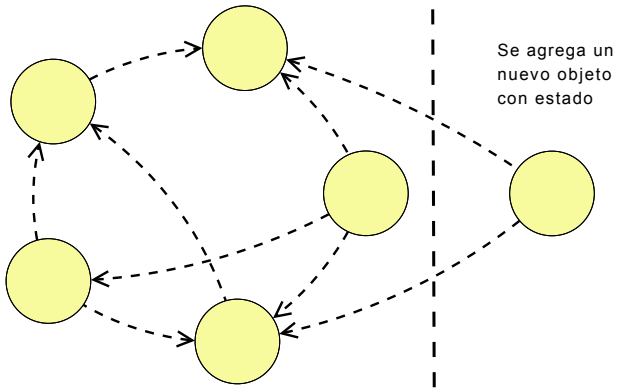

Figura 3.6: Composición de estados

En la versión imperativa (Figura 3.7), la problemática que surge es doble. Por un lado, se hace necesario modificar el sistema existente para contemplar la actualización del nuevo objeto (esto se ve en las aureolas coloradas sobre los objetos afectados). Por otro, es necesario agregar código para cada objeto del cual el objeto agregado depende, lo que se ve a partir de las dos flechas rojas.

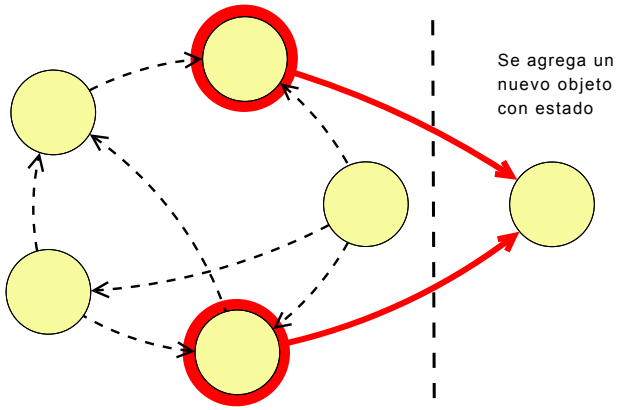

Figura 3.7: Composición de estados imperativa

En cambio, bajo un enfoque declarativo (Figura 3.8), tanto los objetos del sistema afectados como las nuevas actualizaciones que surgen a partir de la inclusión del nuevo objeto, son hechas automáticamente, lo que se indica con las flechas color azul. La aureola roja que rodea al nuevo objeto significa que el usuario todavía es responsable de especificar las dependencias del nuevo objeto. Por lo tanto, esta forma resulta ser composicional y adaptarse mucho mejor a un incremento en la cantidad de estados de la aplicación.

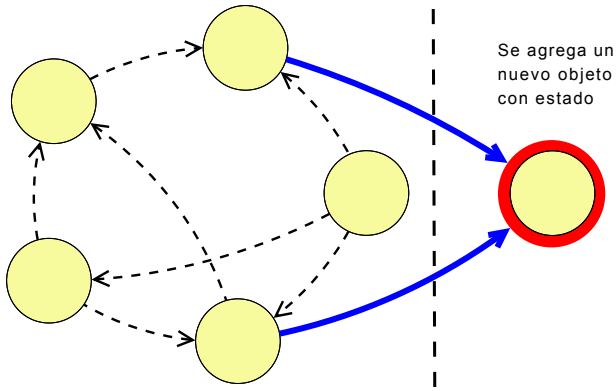

Figura 3.8: Composición de estados declarativa

4

Estado del arte

Para resolver los problemas del desarrollo web, existen ya varias alternativas conocidas. En los apartados que siguen se tendrán en cuenta las que se presentan a continuación:

4.1 Smalltalk: Seaside + GLORP

Smalltalk es un sistema informático que permite realizar tareas de computación mediante la interacción con un entorno de objetos virtuales. Metafóricamente, se puede considerar que un Smalltalk es un mundo virtual donde viven objetos que se comunican mediante el envío de mensajes [22].

Modelo

Glorp[33] es un framework para persistir objetos del lenguaje Smalltalk en una base de datos relacional.

Glorp se sirve de una implementación de un Lenguaje Específico de Dominio (DSL)[45] tanto para la especificación de parámetros de la base de datos como para la especificación de la forma en la cual los objetos son mapeados en tablas en la base de datos. Por ejemplo, el código del Snippet 4.1 especifica la forma que tendrá el Autor de una Producto en las tablas de la base de datos; el mapeo se hará "uno a muchos" entre una Autor y su Producto; además, el autor aparecerá como clave foránea en la tabla usada para persistir los productos bajo el nombre `AUTHOR_ID`, y ésta se corresponderá con el atributo `ID` en la tabla de Autores.

Glorp provee además un lenguaje de consulta similar a SQL, pero una vez más bajo la forma de DSL. Entonces, por ejemplo, obtener de la base de datos todas las productos cuyo Autor es 'Julio Verne', es posible hacerlo como en el Snippet 4.2.

Snippet 4.1 Mapping en Glorp

```
OneToManyMapping new
   attributeName: #author;
   referenceClass: Author;
   mapingCriteria: (PrimaryKeyExpression
      from: (productTable fieldNamed: 'AUTHOR_ID')
      to: (authorTable fieldNamed: 'ID').
```

Snippet 4.2 Query en Glorp

```
aSession executeQuery:
   (Query
      forManyOf: Product
      where: [:product |
         product author name = 'Julio Verne'].
```

Como se puede ver, el lenguaje de consulta aparece en forma de un DSL embebido en el lenguaje Smalltalk[29]

Controlador

Seaside[19] es un framework para el desarrollo de aplicaciones Web complejas en Smalltalk[22]. Sus características más singulares son una estructura de árbol de componentes en contraposición de una organización por página, y el uso de continuaciones para la especificación del flujo de control[41].

Seaside no es una solución "full-stack" como es, por ejemplo, Ruby On Rails; es decir que Seaside resuelve el problema de la presentación Web de una aplicación, pero deja de lado la forma en que se persisten los objetos de ésta. Ahora bien, esto trae aparejadas ciertas complicaciones. A modo de ejemplo, la naturaleza intrínseca de la programación basada en flujo de datos hace que ésta se distribuya a través de todas las capas de la aplicación. Esto sólo es posible cuando las distintas capas están estrictamente diseñadas e implementadas para ser utilizadas junto con las demás. De este hecho se desprende que una separación demasiado marcada entre las capas no es deseable y se hace evidente la necesidad de soluciones "full-stack" diseñadas teniendo en cuenta estas consideraciones.

Hay dos características que resaltan por sobre las demás, y resultan muy positivas sobre Seaside:

- La composición de vistas mediante un árbol de componentes en contraposición a un diseño por página; tal decisión representa una mejora notable en los aspectos navegacionales y de composición de vistas.

- El uso de continuaciones para la especificación del flujo de control. Esta mejora técnica permite reconciliar un flujo de control intrínsecamente no lineal como es el de la navegación Web, con un estilo de programación tradicionalmente lineal.

Vista

La generación de HTML se realiza de forma programática mediante un DSL embebido en el lenguaje Smalltalk, como se puede ver en el Snippet 4.3. Se hace incapié en el uso de HTML como representación estrictamente semántica; las particularidades estéticas quedan relegadas a una especificación mediante hojas de estilos (CSS).

Snippet 4.3 Vista en Seaside

```
renderContentOn: html
    html heading: product.
    html text: product price.
    html space.
    (html anchor)
  callback: [self buyNow];
  with: 'buy now!'
    html text: product author name.
```

Si bien tal diseño es aplicable en un gran número de casos prácticos, CSS no es suficientemente expresivo en determinados casos, lo que invalida el enfoque en cierto punto, ya que ésto obliga al programador a utilizar HTML con fines estéticos y de forma programática. Además, la geración de HTML se hace mediante un streaming de datos. Tal diseño ignora la naturaleza de árbol de todo lenguaje basado en XML; ésto hace más difícil el diseño de determinadas cosas como por ejemplo una actualización transparente de las vistas mediante flujo de datos e independiente de la forma en que se realicen (AJAX, COMET, etc).

4.1.1. Conclusión

Smalltalk tiene un sistema que es relativamente simple de usar, y está muy orientado a la creación de aplicaciones. Lo que le faltaría es por un lado una solución full-stack, que permita que se puedan integrar mas el controlador y el modelo, y por otro lado un manejo de la vista por separado del controller, debido a que que la separación por css no es suficiente para tener la flexibilidad necesaria para realizar una aplicación web.

4.2 Ruby on Rails

Ruby on Rails es un framework de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby, siguiendo el paradigma de la arquitectura Modelo Vista Controlador (MVC). Trata de combinar la simplicidad con la posibilidad de desarrollar aplicaciones del mundo real escribiendo menos código que con otros frameworks y con un mínimo de configuración. El lenguaje de programación Ruby permite la metaprogramación, de la cual Rails hace uso, lo que resulta en una sintaxis que muchos de sus usuarios encuentran muy legible [1]

Modelo

En Rails, la definición del modelo esta dividida en dos partes, una parte escrita en archivos de Ruby, y otra en el esquema de base de datos, lo que implica que los campos del modelo se encuentran directamente en la base, y toda otra información externa se agrega en el modelo (por ejemplo métodos y referencias a otras clases).

La implementación de la clase Producto y Autor quedan como en el Snippet 4.4, y la tabla para cada uno queda como en el Snippet 4.5.

Snippet 4.4 Rails - Clases Author y Product

```
class Author < ActiveRecord::Base
  has_many :products, :dependent => :destroy
end

class Product < ActiveRecord::Base
  belongs_to :author
end
```

Esto complica bastante la programación, siendo que toda la información respecto de un objeto del modelo se encuentra repartida entre estos dos lugares.

Para implementar un método que realice una actualización de la base de datos, se debe hacer como en el Snippet 4.6. El manejo de guardado de datos se hace explícitamente.

Para obtener los productos de un autor, ordenados por precio, se agrega el orden a la colección ya existente de productos del autor, como se ve en el Snippet 4.7.

Como se puede ver, las consultas a la base de datos utilizan directamente objetos de Ruby, lo cual simplifica el uso. Lo único que se especifica en otro lenguaje es el modelo de base de datos, ya que tanto las tablas como las columnas deben ser

Snippet 4.5 Rails - Tablas author y product

```
CREATE TABLE Author (
  id int(11) NOT NULL AUTO_INCREMENT,
  name text NOT NULL
  PRIMARY KEY (id))

CREATE TABLE Product (
  id INT NOT NULL AUTO_INCREMENT,
  name TEXT NOT NULL,
  author_id INT NOT NULL,
  price REAL NOT NULL,
  PRIMARY KEY (id))
```

Snippet 4.6 Rails - Actualización de datos

```
def discount(amount)
    price = price - amount
    self.save
end
```

Snippet 4.7 Rails - Consulta

```
Autor.products.reorder('price')
```

creadas directamente en la base de datos, pero luego el acceso, y las consultas, son directamente realizadas en Ruby.

El almacenamiento no es automático, es decir, cada cambio que se quiera persistir debe directamente ser seguido de un llamado correspondiente a la base de datos.

Para el manejo de transacciones, Rails soporta transacciones anidadas utilizando Savepoints.

Rails tiene un mecanismo de Scaffolding para generar pantallas de administración para los objetos del modelo, que genera los archivos necesarios para una administración simple de objetos, lista para ser modificados por el desarrollador (Snippet 4.8).

Esto, si bien simplifica mucho el trabajo inicial, duplica código similar para cada clase, lo cual conlleva todos los problemas del copy paste.

Las migraciones del modelo se realizan mediante `Migrations`, archivos de script que realizan directamente cambios en la base de datos y se realizan de manera ordenada. Esto permite versionar los cambios que se realizan.

Snippet 4.8 Rails - Archivos del Scaffolding

```
app/controllers/posts_controller.rb # The Posts controller
app/views/posts/index.html.erb # A view to display an index
                               # of all posts
app/views/posts/edit.html.erb # A view to edit an existing post
app/views/posts/show.html.erb # A view to display a single post
app/views/posts/new.html.erb # A view to create a new post
app/views/posts/_form.html.erb # A partial to control the
                               # overall look and feel of the
                               # form used in edit and new views
app/helpers/posts_helper.rb # Helper functions to be used from
                            #the post views
app/assets/stylesheets/posts.css.scss # Cascading style sheet
                                      # for the posts controller
app/assets/javascripts/posts.js.coffee # CoffeeScript for the
                                       # posts controller
```

Controlador

Rails utiliza un mecanismo de ruteo basado en páginas. Cada página define un controlador y una acción, y luego más parámetros. Es posible utilizar rutas especiales, con una forma distinta, que luego mediante un mecanismo interno rails traducirá también en un controlador y una acción. Al estar basado en páginas, la navegación por urls es directa. Es decir, la url `authors/index` va a mostrar el método `index` del controlador `AuthorsController`.

Snippet 4.9 Rails - Controller de listado

```
class AuthorsController < ApplicationController
  def index
    @authors = Authors.all  #La variable @authors va a poder ser
referenciada desde la vista
  end
end
```

Los controladores son creados en cada Request, por lo que para mantener información entre varios de ellos es necesario utilizar variables de formularios, o manejar explícitamente la sesión de usuario. Esto impide un manejo de estado más simple, ya que se hace un manejo explícito de un sector de memoria compartida para persistir información en cada request y para comunicar distintos controladores.

Para reutilizar código en controladores, es necesario utilizar Helpers, o jerarquías de controladores. Esto no permite reutilizar la vista de los mismos, sino que en

Snippet 4.10 Rails - Controller de actualización

```
def add_product
  cart = session[:shoppingcart]        #Obtengo el carrito de la
sesión
  card.addProduct(params[:product_id]) #Agrego el producto al
carrito
end
```

la vista habrá que explícitamente incluir el código a reutilizar. Esto impide un alto grado de reuso del código de los controladores, ya que reusar un controlador implica manejar explícitamente todas las interacciones entre el controlador que reusa otro. Por ejemplo, si programaramos un módulo de login, no podríamos fácilmente embeberlo en una página sin que esta página anfitriona no maneje explícitamente la información necesaria, ya que debe por un lado recibir la información de login que el usuario ha ingresado, enviarla a un controlador (que debemos crear manualmente) o al modelo, y con el resultado enviarlo a un partial que tengamos disponible que será llamado desde la página anfitrión. Por estos problemas, lo habitual que se ve generalmente es un formulario de login, que se incluye mediante un partial en la página anfitrión, y realiza lo más posible por Ajax (contra su propio controlador) y en caso de necesitarlo, forwardea el browser a su propia página para tener un control absoluto de la información que entra y sale. La solución 100 % embebida implica tal trabajo que no se realiza habitualmente.

Para realizar actualizaciones de interfaz por AJAX, hay que manualmente agregar el código necesario en las vistas y preparar acciones de controlador específicamente para tal fin. Rails tiene helpers que permiten simplificar el llamado a métodos Ajax (por ejemplo, `form_remote_tag` es un tag de form que al submitirse envía un request por Ajax). Por default, las actualizaciones de la pantalla son por recarga completa. Rails no tiene integración con Comet por default. Esto agrega una complejidad alta en cualquier aplicación que utilice estas tecnologías basadas en DHTML, que son la mayoría hoy en día.

Vista

Las vistas se generan desde archivos .html.erb, que son templates de html basados en Ruby.

La composición general de una página es:

- Layout
  - template

- partial
- partial
- partial

El layout es un template general, que se comparte en varios lugares del sitio (donde figura por ejemplo el menú de la aplicación, y todo el marco general que tendrá luego el contenido). Dentro del mismo, se encuentra el template, que es elegido específicamente para el par controller/action al que se esta llamando. Por último, los templates pueden incluir "templates parciales" que permiten reutilización de partes de la vista.

Por ejemplo, para mostrar un listado de productos, el template seria:

Snippet 4.11 Rails - Vista de listado de productos

```
<% for product in @products %>
    <%= render_partial 'product', product %>
    <hr />
<% end %>
```

Esto permite escribir un template para un producto, y que sea reutilizado en cada llamado dentro del for. También nos va a permitir utilizar este template desde cualquier otra página que necesitemos, lo que es importante es recordar en el controller asociado a esta nueva página, debemos dejar disponible en una variable con el producto en el controller.

Conclusión

Rails tiene mecanismos para realizar un muy rápido prototipado de funcionalidad, y para un manejo de una aplicación de baja complejidad, con métodos del modelo con poca modificación del grafo de objetos persistidos, y con un reuso básico de la vista, podemos generar aplicaciones muy rápidamente. Construir una aplicación de mayor complejidad va a implicar que luego todo el manejo dinámico de la vista, o la alta complejidad del modelo tendrá que ser manejada cuidadosamente para conseguir el resultado esperado.

4.3 Php: Symfony

Symfony es un completo framework diseñado para optimizar el desarrollo de las aplicaciones web mediante algunas de sus principales características. Para empezar, separa la lógica de negocio, la lógica de servidor y la presentación de la aplicación web. Proporciona varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de una aplicación web compleja. Además, automatiza las tareas más comunes, permitiendo al desarrollador dedicarse por completo a los aspectos específicos de cada aplicación. El resultado de todas estas ventajas es que no se debe reinventar la rueda cada vez que se crea una nueva aplicación web. Symfony está desarrollado completamente con PHP 5. Ha sido probado en numerosos proyectos reales y se utiliza en sitios web de comercio electrónico de primer nivel. Symfony es compatible con la mayoría de gestores de bases de datos, como MySQL, PostgreSQL, Oracle y Microsoft SQL Server. Se puede ejecutar tanto en plataformas *nix (Unix, Linux, etc.) como en plataformas Windows [4].

Symfony es un framework que tiene muchos conceptos que se imitan de Ruby on Rails (como otros frameworks en PHP, como CakePHP por ejemplo). Por eso es que veremos que mucha de las funcionalidades se asemejan bastante a lo que ya vimos de Rails (4.2).

