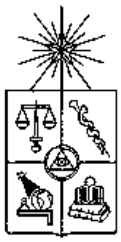


UNIVERSIDAD DE CHILE
FACULTDAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE LA TERCERA VERSIÓN DEL FRAMEWORK JAVA PARA APLICACIONES WEB DE LA EMPRESA DYBOX

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SIMÓN ANDRÉS PAREDES STECHER

PROFESOR GUÍA:
JOSÉ ALBERTO PINO URTUBIA

MIEMBROS DE LA COMISIÓN:
LUIS GUERRERO BLANCO
KURT WILHELM SCHWARZE MEZA

SANTIAGO DE CHILE
ENERO 2009

Resumen

El presente trabajo de título tuvo por objetivo diseñar e implementar una versión actualizada y más fácil de usar del framework de desarrollo para aplicaciones Web de la empresa Dybox. Esta nueva versión debe actualizar las componentes externas, mejorar y rediseñar las distintas capas que lo conforman junto con agregar nuevas funcionalidades.

Dybox es una empresa que se dedica al desarrollo de software y en especial al de aplicaciones Web. Para esto es que cuenta con un framework Java de desarrollo Web que se encuentra actualmente en la versión 2.5 y que data del año 2006. Este framework permite construir todos los tipos de aplicaciones que Dybox desarrolla.

Un framework se puede definir como un conjunto de APIs, herramientas y metodologías destinadas a la construcción de un software de manera que se reduzcan las dificultades técnicas y de diseño general, junto con promover la extensibilidad y adaptación. Esto permite que el foco del desarrollo se centre en los requerimientos específicos del software a construir y no en las problemáticas descritas.

El trabajo se realizó usando la metodología de desarrollo dirigida por pruebas o Test Driven Development (TDD). Además se usó un proceso de desarrollo incremental en el que primero se genera una versión simple de cada requerimiento y luego se analiza y mejora según sea necesario. Durante el desarrollo del framework, se contó con que dos aplicaciones fueran desarrolladas usándolo. Esto permitió tener acceso a una rápida retroalimentación, lo que redundó en una implementación mejorada.

Se obtuvo un nuevo framework con el cual se desarrollaron dos aplicaciones mientras se realizaba este trabajo y dos más comenzarían su desarrollo próximamente. Este nuevo framework redujo considerablemente la cantidad de configuración necesaria para funcionar, permitiendo que sea más simple de usar y aprender. Se actualizaron las componentes en las que depende el framework con lo que sus nuevas capacidades fueron incorporadas. Se agregó la capacidad de desarrollar servicios Web de manera simple y limpia con el uso del estándar JAX-WS. Se documentaron todas las clases y funciones relevantes del framework usando javadoc, lo que permite un desarrollo más fácil apoyado por un IDE. En general se cumplieron todos los objetivos planteados, dejando una base sólida para mejoras y nuevas funcionalidades.

Agradecimientos

Primeramente agradezco a Dios por darme la vida y ayudarme en cada uno de los largos semestres que pasé en esta facultad.

Agradezco también el apoyo prestado por el profesor guía, Sr. José Alberto Pino U. quien hizo posible que este trabajo sirviera como trabajo de memoria para acceder al título. También agradezco a la empresa Dybox, representada por el Sr. Felipe Martin y Hector Araya, quienes me entregaron su apoyo y aprobación para realizar este trabajo.

Del mismo modo, agradezco a mi esposa Sabina, quien me brindo constante apoyo, ayuda y soporte para llevar a cabo este trabajo en las mejores condiciones que me pudo brindar.

Agradezco a mi familia por brindarme su constante apoyo para estudiar esta carrera en la que tuve muchos bajos al principio, pero que fueron superados por grandes altos como es este trabajo. Agradezco a mi tío Carlos por apoyarme monetariamente durante todos los años de estudio en esta universidad, gracias a ti, me pude dedicar 100% a estudiar. Agradezco a mis padres por enseñarme que en la vida las cosas se obtienen con esfuerzo y dedicación.

Índice

1. Introducción

Dybox es una empresa que se dedica al desarrollo de aplicaciones y sitios Web de tipo e-commerce, e-business, e-marketing, e-government, Intranet, Extranet y aplicaciones operacionales [1]. Para esto es que cuenta con un framework Java de desarrollo Web que se encuentra actualmente en la versión 2.5 y que data del año 2006. Este framework permite construir todo el rango de aplicaciones antes nombrado.

El mundo globalizado en el que vivimos hoy promueve el acceso remoto a las aplicaciones. Esta nueva modalidad de acceso permite hacer más eficiente y económico el trabajo con ellas, al poder ser usadas desde cualquier lugar del mundo. Las que más éxito han tenido en este ámbito son las aplicaciones basadas en Web. Estas aplicaciones han ido reemplazando día a día a las tradicionales, desde el correo electrónico hasta las que controlan complejos procesos comerciales [2] o gubernamentales [3].

La creciente demanda de estas aplicaciones significa un desafío para su desarrollo en Dybox. Este desafío nace de la necesidad de interactuar con sistemas legados, la complejidad intrínseca al ambiente de ejecución de este tipo de aplicaciones, sumado a la competitividad del mercado y su constante evolución. Todo esto redunda en la necesidad de contar con un framework que simplifique el proceso de desarrollo.

Se puede definir framework como un conjunto de APIs, herramientas y metodologías destinadas a la construcción de un software de manera que se reduzcan las dificultades técnicas y de diseño general, junto con promover la extensibilidad y adaptación [4]. Esto permite que el foco del desarrollo se centre en los requerimientos específicos del software a construir y no en las problemáticas antes mencionadas.

Para que un framework sea exitoso, éste debe ser sencillo de usar, intuitivo, robusto y, por sobre todo, debe ocultar las complejidades de la implementación. Lo anterior permite al desarrollador trabajar enfocado en resolver el problema del cliente y no los técnicos. Además debe permitir que el desarrollador inexperto en el framework pueda dar sus primeros pasos sin tener que lidiar con toda la complejidad del mismo. La forma de lograr

este objetivo es introduciendo el conocimiento de cada una de sus áreas a medida que se avanza en el desarrollo. El resultado final es una curva acelerada de aprendizaje, lo que es altamente beneficioso para Dybox.

1.1 Justificación

Con el framework actual de Dybox se han construido diversas aplicaciones. Estas aplicaciones pertenecen a variados ámbitos. Algunos ejemplos de desarrollos son:

- Sistema de Captura Automática de Cartolas, BICE VIDA, Compañía de Seguros S.A.
- Sistema de Cálculo y Pago de Subvenciones, Ministerio de Educación, Chile.
- Control de Inventario, M&G Consultores.

Los problemas principales de los que adolece el framework actual de Dybox son:

- Estar desactualizado, ya que data del año 2006.
- Tener una configuración muy extensa y compleja.
- Tener una capa de presentación fuertemente acoplada al framework, lo que hace imposible cambiarla.
- No permite disponibilizar Servicios Web (Web Services).

El desarrollo del framework de Dybox se estancó a mediados del año 2006 producto de fuertes cambios en la organización interna de la empresa. Estos cambios produjeron que el arquitecto de Dybox, quien fue el encargado del desarrollo del mismo, no siguiera trabajando a tiempo completo. Bajo estas condiciones, es que el tiempo disponible del arquitecto se utilizaba en resolver problemas puntuales de cada proyecto y no en seguir su desarrollo. Finalmente a comienzos del 2007, el arquitecto dejó de trabajar para Dybox, prestando sólo asesorías esporádicas hasta el día de hoy. Por otro lado, todos los requerimientos tecnológicos de los proyectos realizados en Java fueron cubiertos por el framework actual.

Las componentes en las que se basa el framework actual de Dybox han evolucionado incluyendo nuevas funcionalidades y mejoras sustanciales. Sin embargo, esta evolución ha significado cambios fuertes en su estructura, por lo que para poder integrarlas al framework, es necesario cambiar el núcleo del mismo.