Modelo

Symfony utiliza dos motores para la administración de la base de Datos: Propel y Doctrine. Nosotros analizaremos Doctrine por ser el que viene habilitado por default en la distribución 1.4. En Doctrine, el modelo se describe en archivos Yaml [7]. Estos contienen la información necesaria para generar por un lado la base de datos, y por otro lado las clases del modelo.

Por ejemplo, para tener las clases Producto y Autor, las se deben describir en el schema.yml (Snippet 4.12), y eso genera tablas en la base de datos (Snippet 4.13). Entre las dos tablas, debido a la relación entre las clases especificada en el .yml, se agrega una Foreign Key. También se generan varias clases para el manejo de objetos de estas entidades (Snippet 4.14).

En los archivos del mismo nombre que los de la entidad, se crean clases que nos van a permitir escribir métodos del modelo. En los archivos de Base, se encuentran todas las referencias a los campos y a accesos a las bases de datos. De esta manera, ante un cambio en el modelo, Symfony puede regenerar los archivos Base completamente, dejando sin tocar los otros archivos, permitiendo que el desarrollador actualice el modelo sin tener que volver a escribir el cuerpo de las clases.

Symfony también genera archivos Filter y Form para cada una de las clases, que van a permitir, el primero un filtrado de las colecciones de entidades, y el segundo

Snippet 4.12 Symfony - Definición del modelo

```
Author:
  columns:
    name: string
Product:
  columns:
    name: string
    author_id: integer
    price: double
  references:
    Author:
```

Snippet 4.13 Symfony - Tablas generadas

```
CREATE TABLE Author (
  id int(11) NOT NULL AUTO_INCREMENT,
  name text NOT NULL
  PRIMARY KEY (id))

CREATE TABLE Product (
        id INT NOT NULL AUTO_INCREMENT,
        name TEXT NOT NULL,
        author_id INT NOT NULL,
        price REAL NOT NULL,
        PRIMARY KEY (id))
```

Snippet 4.14 Symfony - Clases generadas

```
Product.class.php
BaseProduct.class.php
Author.class.php
BaseAuthor.class.php
```

una edición simple y validada de las entidades individuales. Ambas clases son utilizadas por el generador de scaffolding, que genera archivos con controladores y vistas necesarios para una administración básica de las entidades. Luego el programador deberá customizar vista y/o controlador en caso de que quiera una edición avanzada.

Otro mecanismo con el que cuenta Symfony es el de los Admin-Generator, que a partir de un archivo descriptor del módulo de administración de una entidad, y las clases Filter y Form anteriormente generadas, genera dinámicamente las pantallas de administración. Esto permite luego, que si se modificaran las definiciones de las entidades, los módulos generados se actualicen y permitan una

administración apropiada. Los Admin-Generator por lo tanto, tienen una posibilidad mayor de adaptarse a los cambios del modelo que el scaffolding, pero para esto restringen también mucho lo que se puede hacer dentro de estas vistas, perdiendo flexibilidad.

Para hacer una consulta en el modelo, doctrine tiene dos mecanismos: DQL y Criteria.

Para obtener por ejemplo todos los productos de un autor, ordenados por precio, se utiliza DQL (Snippet 4.15). Esto provee un lenguaje simple para hacer las consultas.

Snippet 4.15 Symfony - Consulta en DQL

```
$q = Doctrine_Query::create()
    ->from('Product p')
    ->where('p.author_id = ?', $author->getId())
    ->orderBy('p.price');
$prods = $q->execute();
```

Doctrine no soporta transacciones anidadas, aunque no genera errores si se inicia una transacción dentro de otra. El problema es que un rollback en una transacción interna no tendrá el efecto esperado si se intenta recuperar del error y comitear finalmente. [8]

Controlador

Symfony tine un esquema de navegación basado en urls y páginas. Es decir, que permite que cada página sea accedida por su propia url.

Estas urls definen un controlador y una acción, la cual será ejecutada en caso de que el usuario tenga permisos suficientes.

El controlador para listar los autores se ve en el Snippet 4.16.

Snippet 4.16 Symfony - Controller para listado

```
class AuthorsController extends {
   public function executeIndex(){
           $this->authors =
                   Doctrine::getTable('Authors')->findAll();
   }
}
```

La información que debe persistir entre requerimientos de usuario es guardada en la sesión, ya que los controladores no poseen estado. También se pueden pasar mediante parámetros en el request. Este mecanismo se encuentra presente en

PHP, y Symfony solamente agrega clases para manejarlo de manera un poco más simple. Esto se maneja de manera idéntica a Ruby on Rails.

El manejo de AJAX se hace manualmente mediante el uso específico de librerías javascript (por ejemplo, jQuery o Prototype). Symfony no provee helpers que realicen este trabajo, por lo que la integración con Ajax se deja librada al programador y a la librería javascript que haya elegido.

Vista

La organización de las vistas es similar a Ruby on Rails. La organización de los archivos de templates es:

- Layout
  - template
    - partial
    - partial
    - partial

Los partials son partes de la vista que pueden ser reusados. Aquí vale el análisis hecho para Ruby on Rails por su similitud.

Conclusión

Symfony es un framework muy similar a Ruby on Rails, dado que surgió de los mismos conceptos, y comparte una buena parte de sus funcionalidades e ideas. Una aplicación hecha en symfony por lo tanto va a poder ser prototipada muy fácilmente, y luego tendrá un trabajo mayor y más lento cuando se llegue al momento de realizar el trabajo avanzado de modificación de modelo, de manejo dinámico de vistas, composición y reuso de controladores, y aquellas cuestiones que se alejan de lo que es una administración de datos simple mediante el uso de una web.

4.4 Java: Hibernate + JSF + Seam

En Java hay múltiples frameworks y librerías para desarrollar Aplicaciones Web. Hemos elegido un conjunto de los mismos, considerando que son los más populares y con más apoyo de la comunidad, si bien hay muchos otros que puedan tener características similares.

Los elegidos son: JBoss Seam, Hibernate y JSF.

Modelo

Del sitio de Hibernate: "Hibernate is a powerful, high performance object/relational persistence and query service. Hibernate lets you develop persistent classes following object-oriented idiom - including association, inheritance, polymorphism, composition, and collections. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API." [9]

La especificación del modelo en Hibernate se puede hacer 100 % en Java. Al escribir la clase, se especifica mediante Annotations que la clase debe ser persistida, y Hibernate se encarga automáticamente de persistirla. Al ser Java fuertemente tipado, los tipos de las variables también alcanzan para definir los campos de la base de datos (Snippet 4.17).

Las clases del modelo, además de contener los campos tipados, deben tener métodos de acceso a las variables, lo cual impone trabajo extra en el programador. De todos modos, las IDEs más usadas de Java generan estos métodos automáticamente (Snippet 4.18).

Esto es algo positivo, porque la especificación de una clase queda en un sólo lugar, lo cual facilita el entendimiento del sistema y simplifica las modificaciones.

Para implementar un método del modelo, que modifique datos, si el objeto ya estaba persistido en la base de datos, y se está dentro de una transacción, entonces al cerrar la transacción el objeto será actualizado (Snippet 4.19).

Es más complicado el caso en que se quieran crear nuevos objetos, o eliminarlos, ya que habrá que realizarlo explícitamente. Hibernate provee la posibilidad de indicar que una relación entre 2 objetos tendrá *cascading*, lo que indica de una modificación a uno (actualización, borrado) se realizará también en el objeto relacionado.

Hibernate posee un lenguaje de consulta (el HQL) [11] que permite expresar consultas complejas en la base de datos, utilizando el modelo en objetos.

Java (J2EE) no soporta transacciones anidadas. [12]

Java Seam tiene un generador de formularios de adminsitración, similar a los de Ruby on Rails y Symfony.

Snippet 4.17 Java - Definición del modelo - Author

```
@Entity
@Table(name = "author")
public class Author implements Serializable
  public Author() {

  }
  @Id
  @Column(name = "id")
  private Integer id;

  @Column(name = "name")
  private String name;

  @OneToMany (cascade = CascadeType.ALL, mappedBy = "author")
  private List<Product> products;

  public Integer getId() {
  return id;
  }

  public void setId(Integer id) {
  this.id = id;
  }

  public String getName() {
  return name;
  }

  public void setName(String name) {
  this.name = name;
  }
}
```

Controlador

Del sitio web de Seam: "Seam is a powerful open source development platform for building rich Internet applications in Java. Seam integrates technologies such as Asynchronous JavaScript and XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) and Business Process Management (BPM) into a unified full-stack solution, complete with sophisticated tooling." [10]

Seam se basa en `Beans`, que son objetos Java utilizados para comunicar distin-

Snippet 4.18 Java - Definición del modelo - Product

```
@Entity
@Table(name = "product")
public class Product implements Serializable
  public Product() {

  }
  @Id
  @Column(name = "id")
  private Integer id;

  @Column(name = "name")
  private String name;

  @ManyToOne
  private Author author;

  public Integer getId() {
  return id;
  }

  public void setId(Integer id) {
  this.id = id;
  }

  public String getName() {
  return name;
  }

  public void setName(String name) {
  this.name = name;
  }


}
```

tas capas de la aplicación. Por ejemplo, para implementar un controlador que permita listar a todos los Autores, se debe hacer el `Bean` para un autor (Snippet 4.21), declarar la interface para el `Bean` de la lista (Snippet 4.22), e implementar el `Bean` de la lista (Snippet 4.23).

Como se puede ver, la creación de un controlador en Java requiere cumplir con ciertas reglas, lo cual lleva a tener una cantidad considerable de código escrito. Los datos de sesión son los que están guardados dentro del objeto

Snippet 4.19 Java - Actualización del modelo

```
Author author = (Author) sess.load( Author.class, new Long(69)
);
author.setName("Erich Fromm");
sess.flush();  // los cambios al autor son automáticamente
detectados y persistidos
```

Snippet 4.20 Java - Consulta en HQL

```
Query q = s.createQuery("from Authors as a order by a.name
asc");
List authors = q.list();
```

`AuthorsManagerBean`, por lo que el manejo de sesiones termina siendo más simple ya que hay un contexto simple que se persiste de llamado en llamado.

Vista

Del sitio de JSF: "JavaServer(TM) Faces technology simplifies building user interfaces for JavaServer applications. Developers of various skill levels can quickly build web applications by: assembling reusable UI components in a page; connecting these components to an application data source; and wiring client-generated events to server-side event handlers." [13]

La vista para el controlador de Autores se encuentra definida en el Snippet 4.22. Aquí se llama a los `Bean` que definimos anteriormente, y se muestra la información que contienen. Como se puede notar, las que llevan el control de lo que se debe mostrar son las vistas, que incluyen beans, y no al revés. Las vistas de JSF pueden componerse, utilizando tags del tipo `<ui:include src="inc.jspf" />`. Al ser compuestas de esta manera, y las vistas invocar los `Beans`, se puede llegar a una composición mucho más amplia que en Ruby on Rails o Symfony, ya que en una vista podemos incluir otra, y en esta otra se genera la salida basada en los `Beans`.

Conclusión

Debido a que Java es un lenguaje más orientado al desarrollo de aplicaciones de negocios que requieren un alto grado de seguridad y confiabilidad, los procesos de desarrollo de aplicaciones en Java tienden a ser largos y rígidos, y las herramientas están orientadas por lo tanto a formar parte de estos procesos, tendiendo a ser también rígidas y requiriendo mucho trabajo para hacerlas

funcionar completamente. Si bien, las aplicaciones de negocios tienen estos requerimientos, el enfoque de esta tesis se orienta hacia un rápido desarrollo de aplicaciones completas y con mayor flexibilidad.

Snippet 4.21 Java - Bean de Author

```
@Entity
@Name("author")
@Scope(EVENT)
public class AuthorBean implements Serializable

{
   private Long id;
   private String name;
   @Id @GeneratedValue

   public Long getId()
   {
      return id;
   }

   public void setId(Long id)
   {
      this.id = id;
   }

   @NotNull @Length(max=100)
   public String getName()
   {
      return name;
   }

   public void setName(String name)
   {
      this.name = name;
   }
}
```

Snippet 4.22 Java - Interface de AuthorManager

```
@Local
public interface AuthorManager
{
   public void findAuthors();
}
```

Snippet 4.23 Java - Bean de lista de Author

```
@Stateful
@Scope(SESSION)
@Name("authorsManager")
public class AuthorsManagerBean implements Serializable,
AuthorsManager
{
   @DataModel
   private List<AuthorBean> authorList;

   @DataModelSelection
   @Out(required=false)
   private Author author;

   @PersistenceContext(type=EXTENDED)
   private EntityManager em;

   @Factory("authorsList")
   public void findAuthors()
   {
      authorsList = em.createQuery("from Author a order by
a.name desc").getResultList();
   }
}
```

Snippet 4.24 Java - Vista de Lista de Autores

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
 <head>
  <title>Authors</title>
 </head>
 <body>
  <f:view>
   <h:form>
     <h2>Authors List</h2>
     <h:outputText value="No authors to display"
                   rendered="#{authorsList.rowCount==0}"/>
     <h:dataTable var="aut" value="#{authorsList}"
                  rendered="#{authorsList.rowCount>0}">
        <h:column>
           <f:facet name="header">
              <h:outputText value="Name"/>
           </f:facet>
           <h:commandLink value="#{aut.name}"
                action="#{messageManager.select}"/>
        </h:column>
     </h:dataTable>
     <h3><h:outputText value="#{message.name}"/></h3>
   </h:form>
  </f:view>
 </body>
</html>
```

The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man

George Bernard Shaw

5

Un framework para enfrentarlos a todos

En este capítulo se describen los aportes hechos para solucionar los problemas enunciados en el capítulo anterior. Una vez más, el capítulo está estructurado según la capa que se resuelve, es decir, se presenta una sección para el modelo, otra para el controlador y una para la vista.

5.1 Modelo

5.1.1. Mapeo objeto-relacional

En la solución desarrollada, se prefirió que el modelo se especifique dentro del lenguaje de programación (es decir, PHP) que es el que el programador va a utilizar para el resto de la aplicación. Se tuvo que agregar metainformación al mismo, como los tipos de las variables, valores default, si un atributo es index o no, ya que esta información no se encuentra en PHP por no ser tipado.

Snippet 5.1 Kusala- Declaración de la clase Author

```
class Author extends PersistentObject {
        function initialize() {
                $this->addField(new TextField('name', array (
                        'is_index' => true
                )));
                $this->addField(new CollectionField(array
                     ('fieldName'=>'products',
                      'type' =>Product,
                       'reverseField'=>'author')));
        }
}
```

Los objetos del modelo extienden de una clase `PersistentObject`, que posee los métodos de instancia `#save` y `#delete`, y el método de clase `#getById`, que recibe el ID del objeto a recuperar. Cuando un objeto se intenta recuperar más de una vez, siempre es devuelta la misma instancia, manteniendo la idea de que el modelo de la aplicación es uno solo y que una modificación a un objeto afecta el estado global de la misma. Para esto se mantiene una tabla en memoria con los objetos presentes para que la búsqueda sea directa. Luego, la herencia se especifica de la forma natural en PHP y **Kusala** se encarga automáticamente de mapear esta información.

Las variables de instancia son `ValueHolders`, lo cual permite un uso avanzado más adelante, en la generación y actualización de vistas.

Snippet 5.2 Kusala - Guardado y recuperación de objetos

```
$author = Author::getById("Author", 1);

$author->name->setValue("Erich Fromm");

//El objeto se guarda en la base de datos
$author->save();

$author2 = Author::getById("Author", 1);

$author==$author2 //Esto es true, las 2 instancias son la misma

//El objeto es borrado
$author->delete();
```

5.1.2. Implementación de métodos que modifican el modelo

Se decidió implementar un modelo de persistencia por alcance, que tenga garbage collection. De esta manera se libra al programador de recordar las referencias que se crean y de las que se liberan.

Los objetos que pertenecen al modelo están marcados internamente como pertenecientes. Cuando un objeto nuevo se crea, al asociarse a uno existente se marca como persistente también. Cualquier modificación al modelo de objetos impacta en el estado global de la aplicación, liberando de esta manera al programador del trabajo de llevar el listado de objetos modificados.

En la implementación del garbage collector se tuvo en cuenta el trabajo de [17], adaptándolo a un modelo de base de datos. En el mismo, al remover un enlace

Snippet 5.3 Kusala - Persistencia por Alcance

```
//Se relaciona el Producto con el Autor

$author->products->add($product);

//El producto queda persistido por la persistencia por alcance.