El framework actual de Dybox se basa en la versión 1.4 de Java, sin embargo, en la actualidad Java se encuentra en la versión 1.6 [5]. La comunidad de software de código abierto ya ha migrado casi por completo a la versión 1.5, dado que la 1.6 se considera muy nueva al sólo llevar poco más de un año de vida. Es por este motivo que la nueva versión del framework de Dybox debe basarse en la versión 1.5 de Java. Los cambios y ventajas de la versión 1.5 por sobre la versión 1.4 son significativos. Estos cambios pasan por nuevas estructuras de control y expresividad en el código, nuevas APIs y mejoras de performance, entre otros [6]. Sin embargo, dichos cambios las hacen incompatibles hacia atrás.

Otro problema que presenta la versión actual de framework de Dybox es que cuenta con un sistema de configuración, basada en archivos de texto, muy extenso y complejo. Este problema conlleva a que los desarrolladores no logren manejar la configuración por ellos mismos, requiriendo siempre de algún superior para poder ajustarla a sus necesidades. Finalmente esto causa que el desarrollador pierda la atención en dar solución al problema del negocio, enfocándola en comprender y mantener dichos archivos a medida que logra avanzar.

En consecuencia, la solución que se ha planteado en el mundo del software es utilizar el patrón de diseño que prioriza la Convención por sobre la Configuración (CoC) [7]. Con este patrón se logra reducir el número de decisiones que los desarrolladores tienen que tomar. Esto simplifica su desarrollo sin perder flexibilidad en las opciones de configuración. La idea final del CoC es contar con un valor por defecto para cada configuración necesaria sin tener que escribirla y además poder especificar un valor distinto para ésta cuando sea necesario. Un ejemplo de CoC sería que para una clase Persona su validador debe llamarse ValidadorPersona. Si se hace de esta forma, no será necesario

especificar en ningún lugar que dicha clase es la que valida a la clase Persona. Si por el contrario, se llama al validador VldrPer, debe existir algún lugar donde especificarlo.

Por otro lado, el patrón de diseño denominado Inyección de Dependencia (ID) [8] ha tenido un gran desarrollo en los últimos 3 años. Este patrón nos ofrece los beneficios de simplificar el código y la capacidad de reducir el acoplamiento. Los frutos de este desarrollo son la aparición de variados frameworks tales como PicoContainer [9] y SpringFramework [10]. El primero de éstos es sólo un contenedor que permite el uso de ID. El segundo es una plataforma completa que, además de contar con un contenedor para el uso de ID, permite la integración con otros frameworks. Algunos de los frameworks más relevantes con los que se integra son Hibernate [11] y JUnit [12]. Hibernate es un mapeador de objetos al mundo de bases de datos relacionales y JUnit permite hacer pruebas de unidad al código. SpringFramework también cuenta con gran soporte para el desarrollo de aplicaciones Web, además de ser implementado usando patrones de diseño como CoC y AOP [13].

Dadas las virtudes antes expuestas, es que el uso de ID se plantea como parte fundamental de la nueva versión del framework de Dybox. Para su implementación se considerará el uso de SpringFramework.

El framework actual de Dybox se estructura usando el patrón MVC [14]. La implementación de este patrón lleva a la división en tres capas, las cuales son Presentación, Lógica y Datos. En la capa de presentación se utiliza las tecnologías XML [15] /XSLT [16]. El problema actual con la capa de presentación es que se ha hecho muy dependiente el framework de ella. Esto hace imposible utilizar otras tecnologías como JSP [17] o JSF [18] en esta capa.

Es por este motivo que se desea rediseñar el nexo con la capa de presentación, para que el framework nuevo no esté acoplado a la tecnología a utilizar. Se pretende lograr que sea fácil cambiar de XML/XSLT a JSP u otro.

La Arquitectura Orientada a Servicios (SOA) [19] ha tenido un fuerte impacto en la industria nacional. Esta arquitectura permite que las empresas puedan reusar funcionalidades de una aplicación en otras al ser expuestas como Servicios Web [20]. Asimismo permite integrar sistemas legados con sistemas nuevos al ser expuestos de la misma forma. Este fuerte cambio en la industria hace que se transforme en una necesidad para Dybox el contar con un framework que permita disponibilizar Servicios Web de forma rápida y sencilla.

1.2 Objetivo general

Diseñar e implementar una versión actualizada y más fácil de usar del framework de desarrollo de la empresa Dybox. En esta nueva versión se actualizarán las componentes externas y se mejorarán y rediseñarán las distintas capas que lo conforman junto con agregar nuevas funcionalidades.

1.3 Objetivos específicos

Actualizar las distintas componentes externas de las que depende el framework. Esto permitirá incorporarle nuevas funcionalidades.

Rediseñar cada capa de la implementación MVC del framework. El rediseño se hará usando los patrones ID y CoC como fundamentos. Se busca agilizar el desarrollo al simplificar la configuración.

Introducir la capacidad de hacer intercambiable la capa de presentación. De esta forma se hará más flexible el framework al poder soportar otras tecnologías de presentación como JSF, JSP u otras. Este requerimiento nace de los clientes a quienes no les acomoda el uso de XML/XSLT en esta capa.

Integrar la capacidad de exponer funciones como Servicios Web de forma sencilla y lo menos intrusiva posible.

Comparar básicamente los procesos de desarrollo con el actual y nuevo frameworks. En particular, estudiar los aspectos de usabilidad.

1.4 Revisión bibliográfica

1.4.1 Contenedores pesados y livianos

El estándar de Java para aplicaciones Web plantea que éstas deben dividirse en al menos 2 secciones; una que se encarga de la presentación y otra que maneja la lógica y acceso a datos [21]. Bajo esta especificación es que aparecen los contenedores, cuya función es proveer del ambiente necesario para la ejecución de cada sección.

La especificación del contenedor para la sección de presentación se llama Servlet [22]. Ésta se mantiene en constante desarrollo y en la actualidad se encuentra en la versión de producción 2.5. Este estándar ha tenido una gran aceptación desde sus inicios en el mundo de la industria y el del software de código abierto.

La especificación del contenedor para la sección de lógica se llama Enterprise Java Beans (EJB) [23]. Ésta también se mantiene en desarrollo y en la actualidad se encuentra en la versión de producción 3.0. Sin embargo, la aceptación de esta especificación ha sido baja a lo largo de la historia y muy criticada [24]. La principal crítica de sus versiones anteriores apunta a que era necesario hacer demasiado trabajo para lograr una funcionalidad tan básica como un “Hola Mundo”. Se debía implementar 2 interfaces predefinidas, una remota y una local. Cada interfaz cuenta con sus funciones específicas, las cuales debían ser implementadas y que, en la mayoría de los casos, no se usaban. Además era necesario escribir un archivo XML en el cual se debía especificar absolutamente todo, convirtiéndose en un gran archivo de configuración difícil de mantener. Sin embargo, a contar de la versión actual se hicieron cambios fuertes en la especificación permitiendo que el desarrollo sea más rápido. Los cambios más importantes son que ya no se debe implementar interfaces predefinidas y que en el archivo de configuración sólo debe especificarse lo relevante a la aplicación. A pesar de estos cambios positivos, sigue

teniendo una baja aceptación porque continúa requiriendo de muchos elementos para funcionar.

En este contexto fue que apareció la terminología adhoc de contenedores pesados y livianos. A los contenedores del estilo EJB se les denominó pesados por todo el trabajo adicional necesario para poder hacer uso de ellos y además porque requieren de una gran maquinaria para funcionar [25]. Esta maquinaria extra hace que el tiempo de arranque de la aplicación sea extenso y aumenta el requerimiento de memoria. También influyen en el nombre de liviano o pesado lo complejo que es aprender a usar la tecnología y qué tan fácil es probar los componentes desarrollados con ella. En consecuencia, se les llama livianos a los que requieren pocos recursos para funcionar, son simples de aprender, rápidos y fáciles de testear.

SpringFramework es considerado un contenedor liviano principalmente porque no requiere de una gran maquinaria para ejecutarse y porque es simple de aprender [26]. Una de las principales ventajas que tiene es el énfasis en ser un framework no invasivo. Esto se refleja en que para poder usar las funcionalidades básicas no se requiere agregar código extra referente a SpringFramework en las clases desarrolladas. Sólo se requiere de un archivo de configuración XML en el que se hace referencia a las clases que se desea que SpringFramework administre. A diferencia del archivo de configuración de EJB, éste es mucho más simple y sólo requiere especificar lo mínimo necesario.

Otros contenedores livianos de código abierto para aplicaciones Java son: DNA [27], Excalibur [28] y Soto [29], entre otros.

En términos de desarrollo, la gran ventaja de usar un contenedor liviano es que permite poder ejecutar una prueba de unidad en corto tiempo. Esto se debe a que el ciclo de carga del contenedor es corto. Ésta es una característica vital con la que el nuevo framework de Dybox debe contar. Es necesario contar con esta característica de rendimiento porque el trabajo con pruebas de unidad es parte fundamental de la metodología de desarrollo de Dybox.

1.4.2 Mapeo de objetos a base de datos relacionales (ORM)

El mapeo de objetos a bases de datos relacionales consiste en generar un puente entre dos mundos, el de las bases de datos relacionales y el de los lenguajes orientados a objetos (OO) [30]. Con esto se logra crear una base de datos virtual de objetos que puede ser accedida desde el lenguaje de programación.

Generar el puente entre ambos mundos es una tarea que reviste una gran dificultad. La dificultad nace de las grandes diferencias en los aspectos fundamentales de cada mundo. Estas diferencias son: Tipos de datos, Encapsulamiento, Estructura e Integridad, Capacidad de Manipulación y Transaccionalidad. Este problema se conoce como Object-Relational Impedance Mismatch [31].

El beneficio de usar un ORM está en ganar tiempo al no tener que preocuparse de escribir instrucciones SQL repetitivas y que pueden ser foco de errores. También se logra simplificar el desarrollo al delegar la complejidad de acceder a la base de datos al ORM. Otro beneficio que se logra es que no se requiere de desarrolladores expertos en SQL para implementar las aplicaciones. Sin embargo, estos beneficios pueden transformarse en desventajas bajo ciertas circunstancias donde lo mejor es usar una consulta SQL y no apegarse al paradigma OO.

Hibernate es un framework ORM que provee todos los beneficios que se pueden esperar de este tipo de herramientas además de ser un software de código abierto. Provee una configuración basada en archivos XML relativamente simple, y todas las alternativas necesarias para tener control fino del SQL generado. También provee de un lenguaje tipo SQL que se llama Hibernate Query Language (HQL). HQL es similar a SQL pero orientado a objetos, permitiendo escribir consultas de forma más simple y libres de ser específicas para un tipo de motor de base de datos. Además, provee herramientas que se integran con el IDE Eclipse que permiten probar fácilmente que la configuración esté correcta, probar los queries HQL y generar la configuración a partir de clases Java o una base de datos.

El framework actual de Dybox usa Hibernate en su versión 2.1. Sin embargo, hoy se encuentra en la versión 3.2, presentando grandes mejoras como HQL más potente, una nueva arquitectura del tipo Interceptor/Callback, filtros definidos por el usuario y la capacidad de definir la configuración usando Annotations [32]. Todas estas mejoras se necesitan para la nueva versión del framework de Dybox.

1.4.3 Arquitectura Modelo Vista Controlador (MVC)

Este patrón de arquitectura consiste en separar los intereses propios de cada una de las distintas unidades que define. Estas unidades son: Modelo, Vista y Controlador. La Vista se encarga de todo lo referente a la presentación de la aplicación, como son las imágenes a mostrar, cajas de textos, distribución de los elementos en la pantalla, colores, formatos, entre otros. El Modelo se encarga de la lógica o reglas del negocio y del acceso a datos. Finalmente, el Controlador actúa como un interlocutor entre la Vista y el Modelo llevando las solicitudes del usuario al Modelo y notificando a la Vista según sea necesario [33].

Los beneficios de usar MVC son: interfaz de usuario intercambiable fácilmente, reusabilidad de componentes del Modelo en distintas Vistas, facilidad de soportar nuevos tipos de clientes y facilitar las pruebas de unidad. Los problemas que produce el uso de MVC son: complejidad creciente de desarrollo y acoplamiento desde la Vista hacia el Modelo.

Esta arquitectura ha sido usada en las aplicaciones Java Web desde sus comienzos con la aparición del framework de código abierto Struts [34] hacia el año 2001 con su primera versión estable y una lista grande de otros frameworks que han aparecido a lo largo del tiempo implementando este mismo patrón. Sin embargo, el porqué de su uso no está del todo justificado. La principal motivación para usarlo es la separación de intereses que promueve, ya que el dominio de la Vista guarda demasiadas diferencias con el del Modelo.

Una de las críticas que se le hace a su uso en aplicaciones Web, es que originalmente fue pensado para aplicaciones de escritorio. Por otro lado, ocurre que existen variadas interpretaciones de lo que es un Controlador [35]. Esta falta de definición por parte del autor y su posterior ajuste al mundo de aplicaciones Web, ha llevado a una gran confusión en el medio. Cada framework tiene su propia interpretación e implementación. Lo anterior conlleva a que sea necesario entender primero la filosofía del autor antes de poder usar el framework correctamente.

La alternativa que surge es el uso del patrón Programación Orientada a Eventos (EVP) [36], el cual lleva a asociar una función con cada evento disponible en el sistema, tales como: clic en botones o links, la carga de la página en el navegador, entre otras. Este patrón permite un desarrollo acelerado de la aplicación, pero adolece de no ser escalable y llevar al programador a escribir código desordenado. En general se recomienda su uso para el desarrollo de aplicaciones pequeñas y de tiempo corto de desarrollo.

1.4.4 Arquitectura Orientada a Servicios (SOA)

La arquitectura SOA consiste en modelar las funcionalidades requeridas por una empresa en torno a procesos de negocio. Estos procesos son presentados como Servicios Web al resto de las aplicaciones de la empresa. De esta forma se logra que interactúen entre ellos y que formen las aplicaciones de negocio que el usuario final empleará [37].

La generación de Servicios Web a partir de código Java es un tema abordado por el estándar de Java a partir de la versión 3.0 de EJB. El primer framework de código abierto en hacerse masivo en esta área fue Axis [38] en su versión 1.0 (Axis1) en el año 2002.

Axis1 permite exponer servicios de forma relativamente simple y se integra fácilmente a cualquier aplicación Web ya existente. Sin embargo, no presta soporte a las nuevas especificaciones de Web Services y se considera pobre en rendimiento [39]. Axis en su versión 2 (Axis2) fue un completo rediseño de su arquitectura. Este rediseño le permitió lograr un mejor rendimiento y poder mantenerse al día con las especificaciones para Servicios Web. Sin embargo, esta reestructuración produjo que el salto de Axis1 a Axis2

fuese difícil de llevar a cabo para sus usuarios. Otra consecuencia de la reestructuración fue que ya no fuese fácil integrarlo a una aplicación Web ya existente.

Otro framework para este propósito es XFire [40] que lanzó su versión 1.0 hacia el año 2006. Según las pruebas realizadas por ellos mismos, XFire es de 3 a 5 veces más rápido que Axis1 y más simple de usar. En la actualidad XFire se fusionó con Celtix [41], otro framework con el mismo propósito, dando vida a CXF [42].

CXF es un framework de código abierto que se basa en SpringFramework como contenedor. Los principales beneficios de CXF son: soportar los estándares más nuevos de Servicios Web, soportar varios tipos de "frontend" que le permiten disponibilizar servicios usando Annotations o archivos de configuración XML y finalmente, soportar protocolos binarios y legados. CXF fue diseñado para ser fácil e intuitivo de usar. Además de ser simple y sencillo de agregar a una aplicación Web ya existente. La arquitectura de CXF cuenta con muchos puntos donde se puede añadir interceptores del flujo de datos. Esto permite lograr un sinfín de arreglos posibles según la necesidad de la aplicación desarrollada. Lo anterior no sería posible si CXF no contara con un sistema versátil de configuración.

1.5 Plan de trabajo

1. Definir los requisitos que servirán de guía en el desarrollo de la arquitectura.
2. Definir las componentes y sus versiones a utilizar en el desarrollo.
3. Diseñar e implementar una primera versión de la solución de cada requisito.
4. Analizar la solución obtenida y evaluar si requiere de un nuevo diseño e implementación.
5. Rediseñar e implementar una nueva solución si se estimo necesario en el punto anterior.
6. Se repetirán los pasos 3, 4 y 5 todas las veces que sea necesario hasta lograr un resultado satisfactorio. De esta forma se realizará un desarrollo iterativo e incremental.