//El producto se elimina
$author->products->remove($product);
```

de un objeto que tenga la posibilidad de pertenecer a un ciclo, se recorre el grafo de objetos iniciando desde el objeto borrado, y en caso de contener tantos enlaces entrantes como llegadas a partir de si mismo, se decide que es posible borrarlo.

5.1.3. Consulta del modelo

Para la implementación, fue creado el lenguaje OQL, intentando llegar a un intermedio entre el lenguaje OQL de ODMG y el lenguaje SQL estándard.

Snippet 5.4 Kusala - Ejemplo de OQL

```
select Product p where p.author is $author order by p.price
```

Para simplificar su utilización, fue agregada la utilización de macros (sección 6.5), que precompilan objetos Report y Condition que en tiempo de ejecución se instancian para generar la query SQL correspondiente. Las queries OQL son parametrizables de esta manera con las variables del contexto.

Snippet 5.5 Kusala - Uso de OQL dentro del programa

```
$author = Author::getById($author, 1);

$products = #@select Product p where p.author is $author order
by p.price@#;

$products->first();
```

El lenguaje OQL fue escrito mediante una variación del lenguaje EBNF, y es

compilado usando PHPCC (sección 6.3), un compiler-compiler que se debió generar para tal fin.

Snippet 5.6 Kusala - Gramática de OQL

```
<oql(
        identifier::=/[a-z_][a-z_0-9]*/i.
        phpvar::=/\$[a-z_][a-z_0-9]*/i.
        condition::=subexpression=>"(" <expression> ")"
                |comparison=><value>
                        /=|\<\>|<=|>=|\>|\<|LIKE|IS/i
                        <value>.
        valueorfunction::=<identifier>
                ["(" {(<valueorfunction>|| <value>); ","} ")"].
        expression::=not=>/NOT/i <expression>|
                     exists=>/EXISTS/i "(" <oql> ")"|
                     in=><variable> /IN/i "(" <oql> ")"||
                     logical=><condition>
                        [operator->/AND|OR/i <expression>].
        oql::=class->[name=>[<identifier> ":"]<identifier>|
                     phpvar=><phpvar>|
                     path=>{<identifier> ; "."}
                        "as" <identifier>]
              fields->["(" fields->{<valueorfunction>
                                "as" <identifier> ; ","} ")"]
              from->["from" from->{var-><identifier> ":"
                                class-><identifier> ; ","}]
              where->["where" expression-><expression>]
              order->[/order by/i {<variable> /desc|asc/i;","}]
              limit->[/limit/i <number>].
        variable::={<identifier> ; "."}.
        plainsql::=/\[[^\]]+\]/.
        number::=/[0-9]+/.
        value::=value=>(
                number=><number>|
                str=>/\'[^\']*\'/|
                phpvar=><phpvar>|
                bool=>/TRUE|FALSE/i|
                plainsql=><plainsql>||
                aggregate=>"("<oql>")")||
                var=><variable>.
)>
```

5.1.4. Integridad de los datos en memoria

Durante la realización de una transacción, en la aplicación se persisten cada uno de los cambios que el usuario realizó sobre los datos, desde la memoria en PHP hacia la base de datos.

Por ejemplo, si lo que se intenta realizar es una modificación a los datos del usuario, se podría modificar detalles de su dirección, y al mismo tiempo, su nombre de usuario. Al guardar esta información, el nombre de usuario es chequeado y en la base de datos se detecta que el nombre de usuario nuevo está repetido. Es importante entonces que, además de que la transacción sea revertida, que el estado en memoria de los datos refleje que los cambios no fueron realizados. De esta manera, al solucionar el error, la transacción puede reintentarse y el guardado ser realizado completamente.

5.1.5. Modificación de objetos desactualizados

Al desarrollar una aplicación concurrente, que comparte un espacio de memoria (en este caso la base de datos) es habitual tener que resolver problemas de sincronización de datos. Las transacciones de base de datos solucionan este problema dentro de un mismo requerimiento del usuario, pero la interacción extendida en el tiempo (que sucede en múltiples transacciones) lo vuelve a generar.

Por ejemplo, si un usuario carga un producto en el carrito, del que queda solamente una unidad, y antes de hacer el Checkout otro usuario compra la última existencia de ese producto, es importante no realizar la venta a este usuario, y también darle una respuesta y permitirle seleccionar otro producto en reemplazo del mismo.

En **Kusala** existe en cada objeto una variable de instancia `version`, que se incrementa ante cada guardado. De esta manera, cuando un objeto es recuperado, se tiene la información de la versión correspondiente, y cuando se intenta guardar se chequea si la versión fue actualizada por otro proceso, y en ese caso una excepción es creada y la transacción abortada, lo que deja en manos del desarrollador la posibilidad de darle al usuario, por ejemplo, una pantalla para reemplazar el producto en el carrito por uno similar con stock.

5.1.6. Observabilidad de los cambios

Como fue mencionado en 2.3.1, para definir este problema es mejor presentar un ejemplo: Se quiere implementar un componente de interfaz para la selección de productos de un autor. Se presentan dos listas, una con los productos ya seleccionados, la otra con los objetos aun no seleccionados. Hay además dos botones que permiten cambiar elementos de una lista y moverlos a la otra, y por último un botón de Cancelar y otro de Aceptar.

Snippet 5.7 Kusala - Recuperación de un error de transacción

```
$address = $client->address->first();
$address->city->setValue("La Plata");

$client->username->setValue("pepe");
//Este nombre de usuario ya se encuentra existente en la base de
datos

$session = DBSession::Instance();
$session->beginTransaction();
$session->save($address);
$session->save($client);

try{
  $session->commit();
} (DBException $e){
  //Esto genera una excepción, ya que el nombre se encuentra
repetido
}

$address->isModified(); //true - El objeto aún debe ser
persistido

$client->username->setValue("pepe1982");
//Este nombre de usuario no existe en la base de datos

$session->commit();

$address->isModified(); //false - El objeto fue persistido
$client->isModified(); //false - El objeto fue persistido
```

La solución a estos problemas consiste en hacer los cambios a las colecciones observables localmente. Es decir, que los cambios realizados a la colección sean visibles inmediatamente. Agregar un objeto a una colección debería incluirlo a los objetos devueltos cuando se consulta la misma. Algo que parece trivial cuando se habla de colecciones en memoria, es mas difícil cuando se utilizan colecciones de datos en la base y transacciones, ya que no es deseable que la base de datos se vea modificada fuera de la transacción.

Esto supone restricciones sobre la política transaccional de la base de datos. En particular, es necesario un control de concurrencia optimista para las transacciones y la posibilidad de mantener transacciones abiertas durante múltiples

Snippet 5.8 Kusala - Consulta a una colección modificada

```
$products = #@select Product p where p.stock = 0@#
//Seleccionamos todos los productos sin stock

$libro = Product::getWithId("Product", 1);
//Este libro tiene un stock de 10

$products->includes($libro); //false - El producto tiene stock.

$libro->stock->setValue(0); //Realizamos el cambio en memoria,
sin afectar la base de datos

//EN OTRO REQUEST DE LA APLICACIÓN

$products->includes($libro); //true - ahora el producto no
tiene stock
```

requerimientos de la aplicación. La mayor dificultad es que la mayoría de los motores de base de datos no utilizan control de concurrencia optimista; la posibilidad de mantener transacciones abiertas es aún más inédita.

En la implementación, la aplicación mantiene un log de las modificaciones a realizar en la base de datos, que es ejecutado al principio de cada request. Al finalizar el Request, la transacción es revertida, para evitar que las modificaciones se persistan sin la intención del usuario.

5.1.7. Modificación del esquema del modelo

Después de unos meses de tener la aplicación funcionando, es común que surjan actualizaciones a la misma, para adaptarse mejor al contexto en el que están funcionando. Es importante entonces que estas modificaciones puedan realizarse de la manera más simple posible y con la menor cantidad de errores.

Por ejemplo, en el supuesto caso de que a la aplicación fuera necesario agregarle al producto una fecha de publicación, se necesita entonces que:

- El modelo de la base de datos se adapte a la nueva forma,
- Los datos existentes se mantengan, y que sean consistentes,
- Las pantallas de administración asociadas permitan editar el nuevo campo

En **Kusala**, luego de realizar las modificaciones correspondientes a las clases del modelo, el desarrollador puede ingresar a la plantalla de administración de la aplicación (Figura 5.1). Allí puede ir a la pantalla de administración de la base de datos, y revisar si existen diferencias entre el modelo de la base de datos y el modelo en objetos (Figura 5.2). En caso de que así sea, se listarán un conjunto de SQL que permitirán cambiar la base de datos y adaptarla al modelo. Estos SQL se presentan al desarrollador y no se ejecutan directamente, para permitirle tener un mayor control sobre los cambios a realizar (por ejemplo, el sistema detectará un cambio en el nombre de un campo como un borrado de una columna y una creación de una nueva, cuando esto sería incorrecto ya que los datos contenidos en la misma se perderían).

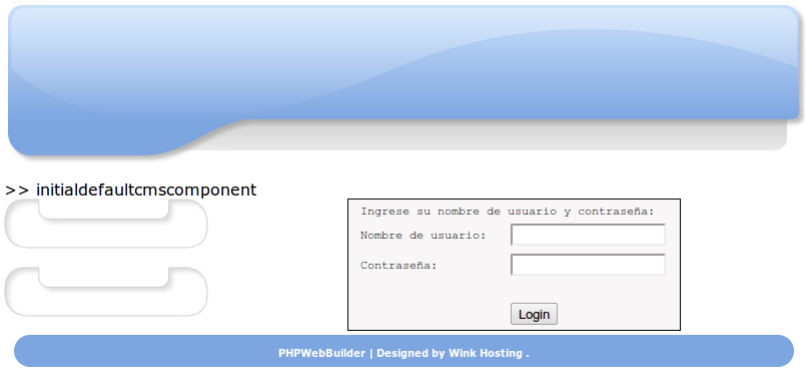

Figura 5.1: **Kusala** - Pantalla de administración del sistema

5.1.8. Administración de objetos

Dentro del Framework, el enfoque utilizado es la generación de la administración a partir del modelo. Existe una clase `ObjectAdmin` que se encarga de presentar un editor con los campos del objeto presentado (a quien evalúa mediante Reflection [20]) y un `Widget` de edición apropiado respecto del tipo del campo, es decir, si es un `String`, para editar el nombre del producto, presenta un `TextInput`, si es una fecha, para editar la fecha de publicación, es un `DateInput`, que contiene un date picker, y si es una referencia a objetos de otra clase, por ejemplo para elegir el `Autor`, un combo de selección.

Esta clase se puede extender para agregarle el comportamiento necesario, ya sea ocultar campos del formulario que no puedan ser editados, o para agregarle, por ejemplo, la cantidad de ventas que tuvo el producto. Se puede agregar simulando ser un campo más del objeto, pero que en lugar de un `TextInput` para la edición, tenga un `Text`, para que solamente se muestre.

Tanto la clase `ObjectAdmin` como la `ObjectsAdmin`, que administra una colección de objetos, utilizan extensamente *multiple dispatching* y *context dispatching*

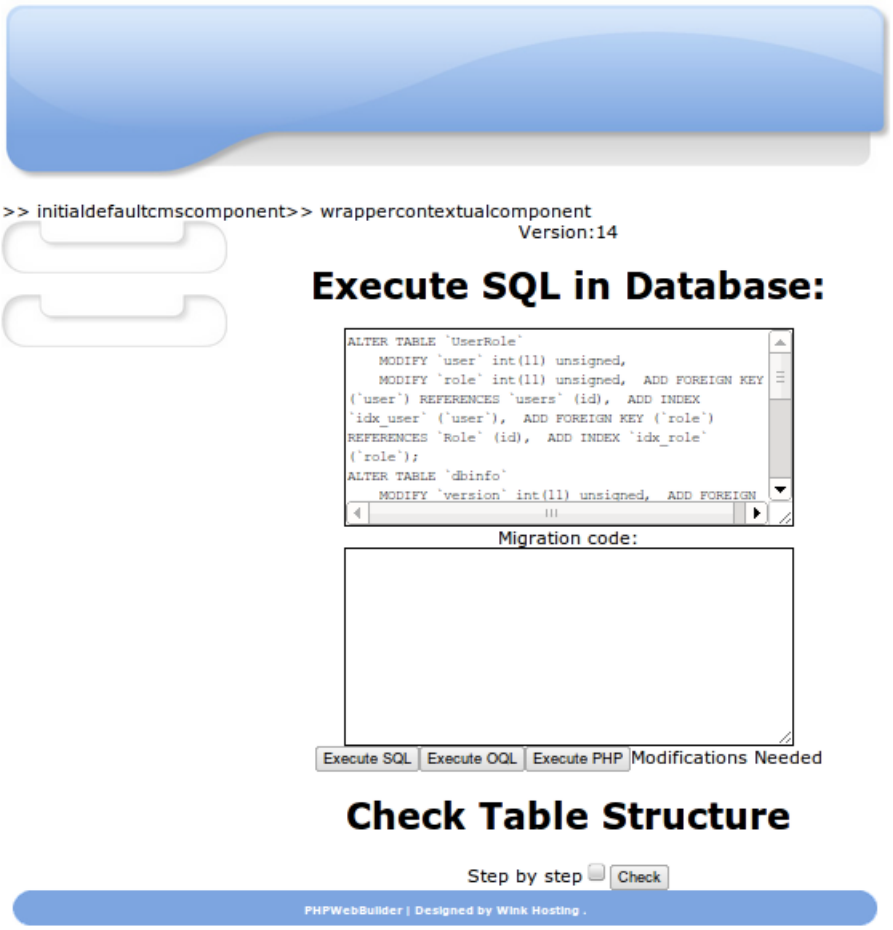

Figura 5.2: **Kusala** - Pantalla de diferencias de la base de datos

	Responde a los cambios del modelo	Permite agregado de datos al formulario
Programación desde 0	No	Sí
Generación de código	No	Sí
Reflection durante la ejecución	Sí	Sí

Cuadro 5.1: Comparación de soluciones a la administración de objetos

(sección 6.1) para elegir el mejor editor para cada tipo de objeto y campo.

Se consideró que una opción que utilice reflection en tiempo de ejecución es mejor, ya que cuando se utiliza reflection para generar código, existe el riesgo (y es algo que sucede habitualmente) de que se genere código sobre clases que luego van a ser modificadas, y que las modificaciones que se hagan para adaptación del código generado, luego deban ser integradas al nuevo código generado que se obtenga de ejecutar los comandos sobre el modelo modificado. En la tabla de 5.1 se puede ver una comparación entre distintas técnicas.

5.2 Controlador

5.2.1. Navegabilidad y composición de interfaces

Anteriormente se vió que los enfoques al momento de definir el control de flujo de las aplicaciones Web eran dos: uno orientado a la conexión entre páginas y otro basado en un árbol de componentes. Se vió que el primero se adapta mejor a la forma en que trabaja la Web; pero sufre de problemas de composicionalidad, especialmente al momento de controlar el flujo de aplicaciones complejas. El segundo simplificaba el control de flujo y promulgaba una mejor composicionalidad, aunque se aleja de la metáfora de páginas interconectadas y por lo tanto se hace más difícil resolver algunos problemas relacionados con la navegabilidad, como el tratamiento de los botones Atrás y Adelante y el bookmarking de documentos.

El enfoque elegido se basa en la modificación de un árbol de componentes. El diseño es muy similar al que presenta el framework Seaside. Como en Seaside, el control de flujo es modal en cuanto existen dos operaciones para modificar el árbol: *call* y *answer*. La operación *call* desactiva el componente actual y pasa el control y foco al componente llamado. Como consecuencia, la representación visual del componente llamador desaparece y ocupa su lugar el componente llamado; además, el flujo de control pasa al componente llamado, esto es, se ejecutan las operaciones definidas en él. Una vez que un componente termina su tarea, realiza una llamada a *answer* con el resultado de la operación. En ese momento el componente pierde el foco y el control de flujo retorna al componente que lo llamó, el cual retoma la ejecución.

Por ejemplo, para hacer que en el componente de vista de un producto se vea al autor relacionado se hace como en el Snippet 5.9.

Snippet 5.9 Kusala - Llamada desde controlador

```
class ProductView extends Controller{
  public function ProductView($product){
    $this->product=$product;
    parent::Controller();
  }
  public function initialize(){
    $this->addComponent(new Text(
            $this->product->name), 'name');
    $this->addComponent(
            new Text($this->product->price), 'price');
    $this->addComponent(
            new CommandLink(array(
                'text'=>$this->product->
                        author->getTarget()->name,
                'proceedFunction'=>
                        new FunctionObject($this, 'showAuthor')

            ));
    $this->addComponent(
            new CommandLink(...), 'buy_now');
  }
  public function showAuthor(){
     $this->call(new AuthorView(
                    $this->product->author->getTarget()));
  }
  ...
}
```

Además, los componentes forman un árbol en cuanto a que contienen componentes hijos. Esta particularidad es la que hace a la solución composicional; no hay límites en cuanto a la forma en que podemos construir el árbol de componentes. Una consecuencia de este diseño es que los componentes bien diseñados son extremadamente fáciles de reusar.

No se explayará más sobre el tema ya que este es tratado en profundidad en varias publicaciones y tutoriales [38, 19].

Si en lugar de que el componente de visualización de producto llame al componente de autor, se quisiera que lo muestre dentro de la misma página, es posible hacer como en el Snippet 5.10.

Ahora bien, la herramienta presenta algunas variaciones respecto a este diseño.

Snippet 5.10 Kusala - Composición de componentes

```
class ProductView extends Controller{
  ...
  public function initialize(){
    $this->addComponent(
            new Text($this->product->name), 'name');
    $this->addComponent(
            new Text($this->product->price), 'price');
    //El componente AuthorView es agregado como cualquier otro
    $this->addComponent(
            new AuthorView($this->product->author->getTarget());

    $this->addComponent(
            new CommandLink(...), 'buy_now');
  }
  ...
}
```

Por un lado, el flujo de control no es especificado de forma lineal (por que para eso serían necesarias continuaciones, como en Seaside), aunque un efecto similar se logra mediante callbacks. Por otro lado, no se especifican los componentes hijos de forma estática, sino que son agregados (comúnmente en un método inicializador) y removidos dinámicamente.

5.2.2. Actualización de vistas

Anteriormente se vió como la actualización de las vistas en los frameworks analizados se hace bajo un esquema imperativo. Esto es, existe una etapa separada y especialmente dedicada a imprimir el HTML ya sea de los componentes o de las páginas modificadas. Además, se vió que este esquema tiene la desventaja de quedar acoplado a la forma de interacción; esto quiere decir que la forma de interacción queda determinada por la forma en que se hace la actualización de las vistas. Entonces, por ejemplo, no es posible cambiar a una forma de interacción asincrónica globalmente; esta debe ser especificada de forma manual. En este apartado se verá cómo cambiando la forma en la cual se actualizan las vistas, se permite un desacoplamiento en cuanto a la forma de interacción.

La herramienta propone una actualización de vistas por medio de flujo de datos. Esto significa que, por un lado, las partes cambiantes de la vista deben definirse declarativamente. Esto quiere decir, no son permitidas operaciones imperativas del lado de la vista. Esto le otorga al controlador de la aplicación control completo acerca de cómo y cuando se hacen las modificaciones a la vista. De esta manera, la forma de interacción queda totalmente desacoplada de la actualiza-

ción de vistas; un cambio en el algoritmo que define la forma interacción hace que sea posible la interacción sincrónica o asincrónica, indistintamente.

Como ejemplo, se puede implementar una calculadora en el precio del producto. Si el usuario elige que quiere más de una copia del producto, el precio final se actualizará correspondientemente (Snippet 5.11)

Snippet 5.11 Kusala -Actualización de datos de un controlador