7. Documentar la convención a usar en cada una de las componentes o áreas del framework. Esta convención debe ser desarrollada a lo largo de las actividades 3, 4 y 5.
8. Documentar el resto del framework.
9. Obtener una primera evaluación del impacto del nuevo framework en el desarrollo de una aplicación real de la cartera de Dybox.

1.6 Metodología de trabajo

El trabajo se realizó usando la metodología de desarrollo dirigida por pruebas o Test Driven Development (TDD) [44]. Esta metodología consiste en primero generar una prueba sobre el o los requisitos a implementar; luego, se comprueba que fallen, seguidamente se escribe el código mínimo necesario para que la prueba pase satisfactoriamente y, como último paso, se procede a refactorizar[1] el código según sea necesario. La idea es generar código simple y que cumpla con el requerimiento, de tal forma que no se genere código que no se usa o que aborda casos que no es necesario considerar. Una ventaja no evidente del uso de TDD es que primero el desarrollador debe enfocarse en cómo el código cliente va a utilizar sus clases, guiando el diseño de la interfaz, y no en cómo implementar su funcionalidad. De esta forma se pone énfasis en la calidad de la interfaz, permitiendo generar código de mejor calidad al enfocarse primero en el "qué" y luego en el "cómo".

Los pasos del TDD son:
- Seleccionar el o los requerimientos a cubrir.
- Escribir la prueba.
- Verificar que la prueba termine con error.
- Escribir la implementación.
- Verificar que la prueba termine con éxito.
- Refactorizar para mejorar la calidad del código.

[1] Termino común dentro de la ingeniería de software que hace referencia al proceso de cambiar la estructura interna de un software sin alterar su funcionamiento externo. El ejemplo más básico de refactoring sería cambiar el nombre de una variable t a tiempo. Por primera vez referenciado en [43].

- Actualizar la lista de requerimientos con nuevas necesidades que pudieran surgir de este ciclo de desarrollo.

Para poder emplear esta metodología de forma exitosa se requiere de herramientas que permitan realizar cada uno de los pasos de la forma más simple y directa. Eclipse fue el IDE utilizado para el desarrollo, el cual fue elegido teniendo en cuenta su capacidad para realizar estas tareas. Para las pruebas se usó JUnit, que es un framework para el desarrollo de pruebas de unidad, el cual está completamente integrado con Eclipse. La rápida y completa funcionalidad de refactoring que posee este IDE permitió al autor mantener el código claro y legible. Las principales facilidades que presta son: permitir cambiar el nombre de las clases, de las funciones, de los campos privados, cambiar el package al que pertenecen, extraer funciones al seleccionar una sección de código, crear una función cuando se hace referencia a ella y no existe, creación de funciones de acceso (getters y setters) y formatear el código.

Esta metodología es fácil y simple de aprender, pero requiere que el desarrollador tenga muy claro cuales son las funcionalidades que debe implementar para poder generar pruebas de unidad que las validen correctamente. Si este conocimiento no está lo suficientemente madurado por el desarrollador, entonces se generarán pruebas de unidad que no reflejan las necesidades impuestas por el requerimiento. Esto redunda en que el código necesario para pasar la prueba de unidad no cumple con lo que el requerimiento pide. Finalmente se termina con pruebas de unidad que pasan satisfactoriamente, pero que no aportan valor al desarrollo. La única forma de evitar este mal, es que el desarrollador se dé el tiempo de comprender el requerimiento a cabalidad antes de programar la primera línea de código.

2 Diseño e implementación

En este capítulo se detalla en profundidad el trabajo realizado. Primero, se enunciarán los requisitos sobre los cuales se realizó este trabajo. Luego, se expondrá el criterio de elección de las componentes principales y la determinación de las versiones a utilizar de éstas. En seguida se verá el diseño e implementación de cada requisito y, finalmente, un resumen de las convenciones generadas.

2.1 Requerimientos

Los requerimientos aquí expuestos fueron los que guiaron el desarrollo de este trabajo. Éstos fueron entregados por el gerente del área de producción de Dybox. Él en su rol de arquitecto general de toda la plataforma de desarrollo de Dybox, generó estos requisitos con miras a mejorar la calidad y el proceso de desarrollo del software producido por la empresa. Cada uno de los requisitos fue analizado en conjunto con el autor para lograr así obtener una visión afín de cuál es el objetivo que se busca con cada uno de ellos.

2.1.1 La aplicación debe ser configurable

Se debe poder especificar valores que la aplicación pueda recuperar durante la ejecución. Ejemplo de esto sería recuperar una casilla de correo a la cual enviar alertas.

2.1.2 La estructura de la aplicación debe ser configurable

Se debe poder especificar cuáles son las componentes de software a usar cuando sea necesario. Se requiere el uso de ID para cumplir con este requisito. Ejemplo de esto es una aplicación desarrollada con el patrón DAO y se desea especificar con cual implementación de DAOs se quiere trabajar, los de desarrollo o producción.

2.1.3 Se debe contar con una herramienta de log configurable e intercambiable

La escritura de logs es fundamental para la depuración de los procesos, así como también para la generación de estadísticas de uso de las componentes del software. Se requiere de una herramienta de log que permita tener varias implementaciones, permitiendo que sean intercambiables según la necesidad de la aplicación. De esta forma, se puede usar implementaciones que guarden el registro en archivos de texto, base de datos, colas y otros.

2.1.4 Se debe evitar la creación de instancias de clases innecesariamente

El ciclo de las aplicaciones Web se basa en peticiones del usuario al servidor. Es muy fácil que el programador cree los mismos objetos una y otra vez para cumplir con cada petición del cliente. Esto es ineficiente, ya que muchos de ellos pueden ser reutilizados para servir a varias o todas las peticiones que se generen. Se debe lograr una arquitectura que favorezca la reutilización de objetos evitando la creación innecesaria de éstos por parte del programador.

2.1.5 Soporte para la publicación de una funcionalidad como Servicios Web

Actualmente existen variadas opciones tecnológicas para lograr este objetivo, sin embargo, muchas de ellas requieren de una gran cantidad de configuración y mantención. Se requiere de una herramienta que permita publicar una funcionalidad como Servicios Web sin tener que incurrir en una gran cantidad de configuración y que al mismo tiempo abstraiga al desarrollador de la necesidad de comprender todas las complejidades de esta tecnología para poder implementar un Servicio Web simple.

2.1.6 Capacidad de intercambiar la capa de presentación

Actualmente, Dybox trabaja usando una capa de presentación basada en la tecnología XML/XSL. Se quiere seguir trabajando con ella, pero que se tenga la suficiente libertad como para poder cambiar a JSP u otra tecnología.

2.1.7 Actualizar las versiones de las componentes del framework

Se deben actualizar a las más nuevas disponibles. Se debe mantener bajo el acoplamiento con estos componentes para que sea lo más fácil posible actualizarlos en el futuro.

2.1.8 Se debe contar con un entorno que permita ejecutar pruebas de unidad

La idea es proveer de las piezas de software necesarias para poder ejecutar la aplicación dentro de un entorno de pruebas de unidad de la forma más simple y directa posible. Se debe lograr ejecutar la prueba de unidad con la menor cantidad de cambios a la configuración. Es muy importante evitar la necesidad de duplicar configuración para ejecutar las pruebas de unidad.

2.2 Determinación de las componentes principales a utilizar

En esta sección se detallarán las razones por las cuales se eligieron cada una de las componentes relevantes del framework. Cada una de estas componentes tiene a su vez sus propias dependencias en otras componentes más pequeñas. Generalmente, parte de estas componentes pequeñas son comunes entre las componentes principales, por lo que se debió investigar además los problemas de choque de versiones entre ellas.

2.2.1 Contenedor Liviano de Aplicaciones

Para dar solución al requisito 2 es que se requiere de un Contenedor Liviano de aplicaciones que permita el uso de ID para configurar la estructura de la aplicación.

El primer candidato fue SpringFramework, el cual ya era conocido por el desarrollador. SpringFramework es una gran biblioteca de componentes donde el núcleo es un contenedor de aplicaciones que permite el uso de ID. Es un proyecto de código abierto gratuito. Cuenta con una gran comunidad que lo respalda y da soporte, lo que permite que el ciclo entre versiones publicadas sea corto, permitiendo así contar con un software robusto y probado. Otro punto a favor de este contenedor es su amplio uso en otros frameworks, lo que permite que sea más fácil integrarlos al nuevo framework de ser necesario. Además, desde el otro punto de vista, SpringFramework posee integración con otros frameworks populares, lo que también permite una fácil integración con éstos.

El segundo candidato fue PicoContainer el cual también permite el uso de ID. Este contenedor enfatiza el uso de la inyección de las dependencias mediante el constructor, a diferencia de SpringFramework que se centra en la inyección mediante las funciones "set" (setter) de cada dependencia. Este planteamiento conlleva a que sea necesario definir de antemano un constructor con todas las dependencias o, en su defecto, varios constructores con las distintas permutaciones aceptadas. Esta forma de inyectar las dependencias hace que la aplicación sea menos flexible, ya que se requiere recompilar si no existe un constructor que satisfaga la configuración que se desea realizar. Por otro lado, PicoContainer es sólo un contenedor, mientras que SpringFramework es una gran librería que tiene como núcleo al contenedor. Una ventaja de ser sólo un contenedor es que es muy liviano y rápido de levantar, versus el otro contenedor. Al realizar una búsqueda en Google se puede apreciar la baja participación que tiene PicoContainer versus SpringFramework en el mundo web que es de 118.000 registros vs. 2.830.000 de SpringFramework.

A pesar de existir otros contenedores como DNA, Excalibur o SOTO, éstos no fueron estudiados en detalle al tener una cuota de participación mucho menor que PicoContainer, lo que implica que la comunidad que les da soporte es más pequeña aún. La falta de soporte en el futuro, por la gran posibilidad que tienen de extinguirse en el tiempo, fue el mayor motivo para descartarlos desde un principio.

Finalmente se decidió usar SpringFramework como contenedor de ID por todas las ventajas que presenta frente a PicoContainer.

2.2.2 Framework ORM

El framework ORM a utilizar debe ser el mismo de la versión antigua del framework. La versión antigua del framework ocupa el ORM Hibernate en su versión mayor 2. Para cumplir con el requisito 7, se buscó la versión más nueva de éste y en base a ella se realizó el desarrollo. Al momento de hacer la búsqueda se encontró que la versión mayor más nueva era la 3, por lo que fue ésta la que se seleccionó.

2.2.3 Framework para Servicios Web

Las funcionalidades esperadas de este framework son que permita cumplir con el requisito 5, es decir, que permita disponibilizar fácilmente una funcionalidad como Servicios Web. Además de esto, se espera que sea lo menos invasivo posible y que permita una integración sencilla con el resto del framework. Para esto, los dos frameworks analizados fueron CXF y Axis2.

CXF se basa en SpringFramework para su configuración, por lo que hace muy simple su integración con el framework, mientras que Axis2 usa sus propios archivos descriptores.

Para poder determinar cuál de los dos se usaría se realizó una prueba de concepto. Esta prueba de concepto consistió en crear una aplicación básica compuesta por una clase Java que prestara un servicio y una página JSP que consumiera dicho servicio. El servicio implementado fue una función que saluda, la que recibe un string y responde el mismo string agregándole al principio el texto “Hola ”. La idea fue evaluar la complejidad de disponibilizar dicho servicio como Servicio Web utilizando ambos frameworks.

El resultado obtenido fue que con CXF se logró el objetivo en menor tiempo, con menos archivos de configuración y clases y con una complejidad menor versus Axis2.

A pesar de que CXF en si es un proyecto mucho más nuevo que Axis2 y, por ende, menos usado, se prefirió usar el primero por su gran facilidad de integración a una aplicación ya existente. Junto a esto se privilegió el hecho de que para su configuración sólo se requiera de SpringFramework, reduciendo la cantidad de archivos y clases y, además, que no sea invasivo a la aplicación para poder ser utilizado.

2.2.4 Framework para el manejo de logs

El framework más ampliamente usado para esta tarea es log4j. Log4j es un framework de código abierto que es muy fácil de configurar y extender para guardar los logs en casi cualquier formato posible, desde archivos de texto, pasando por envíos de email, hasta guardar en base de datos.

La alternativa a este framework es la facilidad de log incluida en la API de Java desde la versión 1.4 en adelante. Esta API es, en resumen, una copia de la de log4j, con la diferencia de que no cuenta con toda la versatilidad que log4j permite para guardar los logs. Su ventaja por sobre log4j sólo radica en que viene integrada en la distribución de Java.

Se decidió usar log4j como framework para log porque todas las otras componentes seleccionadas, y las componentes en las que a su vez dependen, lo utilizan directamente o lo soportan mediante algún tipo de configuración mínima. También se privilegió la gran cantidad de posibilidades que soporta para guardar los logs.

2.3 Versión de las componentes utilizadas

El criterio para elegir la versión de las componentes fue seleccionar la versión estable más reciente que estuviera disponible en el momento de introducirla en el desarrollo. Sin embargo, nuevas versiones aparecieron a medida que se desarrolló el trabajo y según el nivel de cambios presente en las nuevas versiones es que se decidió la factibilidad de realizar la actualización o no. En general, actualizar a las nuevas versiones no revistió gran problema, ya que la dependencia del framework en ellas es relativamente baja.

Las componentes principales utilizadas y sus versiones son:

- SpringFramework Core en su versión 2.5.6
- Hibernate en su versión 3.3.1
- CXF en su versión 2.0.9
- Log4j en su versión 1.2.15
- JDOM en su versión 1.1
- JUnit en su versión 3.8.1

Durante el desarrollo de la integración del componente CXF en su versión 2.0.7, se detectó un problema de compatibilidad con los productos Oracle. El problema impedía que desde un cliente generado con CXF se pudiera invocar un servicio disponibilizado por Oracle. Esta incompatibilidad fue reportada por el autor junto con una solución primaria al equipo de desarrollo de CXF. El equipo de desarrollo aceptó la propuesta y la añadió a contar de la versión 2.1.1. Sin embargo, la línea de versiones 2.1.x de CXF introdujo una nueva API, nuevas dependencias y cambios en la configuración, con respecto a las versiones 2.0.x. Con esto en consideración, fue que se decidió seguir usando las versiones 2.0.x ya que cumplen con el propósito de este trabajo, y se cuenta con una solución propia para el problema de la incompatibilidad con Oracle.

2.4 Diseño e implementación de cada requerimiento

En esta sección se estudiará el diseño e implementación de cada requerimiento. En cada uno de ellos se presentará una sección de diseño, luego una de implementación y finalmente una sección de ejemplos de uso. En algunos casos la sección de ejemplos incluye comparativas con el framework anterior.

2.4.1 Requerimientos 1 y 2: La aplicación y su estructura deben ser configurables

2.4.1.1 Diseño

Estos requisitos fueron el punto de partida para comenzar el diseño y desarrollo del framework. Lo primero que se hizo fue identificar cuáles son las necesidades básicas que requiere una aplicación para ejecutarse. Luego se procedió a identificar cuales requisitos se ven involucrados en estas necesidades y finalmente se prosiguió con el desarrollo concreto de los demás requisitos que no fueron involucrados en este primer paso. Del anterior análisis resultaron las siguientes tareas y necesidades:

- Cargar la configuración de la aplicación.
- Configurar la estructura de la aplicación.
- Aplicar los cambios necesarios según la configuración.
- Se debe contar con un punto de acceso para obtener la configuración cargada.
- Se debe contar con un punto de acceso para obtener una referencia a las instancias de las clases configuradas en el contenedor.
- Se debe contar con una excepción propia del framework, que permita generar una jerarquía de excepciones a medida que se construya el framework.

En base a lo anterior se desarrollaron las siguientes clases:

- **Context**: es la encargada de realizar y proveer las tareas y los servicios identificados.
- **InitServletContextListener**: responsable de llamar a la clase `Context` para que inicie la carga de la configuración estándar cuando se está en un entorno Web y de cargar el contenedor web.
- **DyboxException**: es la excepción no chequeada[2] que será la base de la jerarquía de excepciones.