```
class ProductView extends Controller{
  ...
  public function initialize(){
    $this->addComponent(
            new Text($this->product->name), 'name');
    $this->addComponent(
            new Text($this->product->price), 'price');
    //Agregamos un input para que
    //el usuario ponga la cantidad
    $this->addComponent(
            new Input($null=null), 'qty');
    //Observamos el evento change de 'qty'
    $this->qty->onChangeSend('qtyChanged', $this);
    //Mostramos el precio total
    $this->addComponent(
            new Text($null=null), 'totalPrice');
    //Ponemos el valor en uno, que además
    //ejecuta el evento para incializar
    $this->qty->setValue(1);
    $this->addComponent(
            new CommandLink(...), 'buy_now');
  }
  public function qtyChanged(){
    //Multiplicamos el precio por la cantidad
    //y la seteamos en totalPrice
        $this->totalPrice->setValue(
                $this->product->price->getValue()*
                $this->qty->getValue());
  }
  ...
}
```

Las formas de interacción son tratadas en **??**.

Además, el framework presenta una variación de la arquitectura Modelo-Vista-Controlador tradicional. En el MVC tradicional la vista toma los cambios directamente del modelo a partir de las modificaciones que este sufra. Cuando el

modelo sufre alguna modificación, realiza una notificación; la vista que está "observando" se entera, y se actualiza a sí misma. Sin embargo, este esquema presenta un inconveniente: el controlador no inteviene en esta comunicación que existe entre la vista y el modelo. Esto puede dar origen a problemas de sincronización y acoplamiento. Por ejemplo, los métodos invocados podrían producir errores porque el modelo aun no fue inicializado. Además, la vista podría invocar métodos que no son pertinentes a la vista sobre el modelo, lo que hace que la vista y el modelo queden incorrectamente acoplados. Estos temas son aun más importantes en un ambiente de desarrollo Web, ya que es posible que la persona que diseña la vista y aquella que desarrolla la aplicación ni siquiera sean la misma.

Estos problemas dieron origen a una variación del MVC tradicional, donde las formas de interacción se modifican. Se conoce con el nombre de Modelo-Vista-Presentador, y aparece en varios de otros frameworks de desarrollo[34, 40], en particular en ASP.NET. Allí, toda la interacción y actualizaciones pasa por el controlador y es manipulada por él, como se muestra en la figura 5.3. Bajo esta nueva arquitectura, el controlador ahora se denomina *presentador*, para distinguir entre MVC y MVP. En la figura 5.4 pueden distinguirse las diferencias entre las dos arquitecturas.

Bajo este nuevo esquema, la vista ya no invoca métodos del modelo, sino que espera ser actualizada por el presentador; éste es el que recibe las notificaciones del modelo y ejecuta la funcionalidad necesaria para actualizar la vista. Por lo tanto, los problemas que se producían por la interacción entre la vista y el modelo, ya no aparecen bajo este esquema. Como ya se mencionó, este diseño se hace especialmente deseable para el desarrollo de aplicaciones Web, especialmente al generar la vista de la aplicación; la sección 5.3.7 describe como ésto impacta en el diseño del motor de plantillas.

5.2.3. Casos de uso y transacciones

En el caso de que se necesiten editar objetos del modelo, es necesario lidiar con el botón "Cancelar", con la concurrencia y con la composicionalidad, todo al mismo tiempo. En **Kusala** el problema de la composicionalidad esta resuelto: es posible tener varios componentes activos al mismo tiempo, cada uno de ellos con su propio flujo de control. El problema de concurrencia está resuelto al nivel de la base de datos: se obtiene un error de versionamiento al intentar hacer efectivos los cambios de dos objetos inconsistentes. Esto sólo deja el problema del botón "Cancelar". Cuando el usuario cancela, sea lo que fuese que estuviese haciendo, su intención es descartar las modificaciones hechas a la aplicación por haber usado componentes que modifican los objetos del modelo.

Aquí vale la pena abrir un paréntesis: se menciona concurrencia, pero no parece haber ni hilos ni ni otra estructura de concurrencia relacionada. De hecho, múltiples componentes activos al mismo tiempo pueden ser vistos como diferentes hilos de ejecución. Un componente se considera "activo" si está siendo

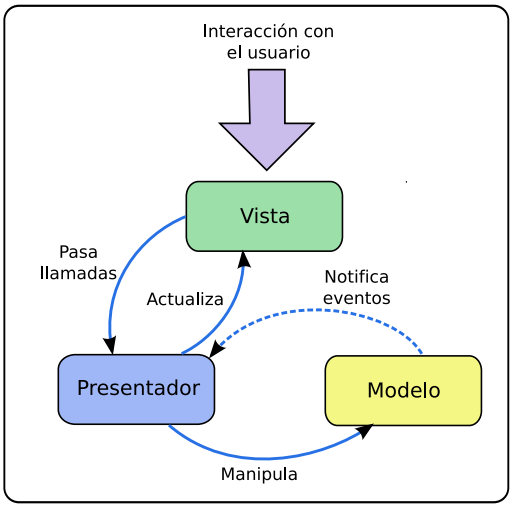

Figura 5.3: Arquitectura Modelo-Vista-Presentador

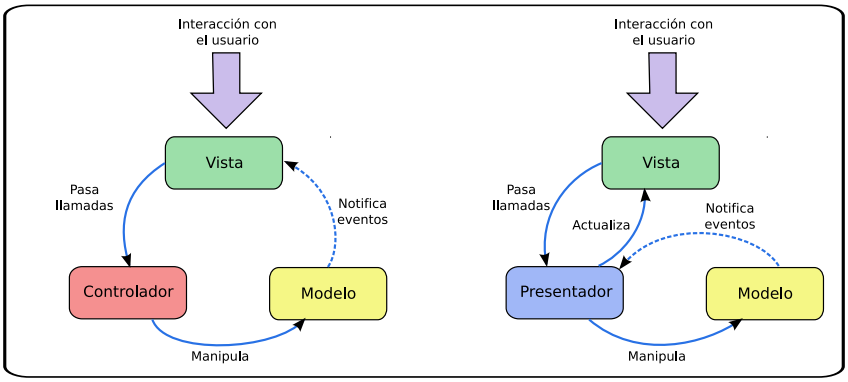

Figura 5.4: Diferencias entre MVC y MVP

mostrado o, sino lo está pero porque ha llamado a otro componente. Por ejemplo, tener dos editores de un mismo objeto del modelo o de objetos del modelo que son colaboradores activos al mismo tiempo provoca claramente un problema de concurrencia.

Una opción es llevar un control de las modificaciones de los objetos del modelo. Ahora bien, ¿de qué objetos? La solución más directa sería llevar un control de *todos*, de forma global. Si bien esto representa un avance respecto a los demás enfoques, tal decisión atenta contra la composicionalidad de los editores.

Si se quisiera editar dos objetos "concurrentemente" mediante dos editores en pantalla, por ejemplo, teniendo una pantalla con un Autor y un Producto. Si el usuario cambia el precio del producto, y luego el nombre del autor, y decide cancelar su cambio de nombre al autor, el precio debería mantenerse cambiado. Por ende, el botón cancelar debe actuar solamente sobre el editor al que pertenece. Además, un cambio en uno de los objetos, no debería poder ser visto desde el otro objeto (ya que se encuentra en otro componente).

Por estos motivos, cada uno de los componentes debe llevar un registro de los cambios que hizo, y no debe compartirlos con los demás componentes hasta que sea el momento oportuno (es decir, hasta que se haga un "commit" de sus cambios, para que el componente padre los pueda manejar).

Una solución posible a estos problemas son las *transacciones en memoria*. Una transacción en memoria es la encargada de registrar los cambios de un objeto. Cada *hilo de edición* debe poseer una. Es decir, es posible componer el hilo de editores si se inicializó la transacción en memoria previa ejecución del hilo.

Al estar implementadas mediante una variable de hilo de componentes (sección 6.9), la transacción en memoria pertenece al hilo de editores, lo que evita la globalidad y conserva la localidad.

De esta manera cambia la perspectiva. Primero, la edición de objetos es globalmente composicional. Segundo, no es necesario hacer copias de los objetos ni copiar datos de la interfaz. Tercero, el código de validación puede pertenecer a los objetos del modelo por lo cual se recupera la simpleza y naturalidad en este aspecto. Cuarto, este esquema no repercute sobre el diseño de la interfaz; tanto un estilo de interfaz optimista y transaccional (aquellas que requieren apretar el botón "Aceptar" para proceder a validar los datos y efectivamente hacer un cambio en el modelo), como aquellas más directas en las cuales los datos son modificados directamente. En este último caso basta hacer efectivos los cambios registrados en la memoria transaccional como respuesta a cada cambio realizado desde la interfaz.

Detalles de implementación

Existe el problema de la edición de un mismo objeto "concurrentemente" por dos editores. En este caso surge un problema de inconsistencia al momento

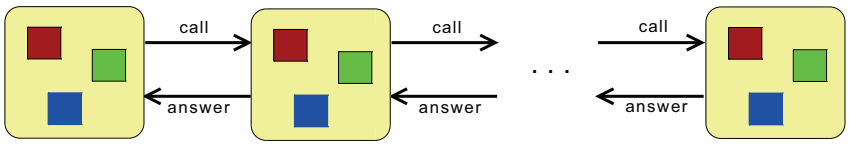

Figura 5.5: Hilo de componentes

	Composicional	Validación Natural	Copia Automática	Transaccional
Edición directa	No	Sí	No	No
Copia datos UI	No	No	No	Sí
Copia objetos del modelo	No	Sí	No	No
Edición transaccional	Sí	Sí	Sí	Sí

Cuadro 5.2: Comparación de soluciones al problema de edición

de hacer efectivos los cambios de las transacciones en memoria. La solución es implementar un esquema de versionamiento. Sólo una de las transacciones bajo la cual se modificó el objeto hace efectivos sus cambios; las demás deben ser vueltas atrás. Esto repercute sobre la interfaz de usuario, la cual deberá informar al usuario y recomenzar la transacción.

5.3 Vista

Para la implementación de la vista, se tuvo como objetivo las siguientes características:

- Mantener una *Separación de Concerns*
- Promover una *Reutilización*

5.3.1. Separación de Concerns

Las guías para desarrollar un motor de plantillas que fuerce la separación de concerns[1] se encuentran en [36]. Sin embargo, si bien existen motores de plantillas que fuerzan la separación, estos constituyen la excepción y no la regla.

[1] Separación de Concerns: Es el proceso de separar un programa en distintas características que se superponen en funcionalidad lo menos posible

Existen cuatro concerns que nos es importante mantener separados en la medida que sea posible:

1. Lógica de la aplicación: Todo lo relacionado con el manejo de datos y las interacciones del usuario.
2. Tipo de vista: Si va a ser HTML, XML, XUL, etc.
3. Diseño de la vista: Cómo se va a ver la aplicación, colores, posición de los datos en la pantalla, etc.
4. Forma de interacción: Si van a haber recargas en cada request, si se va a usar AJAX, COMET (ver 6.8).

Para mantener la lógica de la aplicación separada de la presentación de la aplicación, se decidió, en primer lugar, utilizar un esquema de plantillas. Estas permiten que la capa del controlador no se vea mezclada con detalles de la vista. Pero por otro lado, se decidió que las plantillas no debían tener ninguna forma de manejo de lógica, al contrario de la mayoría de los motores de plantillas existentes, y ser 100 % declarativas (sección 5.3.2). Para esto, es necesario que a la vista se la alimentara con los datos necesarios directamente, como explica el pattern MVP (sección 5.3.3).

Además, el controlador construye con Widgets, los cuales en conjunto con sus manejadores de vista (sección 5.3.4) permiten abstraer el funcionamiento de cada uno del tipo de vista que se está utilizando en ese momento.

Para que el diseño de la vista pueda ser separado completamente, se decidió utilizar como lenguage base XHTML, y extenderlo con algunos tags específicos, para que así los templates los pueda realizar un diseñador con un editor WYSIWYG y que el Skinning (sección 5.3.5) de los componentes sea luego un trabajo sencillo.

Por último, para que las formas de interacción (sección 5.3.6) puedan ser separadas, se introdujo un mecanismo de observer sobre los componentes, que utiliza también el concepto de vistas declarativas a su favor.

Ventajas y desventajas de los motores de plantillas Por un lado, la especificación de la presentación por medio de plantillas supone una mejora respecto a formas de especificación programáticas, en cuanto proveen una mejor separación de los concerns (especialmente de aquellos pertinentes a la vista y al controlador.) La integración directa de diseños de presentación basados en XML, más la especificación declarativa de los datos a mostrar y la independencia en cuanto a su actualización, son las características que permiten ésto. Por otro lado, la Web se ha convertido en una plataforma compleja. Por ejemplo, cada vez más, la programación de la vista de una aplicación Web compleja y de alta interactividad se realiza directamente en el cliente, utilizando lenguajes de programación de scripting (Javascript, Action Script, DSLs basados en XML,

etc), que son distintos a aquellos en los cuales se programó la aplicación. Esto hace que la especificación programática de algunos aspectos de la vista siga siendo valiosa en cuanto provee mayor control para el desarrollo de este tipo de interfaces. Sin embargo, los esfuerzos por el lado del desarrollo de motores de plantillas son valiosos, en cuanto apuntan a lograr una mejora en cuanto a la especificación e integración de aspectos de la presentación; éstos problemas afloran una y otra vez, más allá de si es programada en el cliente o en el servidor y con qué herramientas y lenguajes de programación.

5.3.2. Plantillas declarativas y composicionales

En la generación de vistas, el hecho de hacer las plantillas declarativas limita la expresividad del lenguaje de presentación, que es justamente lo que se necesita hacer para forzar la separación entre la vista y el controlador. A pesar de la separación forzosa y la pérdida de expresión, la mayoría de los problemas de presentación pueden ser resueltos, y mucho más claramente.

Al ser declarativas, las plantillas se adaptan perfectamente al motor de flujo de datos desarrollado; al ser composicionales, resultan ser ortogonales con un diseño de página a partir de un árbol de componentes.

Además, por lo general tanto los frameworks como los motores de plantillas no hacen uso de buenas técnicas de reuso para la generación de vistas. En este framework se utilizaron conceptos habituales para aquellos que programan en objetos: Herencia, y el pattern "Template Method".

Se utilizó la Herencia para elegir el template que debe asignarse a un Componente. Utilizando herencia, cualquier clase A que extiende otra clase B, si A ya tiene un template asociado, y B no lo tiene, B va a mostrarse con ese template.

Por ejemplo, si se tiene la clase `List` y la clase `ProductList`, que extiende la primera, un template de `List` aplicará a ambas clases. Esto permite agregar distintos comportamientos a subclases sin tener que pensar en copiar y duplicar templates. Esto es particularmente útil, ya que en una aplicación estándar, los diseños para mantener una consistencia son muy similares de una pantalla a otra. Entonces, al hacer un estilo definido para las pantallas de listados por ejemplo, o también los formularios de carga de datos, el mismo se va a mantener en toda la aplicación, simplificando el trabajo tanto del programador para implementar, el diseñador para diseñar, y el usuario para aprender y entender las pantallas.

El pattern *Template Method* es utilizado para permitir que, en la especificación de un template, pueda no definirse el template del componente interno, sino solamente especificar dónde el componente hijo debería ir, y diferir el momento de la asignación de ese template al momento de la generación de la vista. Esto permite que distintos componentes se muestren cada uno con su propio template, dentro del mismo componente contenedor. Por ejemplo, si un componente A tiene un subcomponente que puede ser B1 o B2, se puede definir el template

del componente A dejando un "placeholder" para el subcomponente que deba mostrarse, y luego por separado se definen tanto los templates de las clases B1 y B2.

5.3.3. MVP

Existen dos enfoques a la hora de mostrar información en una plantilla. Uno es el método *pull*. Según éste la información del modelo que se desea mostrar se *trae* desde la plantilla invocando métodos sobre el modelo directamente, como muestra la figura 5.6. Este método no es deseable ya que es posible invocar métodos sobre el modelo, el cual podría no estar inicializado y producir algún error; además la invocación de métodos puede resultar en efectos laterales, lo cual es indeseable. Notar que este diseño está intimamente asociado a la arquitectura MVC que vimos antes, donde existe una conexión directa entre la vista y el modelo.

Un enfoque contrario a éste es el denominado método *push*. En este segundo caso la información que puede ser mostrada por la plantilla es previamente provista explícitamente por la aplicación en forma de atributos. La plantilla sólo puede acceder a estos atributos; no es posible invocar ningún método sobre el modelo de la aplicación. Este diseño se corresponde más a la arquictura MVP, donde el controlador manipula la vista para actualizarla, en este caso haciendo disponibles propiedades que serán utilizadas por el motor de plantillas (ver figura 5.7). En el motor de plantillas implementado se utiliza esta segundo método, donde la información a ser mostrada es provista por el controlador de la aplicación en forma de widgets y componentes hijos.

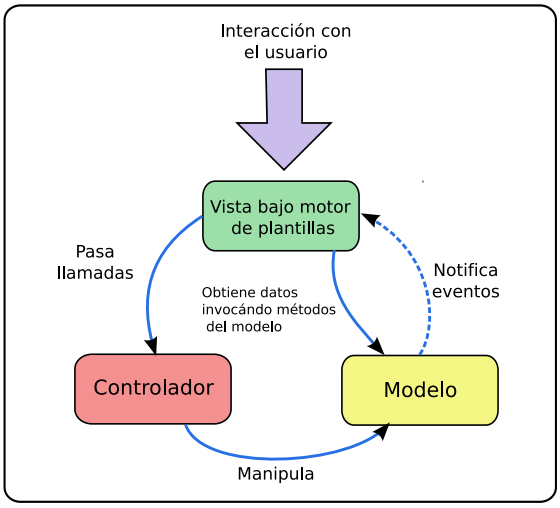

Figura 5.6: Motor de plantillas "pull"

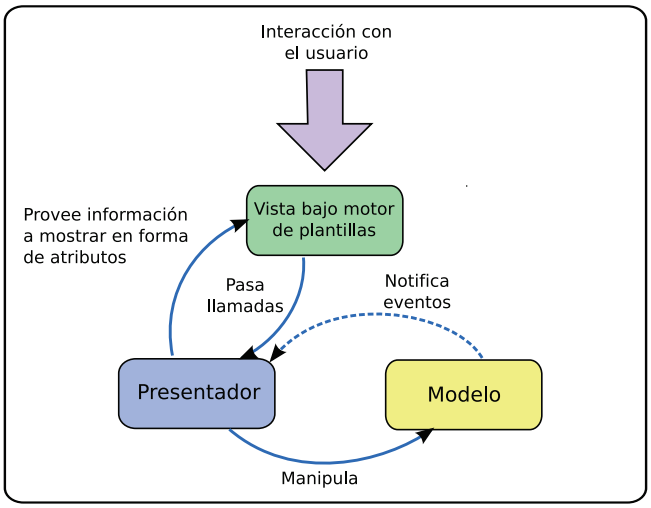

Figura 5.7: Motor de plantillas "push"

	Composicional	Separación con Controlador	Integración de Diseño
Generación Embebida	No	No	Sí
HTML Semántico	Sí	Sí	No
Motor de Plantillas	No	Sí	Sí
Plantillas Declarativas y Composicionales	Sí	Sí	Sí

Cuadro 5.3: Comparación de soluciones al problema de la presentación

Comparación El cuadro 5.3 contiene una comparación de las formas de solución al problema de la implementación de la vista de la aplicación que fueron presentados en los apartados anteriores. La primera columna indica si la solución es composicional. La segunda columna menciona si existe una clara separación entre la vista y el controlador. La tercera, indica si el enfoque soporta una integración casi directa mediante un motor de plantillas.

5.3.4. Widgets y manejadores de vista

La utilización de Widgets, como elementos que definen una forma de interacción con el usuario e implementan esa funcionalidad, permite poder abstraer la programación de un componente, de la interfaz que se va a generar. Por ejemplo, se podría decidir, en lugar de utilizar HTML, utilizar XUL [2] para la interfaz, y el Widget no necesitaría ser modificado.

Para poder introducir esto, es necesario tener *Manejadores de vista*, que son simplemente clases que interactúan con el Widget, y le permiten generar la respuesta apropiada según el tipo de interfaz (sea HTML, XUL, o etc). Teniendo un widget `Checkbox`, que permite una forma simple de seleccionar un booleano, se representaría en HTML como `<input type="checkbox"/>` y en XUL como `<checkbox/>`.

En la utilización de templates declarativos, se fuerza a que no exista ningún tipo de programación sobre la funcionalidad en el template. Esto permite que, en caso de tener que generar los dos tipos de interfaces, al estar toda la programación en el controlador, no haya que replicar código dentro de la vista. Si se quisieran generar interfaces XUL y HTML en Ruby on Rails, por ejemplo, se deberían generar 2 templates, uno para cada tipo de vista, y en cada uno de ellos se estaría forzados a implementar la misma lógica (lo cual lleva a redundancia de código). Casualmente, en Seaside no ocurre lo mismo, ya que al ser tan abstractas las vistas respecto del componente (y a pesar del alto acoplamiento que tienen con el código), si el objeto sobre el que se va a generar la interfaz se cambia, automáticamente se generó una interfaz del nuevo tipo, sin influencia del programador. Vemos que el código del Snippet 5.12 fácilmente podría aceptar un `HTMLInterfaceRenderer` o un `XULInterfaceRenderer`.

Snippet 5.12 Vista en Seaside