[2] Las excepciones en Java pueden ser chequeadas y no chequeadas. Las chequeadas son las que se deben declarar como parte de la interfaz o firma de una función, mientras que las no chequeadas no requieren de ser declaradas y pueden ser lanzadas en cualquier lugar de la aplicación. Dos excepciones comunes chequeada y no chequeada son IOExcepcion y NullPointerException respectivamente.

- **AutoSpringContextConfigurer**: es la encargada de incluir automáticamente dentro del contenedor las clases que se le indiquen.

La clase Context

Esta clase se diseñó usando el patrón Singleton para que sea el punto de acceso único a la configuración. De esta forma, se asegura que exista una única instancia de esta clase en la aplicación y por ende que se levante una sola vez la configuración. Es requisito fundamental que la inicialización de esta clase se produzca antes que la aplicación comience a funcionar de forma normal. Asimismo se asegura que la configuración sea cargada y llevada a cabo antes que el resto de la aplicación comience a funcionar. Si no se cumple con este requisito, la aplicación quedará en un estado impredecible, con todo lo malo que esto significa.

Las tareas principales que realiza en orden de ejecución son:

- Resolver la ruta desde la que se está ejecutando la aplicación.
- Leer el archivo de configuración `application.properties` y fijar las variables correspondientes.
- Configurar log4j en primera instancia para que esté disponible para las demás componentes que lo utilizan.
- Configurar el contenedor de SpringFramework.
- Configurar la herramienta de log del framework.

Al ejecutar estas tareas se termina con la carga de la configuración realizada, con la aplicación de la configuración según sea necesario y con el contenedor ejecutándose. Si cualquiera de estas tareas falla, entonces se arroja una excepción DyboxException con un mensaje descriptivo del error y como causa de la excepción se le asigna la excepción que originó el problema. De esta forma, el desarrollador puede identificar fácilmente cuál es el problema que ocasionó la falla al leer el mensaje de la excepción y luego ver la traza de la pila de cada excepción encadenada como causa.

La configuración básica de la aplicación se realiza mediante el archivo `application.properties`, el que corresponde a un archivo del tipo Properties [45] de Java.

Este tipo de archivo almacena en cada línea una dupla llave valor donde ambas partes de la dupla son strings. Es en este archivo donde cada aplicación deberá agregar las configuraciones simples que requiera.

Las propiedades que se han predefinido son: `app.production`, `app.context.load`, `app.log.path` y `app.context.auto_configurer_rules`. Éstas y todas las demás propiedades que se definan en este archivo, pueden ser accedidas desde la aplicación usando las funciones `getProperty(String name)` o `getProperties` de esta clase.

La propiedad `app.production`

La propiedad `app.production` especifica si la aplicación está en modo producción o no. El modo producción implica que todas las funciones del framework deben cambiar su comportamiento por uno en el que se privilegie la velocidad y eficiencia cuando esto sea posible. El valor booleano de esta propiedad es guardado en una variable de instancia de esta clase para que sea consultado directamente usando la función `isInProduction`.

La propiedad `app.context.load`

Para permitir una mejor capacidad de configuración de la aplicación se diseñó una funcionalidad que permite tener varias configuraciones posibles de la aplicación y en el momento de la inicialización se elige con cual de ellas trabajar. La cantidad de configuraciones posibles son ilimitadas, pero para partir se definieron dos, las cuales son `local` y `deploy`. La primera es para el desarrollo local y la segunda para cuando sea cargada en el servidor. La propiedad `app.context.load` se usa para especificar qué configuración cargar. Si no se especifica, entonces se carga `local` por defecto.

La estructura de directorios esperada es que exista un directorio raíz llamado `context` con subcarpetas nombradas de acuerdo a cada set de configuración. Todos los archivos de configuración que estén directamente bajo `context` serán comunes a todos los sets y los que estén dentro de cada subcarpeta serán los específicos al set. Este tipo de configuración permite además que la configuración común sea redefinida dentro de los sets específicos entregando una gran versatilidad.

Las demás propiedades se detallarán en las secciones en donde se analice las funcionalidades que las involucran.

La clase InitServletContextListener

Esta clase implementa la interfaz `ServletContextListener`, la que provee dos funciones: `contextInitialized` y `contextDestroyed`.

Estas funciones son llamadas cuando ocurren los eventos respectivos en el ciclo de vida de la aplicación web dentro del servidor de aplicaciones. La especificación de Java garantiza que estas funciones son llamadas una sola vez en el ciclo de vida de la aplicación. El orden en el que ocurre el llamado de estas funciones es el esperado, primero `contextInitialized` cuando la aplicación esta iniciando y luego `contextDestroyed` cuando la aplicación está siendo apagada o descartada. Esta clase se encarga de que la clase `Context` sea inicializada antes que el resto de la aplicación y también se encarga de proveerle la ruta de ejecución de la aplicación. Esto último es necesario porque en un servidor de aplicaciones Web, la única forma de determinar correctamente cuál es la ruta de ejecución de la aplicación es a través de una instancia de la clase `ServletContext`. Dicha instancia sólo puede ser obtenida dentro de funciones preestablecidas por la API de Java para aplicaciones Web. Una de estas funciones es precisamente `contextInitialized` de la interfaz `ServletContextListener` que recibe un evento el cual tiene una referencia a un `ServletContext`.

Los contenedores de SpringFramework permiten ser asociados mediante la relación padre/hijo. La visibilidad de una clase configurada se da desde el hijo al padre. De tal forma, se puede tener un contexto padre con la configuración común y varios contextos hijos que van a tener clases configuradas con referencias a las que están en el padre. Capacidad se usó en el diseño de los contenedores para que el contenedor que inicializa la clase `Context` actúe como padre y el contenedor Web actúe como hijo. Así, se enfatiza en el uso de capas claramente delimitadas en su visibilidad. Todas las clases que se configuren dentro del contenedor Web deben pertenecer a la capa de presentación o a la de Servicios

Web y las que se configuren en el contexto normal deben ser de la capa lógica y de acceso a datos.

Para configurar el contenedor Web de SpringFramework, se requiere usar la clase `ContextLoader` que es propia de este framework. Esta clase permite especificar el padre para el contenedor web que se está configurando. Como el `InitServletContextListener` se encarga de primero inicializar la clase `Context`, entonces ya se cuenta con una referencia que será pasada como padre para el contenedor Web que se está inicializando.

En esta clase se aplica el mismo criterio empleado en la clase `Context` de lanzar una `DyboxException` en caso de que algo inesperado suceda en el proceso de inicialización.

La clase AutoSpringContextConfigurer

El contenedor de SpringFramework se configura usando archivos XML. En ellos se especifica cada clase que se desea que el contenedor administre su ciclo de vida y la inyección de sus dependencias. A cada elemento que se configura en el contenedor se le denomina "bean". Cada bean debe ser identificado de forma única dentro de un archivo de configuración xml. De tal manera se puede contar con varios beans configurados de forma diferente, pero del mismo tipo. Un ejemplo de esto es el configurar varios beans de una clase que implemente la interfaz `javax.sql.DataSource` para tener acceso a distintas bases de datos.

Para reducir la complejidad de esta configuración, se generó la convención de nombrar las clases de la capa lógica y de acceso a datos usando el nombre en plural de la entidad principal que afecta. Ejemplo del uso de la convención sería que en una aplicación donde existe la entidad Empleado, se creen las clases `EmpleadosApp` y `EmpleadosDAO`. `EmpleadosApp` debe requerir usar al menos a la clase `EmpleadosDAO`, por lo que tendrá una propiedad llamada `empleadosDAO`. Así, la configuración de ambas clases quedará de la siguiente forma en el archivo xml del contenedor:

```
<!—El nombre o id del bean se extrae del nombre de la clase-->
<bean id=”empleadosApp” class=”cl.company.app.EmpleadosApp”>
      <property name=”empleadosDAO” ref=”empleadosDAO” />
```

```
</bean>
< bean id=” empleadosDAO” class=”cl.company.dao. empleadosDAO” >
      ... otras propiedades
</bean>
```

La administración de esta configuración se puede volver un gran problema para una aplicación que tiene muchas clases. Al respetarse la convención acordada, el trabajo de configurar cada clase App con su respectivo DAO se vuelve repetitivo. Esto puede causar que se cometan errores al usar la técnica de copiar y pega para crear un nuevo bean en base a uno ya existente. Con esta herramienta se busca evitar este problema al configurar automáticamente cada bean que cumpla con la convención.

Esta herramienta se diseñó para cumplir el propósito más general de incluir cualquier tipo de clase que se le indique dentro del contenedor. Esto permite que sea usada para incluir otras clases que no cumplan con la convención pero que de todas formas se quieren configurar por este medio.

El algoritmo que se diseñó para buscar e incluir clases en el contenedor recibe como parámetros de entrada las reglas de inclusión/exclusión de clases y un contenedor. Las reglas permitidas son de la siguiente forma:

- Un string que represente el nombre completo de una clase. Ejemplo: `cl.dybox.company.app.EmployeesApp`
- Un string que represente parcialmente el nombre de una clase usando el carácter “*”. Ejemplo: `cl.dybox.company.app. *App`
- Un string que represente un package usando el carácter “*”. Ejemplo: `cl.dybox.company.app.*`
- Un string que represente una raíz de packages usando la dupla de caracteres “**”. Ejemplo: `cl.dybox.company.**`
- Cualquier combinación de las anteriores. Ejemplo: `cl.dybox.**.Employees*`
- Cualquier string de los anteriores que comience con un carácter “!” para indicar que es una regla de exclusión. Ejemplo: `!cl.dybox.company.**Test`
- Toda regla de inclusión debe comenzar por un carácter distinto a “*”.

Los pasos que sigue el algoritmo se indican a continuación:

- Primero, se separan las reglas en 2 grupos, las que son de inclusión y las de exclusión. Las de exclusión se reconocen por comenzar con un signo de exclamación. Esta idea nace de cómo se niega una expresión booleana en el lenguaje Java. El algoritmo no es sensible al orden en el que se le entreguen las reglas, por lo que siempre que se entregue la misma permutación de reglas el resultado será el mismo.
- Luego, por cada regla de inclusión se obtienen todos los recursos que calcen con ella y que son visibles por la aplicación. Si ningún recurso es encontrado, entonces se envía al log un mensaje de alerta indicando que la regla no es utilizable. De esta forma, se espera que el programador se informe y modifique la regla.
- A continuación, por cada recurso obtenido se verifica que no calce con ninguna regla de exclusión, no sea una clase interna[3], no sea una prueba de unidad[4] y que no sea un directorio. Si no ocurre ninguno de los casos verificados, entonces la clase es agregada a la lista de candidatos a ser configurados en el contenedor.
- Finalmente, se recorre la lista de candidatos y se intentan añadir al contenedor usando la convención de nombrar al bean usando el nombre simple de la clase partiendo con minúscula. Una clase candidata será añadida al contenedor solamente si no existe previamente en el contenedor un bean configurado con el mismo nombre. De tal forma se le da precedencia a la configuración fijada en los archivos propios de configuración del contenedor por sobre la configuración automática.

Este proceso se integra dentro de la tarea "Configurar SpringFramework" que realiza la clase `Context`. Las reglas de inclusión/exclusión son leídas desde la propiedad `app.context.auto_configurer_rules` en donde son especificadas separadas por coma. Ejemplo:

```
app.context.auto_configurer_rules=cl.com.app.*,!cl.com.app.SucursalesApp
```

[3] Las clases internas son aquellas que se definen dentro de otra clase. Su uso suele estar ligado al funcionamiento interno de la clase en donde está definida. Se reconocen porque en el nombre llevan el carácter "$" de la forma: ClaseNormal$ClaseInterna.

[4] Para las pruebas de unidad se usa el estándar provisto por JUnit de nombrar las clases terminándolas en Test. Ejemplo: EmpleadosAppTest

La figura siguiente muestra el diagrama de secuencia de inicialización del framework cuando es iniciado dentro de un contenedor web.

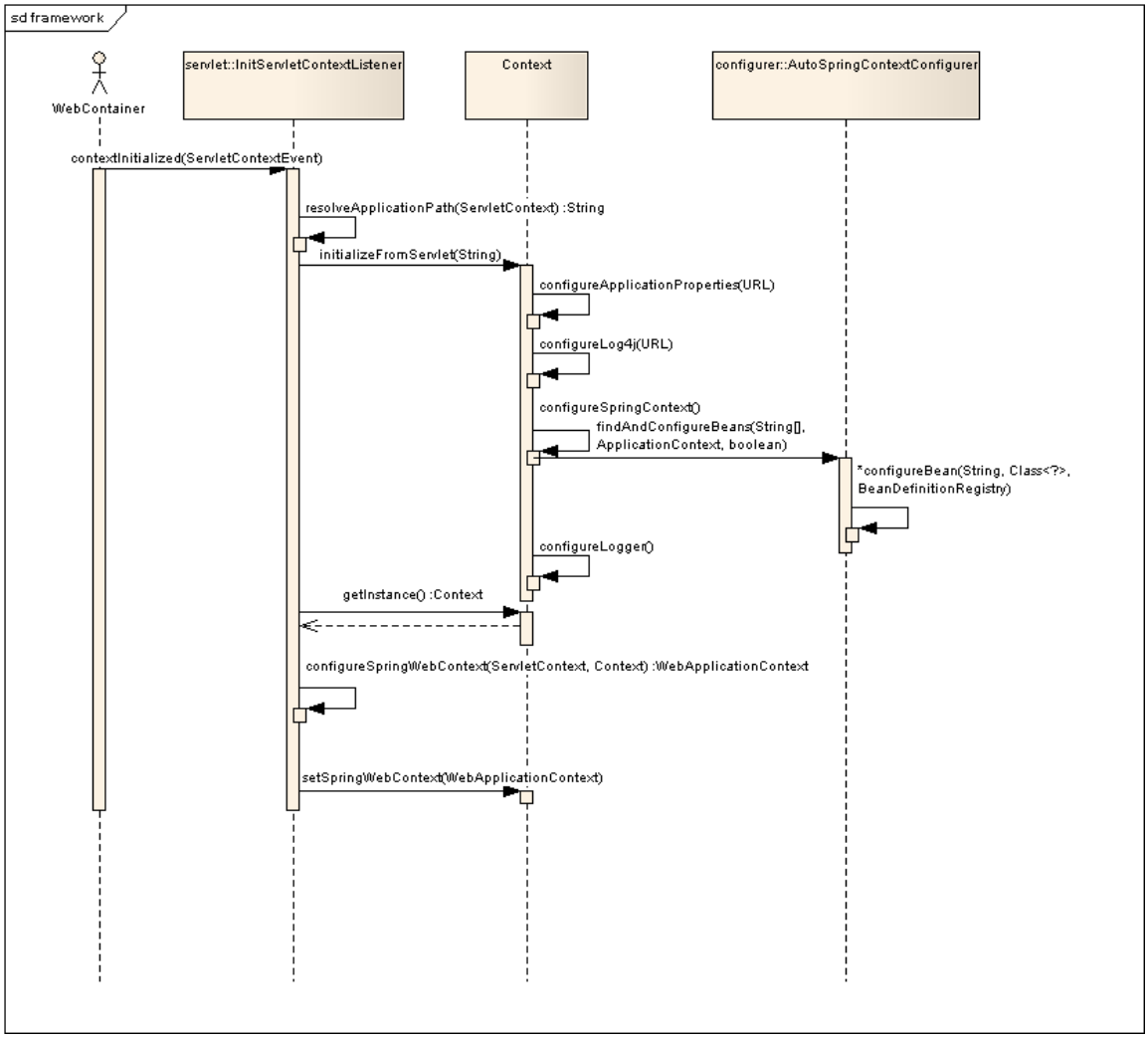


Figura 1: Diagrama de secuencia de inicialización del framework en un contenedor Web.