```
AuthenticationFrame>>renderContentOn: renderer
  self session isAuthenticated
    ifFalse:
      [[renderer text: 'Login: '.
        renderer textInputOn: #username of: self; break.
        renderer text: 'Password: '.
        renderer passwordInputOn: #password of: self; break.
        renderer submitButtonWithAction: [self authenticate]
                    text: 'Log In']]
    ifTrue: [self render: contents].
```

Por otro lado, se debe notar que la forma de interacción también se encuentra acoplada en el controlador. En este caso, la presencia explícita de un formulario HTML supone una interacción sincrónica; si en Rails se quisiese que el formulario sea enviado de forma asincrónica mediante AJAX, se debería codificar ese comportamiento manual y explícitamente tanto en la vista como en el componente.

5.3.5. Skinning de componentes

Para la realización de estos templates, el lenguaje de definición fue el XML. Esto permite escribir páginas web bien formadas internamente, y agregarle distintos

tags que dan la posibilidad de extender de la manera necesaria.

Además, al estar basado en XML, se puede tomar un HTML de base enviado por un diseñador, y realizar los ajustes necesarios, sin necesidad de realizar ningún tipo de trabajo extra. De una página completa, se pueden extraer fácilmente los templates de los distintos componentes, y armar un template con los mismos. De tal manera, el trabajo del skinning pasa a ser un trabajo mayormente de diseño, y con poca intervención del desarrollador.

Un enfoque alternativo podría haber sido la generación completa del HTML, y luego que el diseño sea provisto completamente por medio de CSS, pero eso complica por un lado la expresividad del diseño (ya que hay cosas que son o más complicadas o imposibles de hacer solamente con CSS) y por otro lado complica el trabajo del diseñador, el cual está restringido a hacer solamente retoques sobre la base ya existente de la aplicación. Este es el enfoque que toma Seaside y que no se adoptó por considerar demasiado restrictivo.

5.3.6. Formas de interacción

En la metáfora de "páginas que se navegan", los frameworks al finalizar su procesamiento generan una página web completa (de manera imperativa) que se envía al browser. Luego, en caso de que quisieran actualizar la información presentada en pantalla sin recargar el sitio completo (mediante AJAX, por ejemplo), no tienen otra posibilidad que incluir el código específico para realizar esas llamadas al servidor.

En **Kusala**, los componentes pueden modificarse independientemente, sin afectar el estado de los demás en la pantalla. Ellos mantienen un estado, y solamente ante un evento externo (el click del usuario, por ejemplo) lo modifican o realizan alguna acción, y al realizarse una modificación del árbol de componentes, se generan actualizaciones correspondientes en la vista. Por esto es que la cantidad de modificaciones que se realizan en la pantalla está controlada, ya que se generan a partir de estas acciones.

Al ser las vistas declarativas, en qué momento se generan las vistas para cada componente del árbol está bajo el control del Framework. Esto permite, mediante un simple "switch" en la configuración de la aplicación (sección 6.10), generar las páginas de manera Stándard o Ajax. También se puede seleccionar que esta actualización sea por COMET (sección 6.8), lo cual permite, además de una mayor velocidad de respuesta, la posibilidad de que sea el servidor quien genere eventos y envíe actualizaciones de información en el momento que se generan (Snippet 5.13).

De esta manera, se consigue no sólo desacoplar la forma de interacción de la programación de la vista, sino que directamente lo deja como una opción para que decida el programador (o el administrador del sitio).

Snippet 5.13 Kusala - Configuración de la aplicación - AJAX

```
//En config.php

//Para que lo muestre en modo "normal"
page_renderer=StandardPageRenderer
//Para que lo muestre en modo "Ajax"
page_renderer=AjaxPageRenderer
//Para que lo muestre en modo "Comet"
page_renderer=CometPageRenderer
```

5.3.7. Detalles de la implementación

Snippet 5.14 Kusala - Template

```
<template class="ListElement">
    <div class="list-element">
        <div class="icono">
            <img src="$templatesdir/images/wrench-5.png" />
        </div>
        <div class="element-info">
            <b>Task:</b> <container id="task" /><br />
            <b>Fecha y hora:</b><container id="dateAndTime"
/>
        </div>
        <div>
            <div class="big-text">
                <container id="select" />
            </div>
            <br />
            <br />
            <div class="small-text">
                Sale: <a id="done">[done]</a>
            </div>
        </div>
    </div>
</template>
```

Todo motor de plantillas debe tener un nivel de acoplamiento con el controlador de acuerdo al diseño de éste. En particular, los motores de plantillas tradicionales permiten determinar la forma en que se presentará *cada página*. Este paradigma queda obsoleto en presencia de componentes para la composición de interfaces. Por lo tanto, bajo un árbol de componentes es necesario un motor de plantillas que permite especificar la presentación *por componente*. Esto se ve reflejado en

la sintáxis para la especificación de plantillas para componentes[2] (Snippet 5.15)

Snippet 5.15 Kusala - Template

```
<template class="[class-name]">
   ...
</template>
```

En este ejemplo, `class-name` es el nombre de la clase de componente sobre el cual será aplicado esta plantilla en el momento en que un componente perteneciente a esa clase sea mostrado en pantalla. El mecanismo de asignación de plantillas va del más específico al más general según la jerarquía de clases de componentes. Esto quiere decir que si tenemos una clase de componente B subclase de una clase de componente A, y no hay una plantilla definida sobre la clase B, pero sí para la clase A, entonces a un componente instancia de la clase B le será asignado la plantilla definida sobre la clase A y por lo tanto se hará su presentación en base a lo definido por ésta.

Aquello a mostrar está definido por el cuerpo de la plantilla, es decir, a partir del XML que se encuentra entre las etiquetas *template*. Este puede contener cualquier HTML correctamente construido mas algunas construcciones que son tratadas de forma especial por el motor de plantillas.

Mostrando componentes hijos Una de ellas son las etiquetas *contenedoras*. Estas se usan para especificar el lugar en el cual los componentes hijos del componente asignado a la plantilla serán visualizados. Una aclaración importante es que las etiquetas contenedoras no especifican la plantilla que será asignada al componente hijo; éstas solo indican el lugar en el cual el componente hijo será visualizado; la plantilla que será asignada al componente hijo es determinada por medio de los mecanismos que aplican a todo componente y que se explican más arriba. Estas tienen dos variantes, aquellas definidas sobre el identificador del componente (Snippet 5.16) y aquellas definidas sobre una clase de componentes (Snippet 5.17).

Snippet 5.16 Kusala - Subtemplate para un hijo

```
<container id="[child-component]" />
```

En el ejemplo *child-component* es la variable de instancia del componente sobre la cual se embebió el componente hijo. Este tipo de construcción indica

[2] Se especifican las partes "variables" encerrandolas entre corchetes ([]) en vez de menores y mayores (<>) como se hace tradicionalmente ya que estos últimos se confunden con la sintaxis XML

Sobre el Template Method: Este es el enfoque que utilizan tanto Rails como Symfony en lo que respecta al `layout`, define una estructura, y espacios donde se va a completar con contenido definido en otro lugar, pero no lo especifica. Lamentablemente, no ocurre lo mismo con el mecanismo de inclusión estándard de subestructuras dentro de un template, que son los partials (los cuales deben ser incluídos por nombre, y no pueden ser "inyectados" como en los layouts).

dónde será ubicado el componente con el dado identificador dentro de la actual plantilla.

La otra forma de estructuras contenedoras son aquellas que no son definidas sobre un componente en particular, sino sobre una *clase* de componentes en particular. En el ejemplo *child-component-class* es la clase de componentes hijos que serán embebidos en el lugar en que aparece esta construcción contendora.

Snippet 5.17 Kusala - Subtemplate para una clase

```
<container class="[child-component-class]" />
```

Respecto del pattern "Template Method", el objetivo con los tags `<container>` es justamente que una vista pueda definirse que un componente debe mostrarse, sin especificar el diseño que se va a utilizar. Como ejemplo, se podría tener una lista de elementos. La visualización default es mostrar el nombre del elemento, pero para un listado de productos la visualización es mostrar tanto nombre como precio (Snippets 5.18 y 5.19).

Snippet 5.18 Kusala - Elementos de una lista

```
class Element extends Component{
   function initialize(){
    $this->addComponent(
      new Label($this->element->toString()), 'name');
   }
}

class ProductElement extends Element{
   function initialize(){
     $this->addComponent(
       new Text($this->element->name), 'name');
     $this->addComponent(
       new Text($this->element->price), 'price');
   }
}
```

Snippet 5.19 Kusala - Template con Template Method

```
<template class="List">
  <h1>Listado</h1>
  <div class="elements">
     <container class="Element"/>
  </div>
</template>
<template class="Element">
   <h3><container id="name"/></h3>
</template>

<template class="ProductElement">
   <h3><container id="name"/></h3>
   <span><container id="price"/></span>
</template>
```

Esto permite, al realizar el template de la lista, no preocuparse por el template de los elementos internos que se van a mostrar, y dejar que el engine de templates elija el mejor template para el elemento a mostrarse. Luego, en la implementación de cada listado, solamente ocuparse de los aspectos que realmente se necesiten adaptar, y dejar el diseño default en el resto de la pantalla.

Por ejemplo, suponiendo que se quieren mostrar los datos de un Producto, entre ellos, los datos de su Autor, como en el Snippet 5.10, esto se hace con el template del Snippet 5.20.

Snippet 5.20 Kusala - Template para un Producto

```
<template class="ProductView">
  <h1 id="name"></h1>
  <!--aquí se mostrará el subcomponente "name"-->
  <p id="price"></p>
  <!--aquí se mostrará el subcomponente "price",
         usando el template especificado-->
  <container id="buy_now"/>
  <!--aquí se mostrará el botón de comprar, con
    un template asignado globalmente-->
  <container class="AuthorView"/>
  <!-- aquí se mostrará todo componente de clase
    AuthorView -->
</template>
```

Colecciones Para listar un conjunto de componentes, basta ingresar una etiqueta contenedora de este tipo en el lugar que se desee que los componentes aparezcan; todos los componentes que coincidan con la clase especificada mediante el atributo *class* serán embebidas allí, una detrás de otra. Las limitaciones de este enfoque saltan a la vista ya que para listar un conjunto de componentes es necesario que todos ellos cumplan con determinado tipo. Es posible especificar *Component* en el atributo *class*, pero esto supone una pérdida de control sobre aquellos componentes que se quiere listar y aquellos que no (en este caso, como todos los componentes son de clase Component, todos los componentes hijos serán listados, sin excepción). En la práctica, esta limitación no es un problema mayor ya que en general las colecciones de componentes suelen pertenecer a una misma clase; si no es así, una pequeña refactorización del código permite obtener resultados más que aceptables. Por último, notar la naturaleza declarativa de la solución; si bien se está iterando (se muestra una lista de componentes), en ningún momento se utilizan contadores ni estructuras de control propias de lenguajes imperativos como *while* o *for-each*. En cuanto a la cantidad de elementos a mostrar, ésto queda a cargo de los componentes (es decir, de la capa del controlador); el contador está representado por la cantidad de componentes hijos. A partir de esta estructura puede verse como la semántica declarativa fuerza una estricta separación de la presentación de aquello que corresponde al controlador.

Snippet 5.21 Kusala - Template para un Autor y sus productos

```
<template class="AuthorView">
  <h1 id="name"></h1>
  <!--aquí se mostrará el subcomponente "name"-->
  <div class="products_list">
    <!-- notar que products_list es la clase de CSS, y
       no tiene ningun efecto sobre los templates -->
    <container class="ProductElementView"/>
    <!-- aquí se mostrará todo componente de clase
      ProductElementView, lo cual muestra una
      colección de elementos -->
  </div>
</template>
```

Plantillas locales Otra de las construcciones tratadas de forma especial por el motor de plantillas son aquellas etiquetas *template* que se encuentran embebidas en la plantilla (es decir, dentro de una etiqueta *template* de más afuera). El propósito de este tipo de etiquetas es doble. Por un lado, representan la forma en la cual la asignación de plantillas puede ser variada respecto al mecanismo general de asignación. En particular, un componente del tipo de aquel especificado en una de éstas etiquetas será presentado por medio del cuerpo de ésta,

evitando el mecanismo de asignación general y dando cierta sensibilidad al contexto (en este caso el contexto es la plantilla sobre la cual se está presentando el componente padre, naturalmente). Por otro lado, indican el lugar dónde los componentes de determinado tipo serán presentados. El componente de producto podría definir cómo se ve el Autor dentro de su vista, como en el Snippet 5.22.

Snippet 5.22 Kusala - Template para un Producto y Autor

```
<template class="ProductView">
  <h1 id="name"></h1>
  <!--aquí se mostrará el subcomponente "name"-->
  <p id="price"></p>
  <!--aquí se mostrará el subcomponente "price",
        usando el template especificado-->
  <container id="buy_now"/>
  <!--aquí se mostrará el botón de comprar, con
    un template asignado globalmente-->
  <template class="AuthorView">
  <!-- aquí se mostrará todo componente de clase
    AuthorView -->
    <h2 id="name"></h2>
    <!--aquí se mostrará el subcomponente "name"
        que corresponde a AuthorView -->
  </template>
</template>
```

Asignación de plantillas La asignación de plantillas se realiza de forma dinámica y bajo un algoritmo que se encarga de ello. En los motores de plantillas tradicionales, ésto queda a cargo al desarrollador, el cual debe asignar las plantillas de forma manual.

Además, el motor de plantillas desarrollado sigue las reglas para una estricta separación de los aspectos de la vista de aquellos del controlador y del modelo tal como aparecen en [36].

Las reglas son las siguientes:

1. La vista no puede modificar el modelo de la aplicación invocando métodos sobre éste (con potenciales efectos laterales).
2. La vista no puede realizar computaciones, esto debe ser llevado a cabo en el modelo o en el controlador de la aplicación.
3. La vista no puede comparar diferentes atributos, aunque puede testear la presencia o ausencia de ellos.

4. La vista no puede asumir nada acerca de los tipos de datos.
5. Los datos del modelo no deben contener información acerca de la distribución de los elementos en pantalla ni que tengan que ver con el mostrado de elementos.

Este motor de plantillas cumple todas ellas y por lo tanto fuerza una estricta separación entre la vista y los demás aspectos de la aplicación.

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp

Greenspun's Thenth Rule of Programming

6

PHP como lenguaje de implementación

Para la implementación del framework, PHP[39] fue la plataforma elegida. Esto fue así por su uso extendido para el desarrollo de aplicaciones web, además de las ventajas que posee en cuanto a disponibilidad de hostings y facilidad de deployment.

Sin embargo, gradualmente se hicieron evidentes sus limitaciones y pronto fue necesario implementar dispatch múltiple[30], referencias débiles[25], mixins[21] y versiones limitadas de macros y lenguajes específicos de dominio[45] para proveer sintaxis para éstas y otras abstracciones, como la activación de componentes según contexto[28] y lenguajes de consultas orientados a objetos[14].

Estas implementaciones, si bien son usables, están lejos de tener la madurez alcanzada en otras plataformas. De ahí, la décima regla de programación de Greenspun: *"Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp[23]".*

A continuación se mencionan algunas de las características se agregaron a PHP. Cabe alcarar que las mismas fueron agregadas sobre PHP, es decir, construídas sobre el mismo, y no son cambios hechos al compilador o al intérprete, por lo que son utilizables sobre cualquier servidor que interprete PHP.

6.1 Multiple Dispatching

Una característica implementada, fue el múltiple dispatching de funciones. Dado que muchas veces el método a utilizar depende de más de una clase, las funciones de múltiple dispatching ayudan a resolver este problema. Para implementarla, se utilizó el mecanismo de macros de 6.5.

Primero se define la función, con los parametros a utilizar tipados, y luego se hace el llamado, que utiliza los tipos de los parámetros para resolver el método a utilizar. En el ejemplo del Snippet 6.1, se puede ver de qué manera la clase del componente del elemento de la lista puede ser elegido a partir de la clase del elemento pasado como parámetro. Es decir, el `Product` va a obtener un `ProductElement` cuando se muestre en un listado. Esto permite no tener que subclasificar la clase List para que devuelva esta clase, ni tener que asociar a la clase `Product` el comportamiento de saber con que Componente mostrarse.

Snippet 6.1 Declaración de Múltiple Dispatching

```
#@defmdf &getListElement(&$object:PersistentObject)
{                $le =& new ListElement($object);
                return $le;
}
@#

#@defmdf &getListElement(&$object:Product)
{                $le =& new ProductElement($object);
                return $le;
}
@#
```

Además, las funciones de múltiple dispatching pueden ser utilizadas pasando el contexto (la rama de componentes dentro de la que se hace el llamado) de aplicación, para de esta manera poder responder de manera diferente a un evento, dependiendo del contexto. Esta variación recibió el nombre de "Context Dispatching". Esto permite, por ejemplo, que un producto en un listado default se muestre de una manera, y de otra distinta en, por ejemplo, el listado de productos de un Autor (Snippet 6.2). Luego, el llamado se hace de manera similar al llamado de cualquier función (Snippet 6.3).

Cabe aclarar que la selección del método se hace en tiempo de ejecución, a partir del tipo real de los objetos que fueron pasados como parámetros. Esto es así, a diferencia del Overloading de métodos de Java, en el cual se pueden especificar distintos métodos dependiendo del tipo de los parámetros, pero esto es resuelto luego en tiempo de compilación.

De esta manera, se puede implementar, por ejemplo, un administrador de objetos Autor con la clase default ObjectsAdmin. Si se quisiera, por ejemplo, cambiar

Snippet 6.2 Declaración de Context Dispatching

```
#@defmdf &getListElement[List](&$object:Product)
{                $le =& new ListElement($object);
                return $le;
}
@#

#@defmdf &getListElement[AuthorView->List](&$object:Product)
{                $le =& new ProductAuthorElement($object);
                return $le;
}
@#
```

Snippet 6.3 Llamado de Context Dispatching

```
class List extends Component{
  function getListElement($element){
    return #@callmdf getListElement[$this]($element)@#;
  }
}
```

el elemento de la lista para mostrar el autor, simplemente se puede agregar un Override por Context Dispatching correspondiente, y no se necesitaría modificar otra cosa (Snippet 6.4).

En Rails o en Symfony, la manera de realizar este tipo de acciones es pasando como una variable el nombre del template a incluir, a través de todos los objetos intermedios, y modificar la instanciación del template para que se utilice el correspondiente. Un trabajo similar hay que realizar con los Componentes, pasando el nombre de la clase de la cual se van a instanciar los Componentes.

Snippet 6.4 Overriding por Context Dispatching

```
//Llamamos al Objects Admin
$this->call (new ObjectsAdmin (#@select Author@#));

//Creamos la clase override del elemento de la lista

class AuthorElement extends Element{
  public function initialize(){
    parent::initialize();
    $this->addComponent(
      new Image($this->object->avatar), 'avatar');
  }
}

//Definimos el Contect Dispatching.

#@defmdf &getListElement(&$object:Author)
{
  $le =& new AuthorElement($object);
  return $le;
}
@#
```

6.2 Referencias Débiles

Cuando un objeto `$x` queda escuchando un evento de un `PWBObject $y`, este último necesita guardar una referencia al primero. En caso de que, por el flujo de la aplicación, `$x` deje de ser necesario, el mecanismo de garbage collection de PHP no puede descartarlo, porque `$y` lo conserva referenciado, aunque no lo necesite realmente.

Por esto fue implementado un mecanismo de referencias débiles, en donde `$y` se queda con una referencia de `$x`, pero de la cual el garbage collector no se entera, permitiendo borrar a `$x` en caso de que sea necesario.

Esta implementación fue hecha mediante la clase `WeakReference`, que mantiene estas relaciones débiles. El funcionamiento es como sigue:

En PHP, en cada request, las variables que están relacionadas con la variable global "session"son persistidas. Toda variable que no lo esté referenciada desde session, queda por lo tanto eliminada.

Lo que fue implementado, fue que la clase WeakReference, en lugar de mantener una referencia directa al objeto, lo que mantiene es una referencia indirecta a través de una variable global `references`, que no es persistida junto con la

sesión. Entonces, al crear una `WeakReference`, lo que se hace es guardar la ubicación del otro objeto dentro de esa variable global. De esta manera, puede conocer el objeto sin mantener una variable de instancia.

Luego, en cada request, cada objeto observable, en su evento `#wakeup` (que es llamado por php para cada objeto guardado en la sesión) se vuelve a registrar dentro de este arreglo. De esta manera, solamente van a quedar registrados en el arreglo aquellos objetos que hayan sido efectivamente guardados en la sesión, de manera directa.

Snippet 6.5 Uso de WeakReferences

```
//Creamos una referencia debil a $referred
$referrer->ref =  new WeakReference($referred);
$referrer->ref->getTarget() // Este es $referred

//Luego de un paso del Garbage Collector

$referrer->ref->getTarget() // null
```

6.3 PHPCC

Para crear el lenguaje OQL, se debió crear un Compiler Compiler para PHP (algunos ejemplos conocidos sobre C son Bison[5], yacc[6]). De esta manera se pudo extender el framework con múltiples DSLs (Domain-Specific Languages - Lenguages específicos de Dominio).

El funcionamiento de PHPCC es el siguiente: Por un lado, se define un DSL (el OQL en este caso). Luego, a este OQL se le agregan `PointCuts`, que son los distintos manejadores de los elementos del árbol generado. Por último, el resultado es devuelto como un String. El DSL se define como en el Snippet 5.6.

De esta manera, para la consulta del Snippet 6.6 el resultado obtenido es el del Snippet 6.7.

Snippet 6.6 Consulta en OQL

```
select p:Product where p.author is $author
```

Snippet 6.7 Resultado de consulta en OQL

```
CompositeReport::fromArray(
        array('subq'=>new Report(
                array('class'=>'Product','target'=>'p')),
        'exp'=>new Condition(array(
                'exp1'=>new ObjectPathExpression('p.author'),
                'operation'=>'=',
                'exp2'=>new ObjectExpression($author))),))
```

El llamado que se hace al compiler compiler queda como en el Snippet 6.8.

El compiler compiler PHPCC es tiene un parser top down recursivo a izquierda, con control de loops para evitar una recursión a izquierda infinita.

Snippet 6.8 Invocación de PHPCC

```
class OQLCompiler {
  function fromQuery($query) {
    $oqlGrammar = PHPCC :: createGrammar($oqlDescription);
    $oqlGrammar->setPointCuts(array (
      'condition' =>
        new FunctionObject($this,'parseCondition'),
      'expression' =>
        new FunctionObject($this, 'parseExpression'),
      'oql' =>
        new FunctionObject($this, 'parseOql'),
      'valueorfunction' =>
        new FunctionObject($this, 'parsevalueorfunction'),
      'plainsql' =>
        new FunctionObject($this, 'parseplainsql')));
    return $oqlGrammar->compile($query);
  }
  ...
}
```

6.4 Chequeo de Tipos

Un tiempo grande del desarrollo se destina a la corrección de errores. Una modificación de un módulo de programa puede afectar a otros módulos que lo utilicen, provocando un error difícil de detectar, y por ende costoso de solucionar. Una forma de detectar estos errores es mediante el chequeo de tipos y las aserciones, que permiten un control más estricto sobre los datos que se comunican entre módulos.

La ejecución de estas validaciones, si bien son útiles para el desarrollo, consumen un tiempo innecesario cuando el programa es instalado en producción. Una forma de recortar estos tiempos es eliminar del código final todos los chequeos, pero muchos pueden pasar sin ser detectados y además podrían llegar a ser útiles ante una modificación posterior del sistema.

Por esto mismo, en **Kusala** se utilizan las macros `#@typecheck@#` y `#@check@#` para insertar validaciones desactivables en el código.

Por ejemplo, escribiendo `#@typecheck $author: Author@#` se chequea que la variable `$author` sea un `Author`. También se ppuede hacer cualquier tipo de chequeo, como por ejemplo que el nombre del autor no sea nulo (Snippet 6.9). Esto avisará al programador por problemas durante el desarrollo al intentar inicializar erróneamente un `AuthorView` con algo que no sea un `Author`.

Habilitando en el config.php (sección 6.10) `compile=typecheck`, se pueden utilizar estos chequeos, y deshabilitandolo se obtiene un código listo para salir a producción.

Snippet 6.9 Uso de Typechecking y assertions

```
class AuthorView extends Component{
  function AuthorView($author){
    #@typecheck $author: Author@#
    //Chequeamos que la variable $author
    //tenga un dato del tipo correcto
    #@check $author->name->getValue()!=null@#
    //Chequeamos que el nombre del
    //autor no sea nulo.
    $this->author=&$author;
    parent::Component();
  }
  ...
}

//Con los checks habilitados:

class AuthorView extends Component{
  function AuthorView($author){
    #@typecheck $author: Author@#
    if (!hasType($arg,'$type')) {
      print_backtrace(
        'Type error. Argument: $author. Type: '
        . getTypeOf($author) . '. Expected: Author');
        }
    assert('$author->name->getValue()!=null');
    $this->author=&$author;
    parent::Component();
  }
  ...
}

//Con los checks deshabilitados:

class AuthorView extends Component{
  function AuthorView($author){
    $this->author=&$author;
    parent::Component();
  }
  ...
}
```

6.5 Macros

Tanto el OQL (sección 6.3) como `check` y `typecheck` son *macros*: fragmentos de código que se ejecutan una sola vez, en tiempo de compilación (ver 6.7), y reemplazan su texto en el código de la aplicación final.

Un programador puede agregar sus propias macros declarando una función simple de PHP `mi_macro` y luego llamándola con `#@mi_macro` *parámetros* `@#`. Como se ve en lenguajes como C, las macros pueden llegar a tener un rol muy importante en un proyecto ya que son otra forma de modularización.

En el código fuente de un archivo, el llamado a una macro queda luego reemplazado por el resultado de ejecutar la función correspondiente, como se puede ver en el Snippet 6.9

La definición de una macro es sencilla, para definir por ejemplo la macro `check`, se define una función check que devuelve el string que va a ser parte del código (Snippet 6.10).

Snippet 6.10 Definición de una macro

```
function check($text) {
        return optionalCompile('assertions',
                "assert('" . addslashes($text) . "');\n");
}
```

A diferencia de C, en donde las macros tienen su propio lenguaje para definirse, en este caso las macros se definen directamente en el lenguaje objetivo (es decir, PHP).

Una dificultad que tiene este esquema de macros, es que el código llega a la función como un string, y la funcion misma debe encargarse de procesar los distintos datos. Esto se ve mas claramente en un ejemplo como el de la macro defmdf, que crea una función de multiple dispatching (Snippet 6.11).

Snippet 6.11 Definición de la macro defmdf

```
function defmdf($text) {
        preg_match('/(&?[[:alpha:]]*)'
            .'[\s\t]*(?:\[(.*)\])?'
            .'[\s\t]*\((.*?)\)'
            .'[\s\t]*\{(.*)\}/s', $text, $matches);
        $name = $matches[1];
        $context = $matches[2];
        $params = $matches[3];
        $body = $matches[4];

        $rules = array();
        if ($context != '') {
                $cs = explode('<-',$context);
                foreach (array_keys($cs) as $i) {
                        $cs[$i] = trim($cs[$i]);
                }
                $rules['in'] = $cs;
        }

        $ps = explode(',', $params);
        $pss = array();
        foreach($ps as $p) {
                $pp = explode(':', $p);
                $arg = trim($pp[0]);
                $type = trim(str_replace(
                    '<','__tp_',
                    str_replace('>','_tp__',$pp[1])));
                $pss[$arg] = $type;
        }
        $rules['with'] = $pss;
        $rules['do'] = $body;
        return compile_md_function($name, array($rules));
}
```

6.6 Mixins

Un `Mixin` es una forma de agrupar funcionalidad y agregársela a múltiples clases de objetos sin que estas clases estén conectadas por subclasificación. Esun buen trade-off entre herencia simple y múltiple.

En el Snippet 6.12 se puede ver la declaración de un mixin `ValueHolder`, que provee los métodos `#getValue` y `#setValue`.

Snippet 6.12 Definición de un mixin

```
#@mixin ValueHolder {
  var $value;
  function getValue(){
    return $this->value;
  }
  function setValue($value){
    $this->value = $value;
  }
}@#

class Contador{
  #@use_mixin ValueHolder@#
  function increment(){
    $this->setValue($this->getValue()+1);
  }
}

class Direccion extends PersistentObject{
  #@use_mixin ValueHolder@#
  function initialize(){
    $this->addField(new TextField(array('fieldName'=>'value')));
  }
}
```

De esta manera se puede implementar una sola vez el comportamiento de `ValueHolder` y utilizarlo en 2 clases distintas (`Direccion` y `Contador`), sin necesidad de relacionarlos por herencia.

Además, los mixins fueron incluídos en el chequeo de tipos. Si existe un chequeo `#@typecheck $v: ValueHolder@#`, tanto un objeto de clase `Direccion` como uno de clase `Contador` cumplen con la condición.

Al estar dentro del chequeo de tipos, los Mixins también toman parte en la selección de un template para un objeto. Si el objeto no tiene template para su clase, pero sí para su Mixin, entonces el template que se le asigna es del del Mixin.

6.7 Compilación

Para la implementación de las Macros 6.5, se realizó una compilación de PHP a PHP. Cada archivo PHP con Macros es primero procesado por un compilador, que procesa la macro, reemplaza su resultado dentro del documento original, y genera un nuevo archivo PHP ya sin macros y 100 % interpretable por el intérprete PHP estándard.

Esta compilación, además, es mínimamente costosa, por lo que se implementaron distintos mecanismos de cacheo de esta compilación, los cuales se pueden elegir desde el mismo archivo de configuración 6.10. Uno de los mecanismos es el directo, que guarda el archivo compilado para cada archivo fuente, y luego en cada request revisa si la fecha de modificación del archivo original es posterior al del compilado, para entonces recompilarlo. Otra forma de compilación es en un sólo archivo, la cual incluye cada uno de los archivos existentes en un sólo archivo compilado. Esto es más eficiente en cuanto a cargas de archivo en cada request, ya que requiere un sólo pedido a disco por el archivo. Por último, se creó un tipo de compilación "óptimo", que intenta incluir solamente aquellos archivos que incluyen clases utilizadas en la aplicación.

Este mecanismo de compilación permitió utilizar las macros extensamente, ya que el código generado es cacheado, por lo que esta generación de código puede ser tan costosa como sea necesario y no afectar los tiempos de ejecución en producción.

6.8 COMET

Uno de los mecanismos de comunicación implementados, además del Standard (de un rendereo de página por request) y el mecanismo de AJAX, fue el de COMET.

Para el mismo, la forma de rendereo en el browser es similar a AJAX (se recibien los comandos por javascript y se ejecutan en el browser). La diferencia es que la página, al terminar de cargar, abre una conexión con el servidor que queda abierta por un largo tiempo, y se cierra cuando desde el browser no se establece una conexión por más de 5 minutos. Dentro de esta conexión, la aplicación se mantiene viva, lo cual para cada request reduce tiempos de carga tanto de archivos PHP como de la sesión de usuario. El browser, al hacer un request, se comunica con un archivo php especial, que lo que hace es enviar un mensaje por envío de mensajes del S.O. (por esto es, que COMET solamente funciona para Unix).

Esto permite también, que si hay 2 instancias de la aplicación funcionando, una pueda enviarle mensajes a la otra, y el browser de cada usuario sea actualizado con información generada por otra sesión de la aplicación.

6.9 Variables Dinámicas

PHP tiene dos mecanismos para la comunicación de datos entre módulos: Variables globales, y pasaje de mensajes. Las variables globales tienen alcance global, pero su extensión es ilimitada. Por eso son globales: pueden ser accedidas desde cualquier lugar y preservan su valor. El pasaje de mensajes se basa en pasar, como parámetros en los llamados a métodos, los datos en variables.

Para comunicar datos entre un componente padre con sus componentes hijos, ninguna de las opciones anteriores resulta acertada, por lo que se utilizo otro enfoque. Es un concepto generalmente no visto en los lenguajes de programación de la industria. Este es el de las variables dinámicas[26]. Son variables que tienen tanto un alcance global y una extensión delimitada. No obstante su escasa popularidad, son una característica vital para el desarrollo de sistemas extensibles[43]. El cuadro 6.1 muestra las diferencias de alcance y extensión entre los tipos de variables.

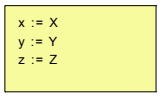

Figura 6.1: Representación gráfica de un registro de activación

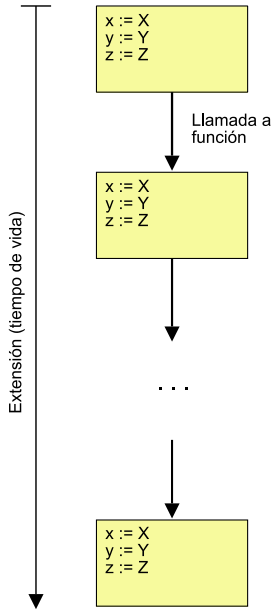

Figura 6.2: Extensión de variables de lenguajes de programación

Su funcionamiento es relativamente simple: Inicialmente se declara una variable,

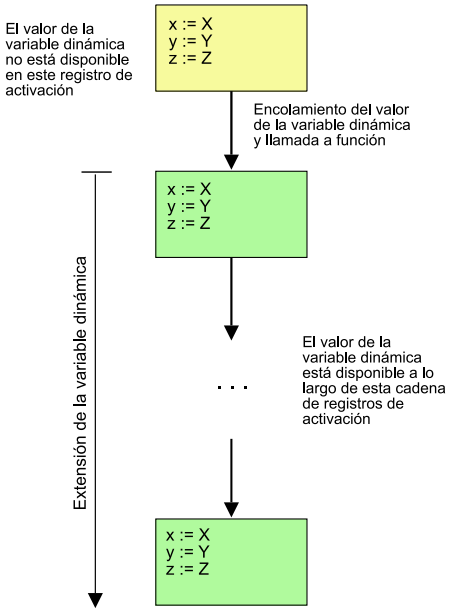

Figura 6.3: Extensión de variables dinámicas

Tipo de Variable	Alcance	Extensión
Léxica	Local	Delimitada
Global	Global	Ilimitada
Dinámica	Global	Delimitada

Cuadro 6.1: Tipos de variables

con un valor inicial. Al intentar acceder a esa variable, se obtendrá ese valor. Luego, al entrar en otro "contexto", se puede definir un nuevo valor para la variable. Cualquier acceso a la misma dentro de ese contexto obtendrá el nuevo valor, y una vez que volvamos a salir, el valor volverá a ser el inicial. Si se accede sucesivamente a varios contextos que modifican el valor, entonces los valores se apilarán al entrar, y se desapilarán al salir, dando siempre el valor del tope de la pila cuando se lo solicite. Esta lista (ordenada) de contextos, ya sea los que modifican el valor de la variable, como los que no, son llamados *cadena dinámica*. Para obtener el valor de la variable, hay que recorrer la cadena dinámica hasta encontrar su valor como se muestra en la figura 6.4.

En el framework se utilizó como cadena dinámica el *hilo de componentes*. Un *hilo de componentes* es el conjunto de componentes que pertencen a una misma cadena formada por la invocación de *call* y *answer* (Figura 5.5), junto a los subcomponentes de cada uno. Notar que el equivalente de la cadena dinámica es la cadena formada por las llamadas y respuestas entre los componentes del hilo más las relaciones entre los subcomponentes y los componentes padres, como se muestra en la figura 6.5.

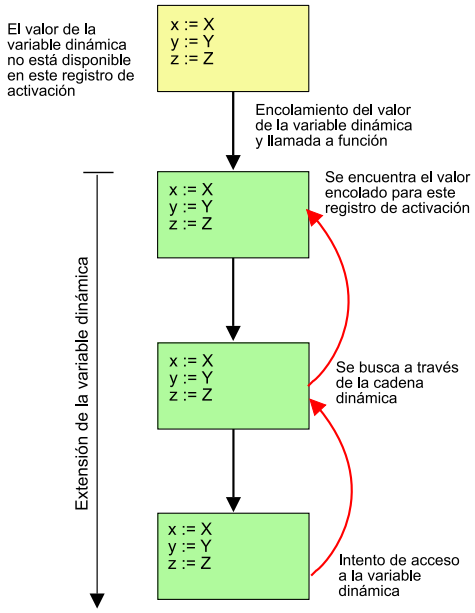

Figura 6.4: Búsqueda del valor de una variable dinámica

En **Kusala** las variables dinámicas se utilizan como mecanismo de comunicación de un componente superior con un subcomponente. Un claro ejemplo de esto son las transacciones en memoria. La aplicación posee una `DBSession` a nivel de aplicación, por lo que cualquier componente intentando acceder a la sesión de base de datos, al solicitar dinámicamente esta variable, obtendrá este objeto. Si

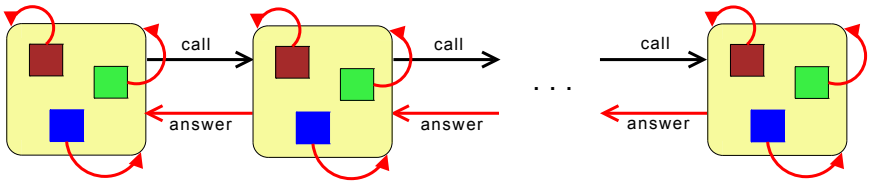

Figura 6.5: Búsqueda una variable de hilo de componente

un subcomponente inicia una transacción, la variable dinámica tendrá entonces una `MemoryTransaction` y será eso lo que cada hijo de este subcomponente obtenga al solicitar esa variable.

6.10 Configuración

Para la configuración de la aplicación se decidio mantener en un sólo lugar la configuración completa del sistema. Además, el sistema tiene una configuración global (en donde se especifica por ejemplo cual es la subclase de `Application` que será el punto de partida para el sistema), y por otro lado una configuración específica para los entornos de desarrollo de cada uno de los desarrolladores, el entorno de integración, el de staging, el de producción, etc.

Se decidió que el archivo de configuración debía ser uno sólo, con la información completa de cada uno de los usuarios, para de esta manera poder ser versionado fácilmente y sin miedo a romper la configuración de los demás usuarios. Luego, existe un archivo `serverconfig` que contiene simplemente el nombre de la configuración que se debe usar en un entorno en particular. De esta manera, evitando versionar este último archivo, se pueden hacer modificaciones a la configuración completa sin problemas de conflictos.

En la configuración, se mantienen variables globales que van a afectar el comportamiento de la aplicación. Un ejemplo de esto son los parámetros de acceso a la base de datos (motor, usuario, nombre de la base, etc). También se encuentran los parámetros de compilación, que afectan al modo de compilación y la generación de macros, y otras configuraciones, como la forma de rendereo del sistema.

Snippet 6.13 Ejemplo de configuración del sistema

```
[global]
app_class=ExampleApplication
db_driver=MySQLDriver
tables_type=InnoDB
sessionHandler=PHP
error_reporting="E_ERROR | E_WARNING
   | E_PARSE |E_COMPILE_ERROR"
page_renderer=AjaxPageRenderer
compile=recursive
modules="Core,Application,Model,
  YATTAA,Instances,View,
  database,DefaultCMS"

[server]
pwb_url=../pwb/
basename=example-prod
baseuser=example-prod
basepass=*******
serverhost=localhost
baseprefix=
db_driver=MySQLDriver
compile=recursive


[development]
error_handler=disabled
page_renderer=StandardPageRenderer
;page_renderer=AjaxPageRenderer
serverhost=localhost
basename=example-dev
baseuser=example-dev
basepass=*******
pwb_url=pwb/
db_driver=MySQLDriver
compile=recursive
;compile=
;compile=sql_dml_echo,sql_query_echo
```

The only worthwhile achievements of man are those which are socially useful.

Alfred Adler

7

Caso de éxito: Intranet

En esta sección se vera la utilización de **Kusala** en la construcción de una aplicación de intranet de una empresa. Si bien el sistema es real y está en funcionamiento, se reserva el nombre de la empresa y fueron cambiados detalles a la funcionalidad para preservar la privacidad de la misma.

El objetivo es demostrar que no sólo fue posible generar un **Kusala** con estas características, sino que fue usado para resolver un problema real, y además, cómo las características del **Kusala** proveyeron múltiples beneficios.

7.1 Modelo

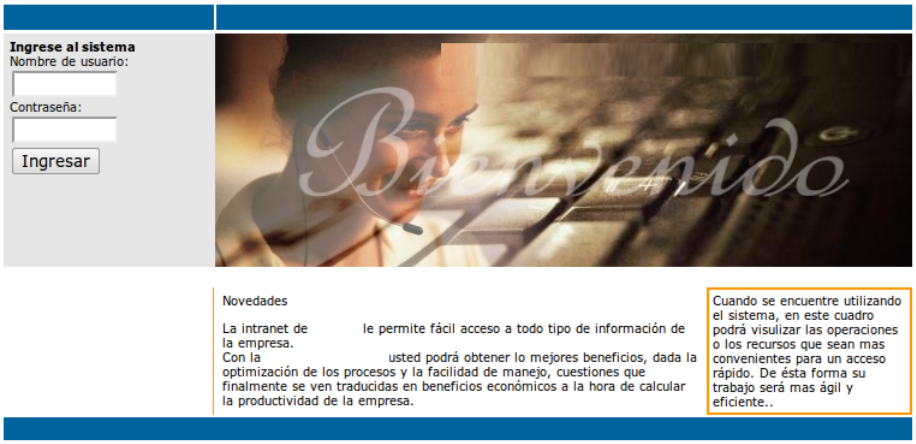

Figura 7.1: Pantalla de Login

El sistema se compone de un conjunto de módulos separados, que se encuentran todos dentro de la aplicación. Uno de estos módulos es una Agenda, que tiene el modelo de la imagen 7.2.

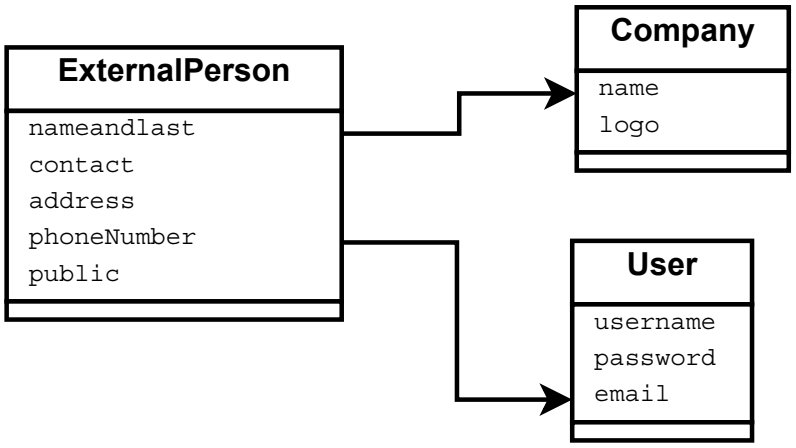

Figura 7.2: UML del modelo de Agenda

Para definir la clase `ExternalPerson`, que es el corazón de este módulo, se hace como en el Snippet 7.1. Se puede ver que se debió incluir, dentro de la definición de cada variable de instancia del objeto, la clase que va a representar cada tipo de dato, para que pueda de esta manera generar el dato de la manera correspondiente. Se puede ver el comportamiento especial de dos campos: `IndexField`,

que tiene una referencia a otro objeto (el `owner` y la `company`, en este caso), y la colección de `viewers`, que se implementa con un `CollectionField`. Este último necesita de un `IndexField` en el otro objeto, y hace referencia a ese campo mediante la variable `reversefield`.

Snippet 7.1 Definición de la clase ExternalPerson

```
class ExternalPerson extends PersistentObject {
  function initialize() {
    $this->addField(new TextField('nameandlast'));
    $this->addField(new TextArea('contact'));
    $this->addField(new TextField('address'));
    $this->addField(new TextField('phoneNumber'));
    $this->addField(new IndexField('owner', array (
      'type' => 'User'
    )));
    $this->addField(new IndexField('company', array (
      'type' => 'Company',
    )));
    $this->addField(new BoolField('public', array(
      "default"=>false
    )));
    $this->addField(new CollectionField(
      array('fieldName' => 'viewers',
      'reverseField' => 'person',
      'type' => 'ExternalPersonDistributionList'
    )));

  }
  function printString() {
    $nl = $this->nameandlast->getValue();
    return $nl!=""?$nl:"<<Sin Nombre>>";
  }
  function validate() {
    $this->checkNotEmptyField('nameandlast',
      'Ingrese un nombre');
  }
}
```

Validación Para la validación del modelo, en la misma clase de la instancia se incluye un método `#validate`, que realiza la validación de los datos de la instancia. Este método se apoya en varios métodos auxiliares definidos en la clase `PersistentObject`.

Consultas Para obtener los `ExternalPerson` que son visibles por un usuario, se utiliza la función del Snippet 7.2. Ahí se puede apreciar el uso de OQL (ver 5.1.3) para realizar la consulta.

Snippet 7.2 ExternalPerson visibles por un User

```
function &GetContacts() {
  $user=User::logged();
  return #@select c:ExternalPerson
    where (c.owner is $user
      or c.public=1
      or exists (c.viewers as v
        where v.viewer is $user))@#
  ;
}
```

Archivos Una clase especial de objeto del modelo, es la clase `File`. Esta clase forma parte de **Kusala**, y contiene el comportamiento necesario para poder agregar archivos al modelo. Esta clase, al extender `PersistentObject`, se puede conectar a otros objetos del modelo mediante un `IndexField`. En este sistema uno de los usos fue la imagen del Logo para una Empresa (Snippet 7.3).

Snippet 7.3 Uso de File

```
class Company extends PersistentObject {
  function initialize() {
    $this->addField(new TextField('name', array (
      'is_index' => true)));
    $this->addField(new IndexField(array('fieldName'=>'image',
'type' => 'File')));
    ...
  }
  ...
}
```

7.2 Controlador

Context Dispatching En la pantalla de la agenda se ve un componente que se repite en todo el sistema, que es el componente `ObjectsNavigator`, que tiene formularios de búsqueda y opciones de ordenamiento, y varios `ObjectElement`, que son los elementos de la lista. Estos componentes vienen con **Kusala**, por lo que simplemente se puede usarlos y generan automáticamente las pantallas de administración en el sistema. Estos componentes utilizan context dispatching, para que la customización de los mismos sea extremadamente fácil.

Figura 7.3: Listado de Agenda

Snippet 7.4 Context dispatching en Empresas

```
/*Invocación de un administrador genérico para Empresas*/
$this->call(new ObjectsAdmin(
  new CompaniesCollection(AgendaModule::GetCompanies())));

/*Navegador de Empresas - Encargado de establecer los filtros
del listado*/

class CompaniesNavigator extends ObjectsNavigator {
    function initializeRestrictions() {
        /*Filtros para la búsqueda de empresas*/
    }
}
/*Context dispatching - Navigator para una colección de
empresas*/
#@defmdf &getListComponent[Component]
             (&$companies:Collection<Company>) {
        $cn =& new CompaniesNavigator($companies);
        return $cn;
}
//@#
```

Snippet 7.5 Context dispatching en Empresas - Continuación

```
/*Elemento del listado para una Empresa*/
class CompanyElement extends Element {
  function initialize() {
    parent::initialize();
    $element =& $this->getElement();
    $element->image->addInterestIn('changed',
      new FunctionObject($this, 'displayImage'));
    $this->displayImage();
  }
  function displayImage() {
    $element =& $this->getElement();
    if ($element->image->isNull()) {
      $this->addComponent(new Label(''), 'image');
    }
    else {
      $this->addComponent(new ImageDisplay(
        $element->image->getTarget()), 'image');
    }
  }
}
/*Context dispatching - Element para una empresa*/
#@defmdf &getListElement[ObjectsList](&$company:Company)
{
  $ce =& new CompanyElement($company);
  return $ce;
}
//@#
```

Editores Para la edición de Objetos, se utiliza en la mayor cantidad de los casos el editor por defecto. Ya que estos generan dinámicamente los campos para el objeto, y muestran controles correspondientes al tipo de datos del mismo, suele funcionar sin cambios automáticamente la mayoría de los casos. En los pocos casos en los que fue necesario mejorar el editor (por ejemplo, en el caso de las empresas, para mostrar el logo de la empresa en el caso de que exista) se extiende la clase `ObjectEditor`, agregando el comportamiento necesario.

Transaction safe Un problema recurrente encontrado, es que el usuario puede intentar salir de una pantalla luego de haber llenado un formulario, sin haber terminado de guardar. Peor aún, en un sistema en donde los distintos formularios se pueden encolar y el contexto se mantiene, el usuario podría estar cancelando múltiples pantallas de formularios que habían quedado abiertas. Por esto es que se introdujo un mecanismo que impide que el usuario navegue de una página cuando una transacción está abierta y se le obliga a tomar una decisión respecto de los datos existentes.

Figura 7.4: Transacción activa

Permissions Para el manejo de permisos, **Kusala** los implementa de manera similar a lo que sería un "aspecto". El manejo de permisos no es algo que forme parte del desarrollo estándar, sino que se lo agrega después mediante métodos especiales. Cada controller implementa un método `#checkAddingPermissions`, que especifica si el usuario logueado puede utilizar ese controller.

Para determinar los permisos, debido a que el mecanismo está basado en el llamado a un método en el mismo objeto ya creado, y se ejecuta dentro de PHP, el programador puede tener un sistema tan complejo como quiera (puede utilizar restricciones por tiempo, por tokens, hasta puede implementar un mecanismo de clases que se encarguen justamente de determinar los permisos del usuario).

Snippet 7.6 Editor para Empresas

```
class CompanyEditor extends ObjectEditor {
  var $with_image;
  function addDisplayFields() {
    parent::addDisplayFields();
    $this->with_image =& new ValueHolder($b = false);
    $this->addComponent(
      new Checkbox($this->with_image),
      'cb_with_image');
    $this->cb_with_image->addInterestIn('changed',
        new FunctionObject($this, 'withImage'));
    if (!$this->object->image->isNull()) {
      $this->with_image->setValue(true);
      $this->withImage();
    }
    $this->object->registerModifications();
    $this->with_image->addInterestIn('changed',
        new FunctionObject($this, 'updateModelImage'));
  }
  function updateModelImage() {
    if (!$this->with_image->getValue()) {
      $this->object->image->removeTarget();
    }
  }
  function withImage() {
    if ($this->with_image->getValue()) {
      $this->addFieldComponent(
        new Filename(new ValueHolder(''),
        new AspectAdaptor($this->object->image,
          'target')), 'image');
      $this->image->component->onChangeSend("asdf",
          $this->object);
    } else {
      $this->image->delete();
    }
  }
}
```

El Widget `CommandLink`, que es utilizado para ejecutar comandos en el sistema, chequea que la función con la que es configurado pueda ser ejecutada, y para eso esta función llama a un método de nombre `check<nombreDelMétodo>Permissions`, y de esta manera se determina si un usuario tiene permisos de ejecutar un co-

mando. Un ejemplo de esto es el mecanismo del menú superior del sistema, que se implementa como podemos ver en el Snippet 7.7.

Snippet 7.7 Permisos del Menú superior

```
/*Agregamos un Command Link con el llamado al método*/
function initialize(){
...
$this->addModuleLink(array(
    'text' => 'Administración de Documentos',
    'proceedFunction' => new FunctionObject(
       $this, 'gotoDocumentsModule'))
  , 'documents');
...
}

/*Implementamos el llamado al método*/
function gotoDocumentsModule() {
  $this->changeBody(
    new DocumentsModuleComponent(new DocumentsModule));
}

/*Implementamos el chequeo de permisos para el método */

function checkGotoDocumentsModulePermissions() {
  return SecurityAdministrator
    ::isAllowedForModule(
       User::logged(),
       'documents');
}
```

7.3 Vista

Reúso de templates En esta aplicación se hizo un reúso intensivo de los templates. Por ejemplo, existe un sólo template de listado de elementos en el sistema, y es reusado en cada una de las pantallas de listado. La pantalla de la figura 7.3 es un ejemplo de estos listados. También, todas las pantallas de edición y creación de objetos se generan con el mismo template, con muy pocas adaptaciones a algunos casos específicos. Esto permitió tanto una coherencia muy alta entre todas las pantallas del sistema, como también mucha facilidad para crear nuevas pantallas de edición (ya que las mismas, tanto en diseño como en funcionalidad son autogeneradas).

Adaptación de diseño El diseño original del sistema fue hecho en HTML, y luego fue adaptado a templates para los distintos elementos muy rápidamente y de manera simple. Estos diseños se integraron instantáneamente al funcionamiento del sistema, incluída toda interacción por AJAX.

7.4 Conclusión

La utilización de **Kusala** en este proyecto tuvo un corto tiempo de aprendizaje, y luego la creación de nuevos módulos y pantallas se vió reducida en tiempos a aproximadamente un 10 % del tiempo que hubiera tomado, al mismo grupo de programadores, realizarlo en un Framework como Symfony (que es sobre el que el mismo grupo poseía mayor experiencia). Como dato comparativo, para desarrollar el mismo sistema, un equipo de desarrolladores trabajó durante 10 meses y debió desistir debido a ciertas complejidades del proyecto. El mismo sistema, desarrollado sobre **Kusala** tomó 4 meses, debido a las mejoras que aún debieron hacerse al framework en ese momento. En este momento, el desarrollo de un nuevo módulo toma menos de 1 mes en total.

Si bien inicialmente el manejo de conceptos como el múltiple y context dispatching pueden parecer complicados, luego de un pequeño aprendizaje de los mismos, se convirtieron en herramientas invaluables. Actualmente, el tiempo de desarrollo de un nuevo módulo está restringido por un lado, por el diseño del modelo (que tiene una duración similar al diseño del modelo para cualquier otra tecnología), y luego por las pantallas con funcionalidades avanzadas para simplificar el uso del sistema.

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

8 Conclusión y Trabajo a Futuro

8.1 Conclusión

En el capítulo 2, se detallaron algunos de los problemas mas habituales e importantes que el desarrollador de aplicaciones web tiene que enfrentar.

En el capítulo 3, se pudo ver que los conceptos de reuso y composicionalidad, además de enfocarse en la declaración más que en la programación secuencial, no solamente son teóricamente factibles, sino que además son extremandamente útiles en la práctica.

Se analizó como se trabajan estos problemas en algunos Frameworks del mercado en el capítulo 4, viendo que todos ellos abarcan estos problemas de alguna manera, pero en la mayoría de los casos con soluciones parciales o inconexas con las otras soluciones, dejando al programador la correcta solución de los problemas y la integración de estas soluciones.

En el capítulo 5, se expuso el diseño de un Framework que provee soluciones a todos y cada uno de los problemas listados en el capítulo 2, de manera integrada y simple, siguiendo además los conceptos de diseño desarrollados en el capítulo 3.

En lo que respecta al desarrollo del modelo, se consiguió que la programación sea lo más cercana al ideal que el desarrollo del modelo puede llegar a ser, separando completamente el aspecto de la persistencia. Sumado a eso, se obtuvo un lenguaje de consulta que permite navegar el grafo de objetos de manera muy efectiva. El desarrollador del modelo puede enfocarse en relacionar los datos, en crearlos, conectarlos, desconectarlos, y buscarlos, sin tener que pensar en la manera en la que estos finalmente van a ser almacenados. También puede realizar estas tareas sin pensar en si los datos sobre los que está trabajando pueden ser accedidos concurrentemente por varios usuarios y se generarán errores de versionamiento. Y también puede utilizar los mecanismos de consulta independientemente de si se encuentra dentro o fuera de una transacción, lo cual consigue una abstracción total del modelo respecto de la persistencia.

Con respecto al desarrollo del controlador, con el uso de Componentes se consiguió un reúso muy alto de funcionalidad, que en conjunto con características como Mixins, Context Dispatching y Eventos, hacen que crear un componente para que sea reutilizado sea tan simple como el crear uno para ser utilizado solamente una vez. El desarrollador puede desacoplar el desarrollo del componente en el que está trabajando con el contexto en el que está embebido, y dejar al desarrollador que realiza la integración encargarse de ensamblar componentes sin tampoco preocuparse de posibles fallas en la interacción. Características como Transacciones Largas y en Memoria consiguen que el desarrollador pueda enfocarse en editar los objetos de manera concisa, sin preocuparse de temas como el pasaje de variables, la recuperación en cada request de los datos de la base de datos, y luego de la persistencia (o no) de esos datos al terminar el trabajo del componente. Además, con los Componentes que vienen con el Framework, se provee la funcionalidad común a muchos sistemas, y con todas las herramientas que se nombraron anteriormente se consigue una flexibilidad que permite adaptar cada uno de esos Componentes a la necesidad que se presente.

Por último, en cuanto a lo que es vista, con el uso de los templates declarativos y la asignación dinámica de templates se consiguió que los cuatro concerns que deberían ser separados, que son la lógica de la aplicación, el tipo de vista que la aplicación va a tener, la forma de interacción del usuario con el browser, y la presentación y estilos de la aplicación por el otro, puedan realmente ser trabajados por separado. El sistema se encarga de hacer integrar la vista correctamente sin necesidad de realizarlo manualmente cada vez.

Además de esto, el uso de PHP presentó inicialmente algunos inconvenientes para realizar todo el trabajo que era necesario para el Framework, por lo que fue necesario implementar herramientas como las macros, las assertions y el typechecking, la compilación de archivos, las referencias débiles, las variables dinámicas, y el COMET, que terminaron dando una herramienta que por un lado es altamente efectiva para realizar detección y corrección de errores, y por otro lado el sistema final es performante y utilizable en producción. Todo esto fue visto en el capítulo 6.

Por último, la posibilidad real de utilizar el Framework quedó explícita en el capítulo 7, donde se vió que no sólo se podía realizar una aplicación sobre el Framework, sino que esta es usable y performante.

8.2 Trabajo a Futuro

Durante la realización del trabajo, se encontraron varias funcionalidades que serían interesantes agregar al Framework, pero dada su complejidad se debió dejarlas para más adelante.

8.2.1. Value Expressions

Una de estas son las *Value Expressions*. Algo muy habitual en una aplicación es mostrar datos que son calculados en base a ciertas otras variables, y que tienen una dependencia directa [3].

Un ejemplo de sintaxis es el siguiente:

Snippet 8.1 Sintaxis de *ValueExpressions*

```
#@ve <expression> @#

//Que compilaría a
new ValueExpression(
  "<printf_expression>",
  <observedCollection>)

//Ejemplo, $a y $b son ValueModels
#@ve $a + $b @#

//Que compilaría a
new ValueExpression("%s +%s", array($a, $b))
```

Esto daría una expresión que se podría agregar a un Component y directamente mostrar en pantalla. El FunctionObject se ejecutaría para recalcular el dato, y la ObservedCollection no es más que una colección con múltiples elementos observables (que se envía como parámetro al function object). Si bien una implementación como la que decimos podría ser relativamente simple, cuanta mayor posibilidad de que haya un loop de dependencias haya, mayor se hace necesario un mecanismo para poder, o bien detectarlo, o bien eliminarlo directamente. Por lo tanto se decidió por el momento no implementar tal funcionalidad hasta no tener definido cómo el mecanismo de eventos debía evitar una dependencia circular.

Como ejemplo se puede tomar lo visto en la sección 5.2.2), una calculadora en el precio del producto. "Si el usuario elige que quiere más de una copia del producto, el precio final se actualizará correspondientemente (Snippet 5.11)". Haciendo unas modificaciones, y utilizando ValueExpressions, el código podría quedar como en el Snippet 8.2)

8.2.2. Testing

Algo que le sería de mucha utilidad al Framework, sería un buen soporte de casos de prueba sobre cada una de las funcionalidades. En este momento, el Framework no posee casos de prueba, y por ende cualquier modificación o refactoring se hace muy difícil.

Snippet 8.2 Sintaxis de *ValueExpressions*

```
class ProductView extends Controller{
  ...
  public function initialize(){
    $this->addComponent(
            new Text($this->product->name), 'name');
    $this->addComponent(
            new Text($this->product->price), 'price');
    //Agregamos un input para que
    //el usuario ponga la cantidad
    $this->addComponent(
            new Input($null=null), 'qty');
    $this->qty->setValue(1);
    //Mostramos el precio total
    $this->addComponent(
            new Text(
              #@ve $this->product->price * $this->qty@#,
              'totalPrice');
    $this->addComponent(
            new CommandLink(...), 'buy_now');
  }
  ...
}
```

También serviría, para el usuario final, que el sistema tuviera algún soporte o integración con un mecanismo de testing, para no sólo permitirle que escriba sus casos de prueba, sino también para incentivarlo a que los haga.

8.2.3. Múltiples Bases de Datos

El Framework tiene una estructura que separa el motor de base de datos del funcionamiento general, pero al no haber sido desarrollada una integración con otro motor de base de datos, es muy posible que mucha de la funcionalidad esté muy relacionada con MySQL. Sería bueno tanto poder dedicarle trabajo a conectar con otro motor, como también conectar directamente el sistema con una librería de abstracción, y de esa manera reutilizar el trabajo ya realizado al respecto en otras librerías, como PDO.

8.2.4. Integración con Librerías

Muchas de las funcionalidades que tiene el Framework, se verían mejoradas con la integración de distintas librerías, que no estaban disponibles para el momento

del comienzo del desarrollo y que ahora sí son de uso popular. Una de ellas es jQuery, una librería de Javascript que hubiera simplificado mucho el desarrollo de la integración con Ajax y la modificación del arbol DOM, y además hubiera traído otras cosas interesantes (como la posiblidad de agegar Widgets con animaciones).

8.2.5. Continuaciones

Por el lado del controlador, sería bueno experimentar con continuaciones y control de estructuras dinámicas para, por ejemplo, incorporar construcciones de la Programación Orientada al Contexto (POC), un campo de la informática que se desarrolló recientemente.

8.2.6. Plantillas avanzadas

Por el lado de la presentación, existe la posibilidad de mejoras en cuanto a la composición de plantillas y mejoras en la especificación de la interacción con los aspectos del controlador. Por ejemplo, es posible incorporar la noción de *combinación de plantillas*, las que reducen el uso de la copia y el pegado de XML y proponen una mejor modularización de los aspectos que conciernen a la presentación. Además, la incorporación de opciones en las plantillas permitirían experimentar con la POC, mediante *layered templates*.

8.2.7. PHP 5.3

Cuando se inició el desarrollo de **Kusala**, la versión de PHP de uso normal era la 4. Desde ese entonces se realizaron muchos cambios que sería muy bueno poder aprovechar, como el uso de los métodos `#__call()`, `#__set()` y `#__get()`, que permiten agregar dinámicamente mensajes a un objeto, el uso de interfaces, etc. Además, el paso de variables por rerefencia en PHP4 era muy engorroso, y propenso a errores por manejo de punteros, algo que fue solucionado en versiones posteriores del lenguaje. Por estas razones consideramos que sería conveniente una migración hacia PHP 5.3.

Bibliografía

[1] Ruby on rails. *http://rubyonrails.org/*, Vista por última vez: 10/05/2010.

[2] El proyecto XUL. *http://www.mozilla.org/projects/xul*, Vista por última vez: 11/12/2010.

[3] Value model. *http://st-www.cs.illinois.edu/users/brant/papers/ValueModel*, Vista por última vez: 12/04/2011.

[4] Symfony. *http://www.symfony-project.org/*, Vista por última vez: 12/09/2010.

[5] Bison. *http://www.gnu.org/software/bison*, Vista por última vez: 13/06/2011.

[6] Yet another compiler-compiler. *http://dinosaur.compilertools.net /yacc/index.html*, Vista por última vez: 13/06/2011.

[7] The official yaml web site. *http://www.yaml.org/*, Vista por última vez: 15/06/2010.

[8] Doctrine 2.0 transactions. *http://www.doctrine-project.org/docs/dbal/2.0/en/reference/transactions.html*, Vista por última vez: 17/09/2010.

[9] Java hibernate. *http://www.hibernate.org/*, Vista por última vez: 22/04/2010.

[10] Jboss seam framework. *http://seamframework.org/*, Vista por última vez: 22/04/2010.

[11] Hibernate HQL. *http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html*, Vista por última vez: 23/04/2010.

[12] Java - transaction management. *http://java.sun.com/blueprints/qanda/transaction_management/index.html #nested*, Vista por última vez: 23/04/2010.

[13] Java server faces - jsf. *https://javaserverfaces.dev.java.net/*, Vista por última vez: 25/04/2010.

[14] A. M. Alasqur, S. Y. W. Su, and H. Lam. OQL: a query language for manipulating object-oriented databases. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 433, Amsterdam, The Netherlands, August 1989.

[15] Scott W. Ambler. The design of a robust persistence layer for relational databases. 2005.

[16] Kenneth E. Ayers. The MVC paradigm in Smalltalk/V. *Dr. Dobb's Journal of Software Tools*, 15(11):168, 170, 172–174, 175, November 1990.

[17] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *In European Conference on Object-Oriented Programming*, pages 18–22. Springer-Verlag, 2001.

[18] Driscoll, Sarnak, Sleator, and Tarjan. Making data structures persistent. *JCSS: Journal of Computer and System Sciences*, 38, 1989.

[19] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC'04)*, pages 231–257, September 2004.

[20] J. Ferber. Computational reflection in class based object-oriented languages. *SIGPLAN Not.*, 24:317–326, September 1989.

[21] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *ICFP*, pages 94–104, 1998.

[22] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[23] Paul Graham. *ANSI Common LISP*. Prentice-Hall, pub-PH:adr, second edition, 1999.

[24] T. Haerder. Observations on optimistic concurrency control. *Inf. Sys.*, 9(2):111, 1984.

[25] Joseph Hallett and Assaf Kfoury. A formal semantics for weak references. Technical Report 2005-031, CS Department, Boston University, August 8 2005. Tue, 17 Feb 2009 09:46:12 GMT.

[26] David R. Hanson and Todd A. Proebsting. Dynamic variables. In *PLDI*, pages 264–273, 2001.

[27] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *PPOPP*, pages 48–60. ACM, 2005.

[28] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[29] Hudak. Building domain-specific embedded languages. *CSURVES: Computing Surveys Electronic Section*, 28, 1996.

[30] Gregor Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In Andreas Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 14, pages 335–348. MIT Press, Cambridge, Mass., 1993. An earlier version appeared in the ACM Confreence on Lisp and Functional Programming, 1990.

[31] Herbert Kircher. Web 2.0 - plattform für innovation. *it - Information Technology*, 49(1):63, 2007.

[32] Engin Kirda and Clemens Kerer. MyXML: An XML based template engine for the generation of flexible web content. December 01 2000.

[33] Alan Knight. GLORP: generic lightweight object-relational persistence. 2000.

[34] Derek Law. Taligent MVP in interactive statistical graphics. 2008.

[35] J. W. Lloyd. Practical advantages of declarative programming. Technical report cstr-94-06, Univ. of Bristol, 1994. To appear in the 1994 Joint Conference on Declarative Programming GULP-PRODE'94.

[36] Terence Parr. Enforcing strict model-view separation in template engines. January 01 2004.

[37] Linda Dailey Paulson. Building rich web applications with ajax. *IEEE Computer*, 38(10):14–17, 2005.

[38] Michael Perscheid, David Tibbe, Martin Beck, Stefan Berger, Peter Osburg, Jeff Eastman, Michael Haupt, and Robert Hirschfeld. *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 2008.

[39] PHP Hypertext Processor. *http://www.php.net*.

[40] Mike Potel. Mvp: Model-view-presenter the taligent programming model for c++ and java, taligent inc. Technical report, 1996.

[41] Christian Queinnec, Universit Paris, and Pierre Marie Curie. Inverting back the inversion of control or, continuations versus page-centric programming. Technical report, May 04 2001.

[42] Nandakumar Sankaran. Bibliography on Garbage Collection and Related Topics. *SIGPLAN Notices*, 29(9):149–158, September 1994.

[43] Richard M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. Technical Report AIM-519A, MIT Artificial Intelligence Laboratory, March 6 1981.

[44] Jason Cranford Teague. *DHTML for the World Wide Web*. Visual quickstart guide. Peachpit Press, Inc., pub-PEACHPIT:adr, 1998.

[45] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.