2.4.1.2 Implementación

La clase Context

Para implementar el patrón Singleton se usó la implementación propuesta por Joshua Fox [46]. Esta implementación garantiza que el patrón se cumpla en un entorno con más de un hilo de ejecución y que sólo se cree la instancia en el momento que se necesita. A este modelo se le agregó la verificación de que no se vuelva a intentar construir la instancia cuando este proceso falló previamente. El código siguiente muestra sólo las líneas relevantes a la implementación del Singleton.

```
public final class Context{
	private static Context instance = null;
	private Context(){}
	public static synchronized Context getInstance() {
		if (isBadInitialized()) {
			throw new IllegalStateException("The initialization of the
					Context failed before.");
		}
		if (instance == null) {
			instance = new Context(null);
		}
		return instance;
	}
}
```

Otro problema importante es la determinación de la ruta donde la aplicación se está ejecutando. La tarea es fácil cuando se trata de una aplicación que se ejecuta dentro de una JVM para ella sola, ya que sólo se necesita preguntar por el valor de la variable de entorno `user.dir`. Este caso también se da cuando se ejecuta una prueba de unidad la cual se ejecuta en una JVM propia. Sin embargo, cuando se ejecuta en un entorno donde muchas aplicaciones comparten la misma JVM, el valor devuelto por la variable `user.dir` ya no sirve. Este caso se da cuando la aplicación se ejecuta como aplicación Web en un contenedor de aplicaciones para este propósito. La solución es usar la función `getRealPath(String path)` de la clase `ServletContext` con el argumento "`/`". Entonces se creó la función `initializeFromServlet(String applicationPath)` para permitir que la clase `InitServletContextListener` entregue la ruta de ejecución de la aplicación**.** El siguiente código muestra esta función:

```
public static synchronized void initializeFromServlet(String
                                                    applicationPath){
	if(!Reflection.getCallerClass(2).getName().equals(
                    InitServletContextListener.class.getName())){
		throw new IllegalAccessError("this method can only be called by
		the class: " + InitServletContextListener.class.getName());
	}
	if (instance == null && !isBadInitialized()) {
		initFromServlet = true;
		instance = new Context(applicationPath);
	}
}
```

Esta función primero verifica que la clase que la llamó sea efectivamente `InitServletContextListener` para evitar que esta función sea mal utilizada.

La implementación de la configuración por niveles se realizó basándose en cómo SpringFramework carga los archivos de configuración. La carga se realiza exactamente en el orden que se le dan a los archivos de configuración, cumpliendo con que si en un archivo posterior se declara un bean con un nombre igual al de uno anterior, entonces se sobrescribirá el bean anterior por el posterior. Entonces, de lo que se preocupa esta clase es de entregar en el orden correcto los archivos de configuración; primero, los que están dentro de la carpeta `context` y luego, los que están dentro de la carpeta seleccionada mediante la propiedad `app.context.load`. Para evitar tener que configurar uno a uno los archivos a ser leídos, se fijó la convención de que los nombres deben terminar en `Context.xml`. Así, los siguientes archivos serán leídos: `applicationContext.xml`, `daoContext.xml`, entre otros, mientras que estos no serán leídos: `revisarContextNo.xml`, `errorcontext.xml`, `mal_context.xml`.

Será de responsabilidad del programador preocuparse de que no se repitan los nombres de los beans en los archivos que estén dentro de una misma carpeta para evitar la sobreescritura accidental. Para esto se propone que los archivos sean separados según el propósito que cumplan los beans que serán definidos en él. Ejemplo de esto sería definir los beans de la capa DAO en un archivo llamado `daoContext.xml`, los que prestan servicio de mail en el archivo `mailContext.xml`, y así sucesivamente.

La clase InitServletContextListener

Lo más complejo de la implementación de esta clase fue averiguar cómo entregarle al contenedor web, que esta clase debe crear, la referencia del contenedor ya creado por la clase Context como padre. Para esto se investigó el funcionamiento de la clase `org.springframework.web.context.ContextLoaderListener`. Esta clase es la que provee SpringFramework para cargar el contenedor automáticamente. Por este motivo fue que se eligió como punto de partida para investigar. El resultado de la investigación fue aprender que esta clase delega su trabajo a la clase `org.springframework.web.context.ContextLoader`. Al comprender como funciona esta clase, se llegó a la conclusión de que la forma más fácil de entregar un contenedor como padre a un contenedor Web es creando una clase que extienda de `ContextLoader` y que

sobrescriba dos funciones de ésta, `loadParentContext` y `closeWebApplicationContext`. Esta nueva clase se creó como una clase interna ya que no va a ser reutilizada. El siguiente código muestra la función `contextInitialized` de la clase `InitServletContextListener`.

```
public void contextInitialized(ServletContextEvent servletContextEvent) {
	ServletContext servletContext;
	final Context context;
	String modulePath;
	Logger.infoMessage("InitServletContextListener initializing...");
	// Get servlet context
	servletContext = servletContextEvent.getServletContext();
	try {
		// get real path to application
		modulePath = servletContext.getRealPath("/");
		if(modulePath != null && !modulePath.endsWith(File.separator)) {
			modulePath = modulePath + File.separator;
		}
		// Do init stuff
		Context.initializeFromServlet(modulePath);
		context = Context.getInstance();
		// Create and Set springWebContext
		springContextLoader = new ContextLoader(){
			protected ApplicationContext loadParentContext (
				ServletContext servletContext){
				return context.getSpringContext();
			}
			public void closeWebApplicationContext (
				ServletContext servletContext){
				//Don't do anything. The close is made by
					InitServletContextListener.contextDestroyed
			}
		};
		WebApplicationContext webContext =
		springContextLoader.initWebApplicationContext(servletContext);
		context.setSpringWebContext(webContext);
	} catch (Exception e) {
		throw new RuntimeException("InitServletContextListener failed to
		initialize", e);
	}
	Logger.infoMessage("InitServletContextListener ready");
}
```

La clase AutoSpringContextConfigurer

Para implementar esta clase se usó las herramientas provistas por SpringFramework para inspeccionar el `classpath` mediante el uso de las expresiones AntPath. Las expresiones AntPath, son expresiones que permiten identificar un recurso mediante el uso del carácter "*". Podemos ver por ejemplo, la expresión `/resources/**/btn*.jpg` que

simboliza todas las imágenes con extensión `jpg` y que comienzan por `btn` que se encuentren dentro del directorio y los subdirectorios de `resources`.

La clase de SpringFramework `PathMatchingResourcePatternResolver` permite obtener una lista de todos los archivos que calcen con una expresión AntPath. Esta clase es la base de la implementación de `AutoSpringContextConfigurer`. Dado que esta clase no es la única que necesita encontrar recursos, se creó la clase utilitaria `ResourceUtils` que permite encontrar recursos y entregarlos en distintos formatos.

A continuación se mostrará parte de la función estática `findAndConfigureBeans` que es el punto de entrada usado por la clase `Context` para generar la auto configuración. Se decidió usar funciones estáticas porque esta clase no necesita manejar estados.

```
public static void findAndConfigureBeans(String[] packages,
                            BeanDefinitionRegistry beanDefinitionRegistry,
                                                    boolean overrideContextBeans) {
        Map<String, String> beanClasses = new HashMap<String, String>();
        List<String> inclusionRules = new ArrayList<String>();
        List<String> exclusionRules = new ArrayList<String>();
        ....
        //Aqui se separan las reglas de inclusion/exclusion en dos listas

        for (String rule : inclusionRules) {
                boolean concreteClass = ! rule.endsWith("*");
                boolean deepSearch = rule.endsWith("**");
                String initialPackage = rule.substring(0,
                        rule.indexOf("*") != -1 ? rule.indexOf("*") :
                                                    rule.length());
                while (initialPackage.endsWith(".") || initialPackage.endsWith("*"))
                        initialPackage = initialPackage.substring(0,
                                                    initialPackage.length() - 1);
                String initialFilePackage = "/" + initialPackage.replace('.', '/');
                List<URI> beanClassURIs = ResourceUtils.getURIResources("classpath*:" +
                                            rule.replace('.', '/') +
                                            (concreteClass ? ".class" :
                                            (deepSearch ? "/*.class" : ".class")));
                if(beanClassURIs.isEmpty()){
                        Logger.warnMessage("This rule doesn't match with anything: "
                                + rule);
                }
                for (URI uri : beanClassURIs) {
                        ....
                        //Aquí se revisa que el recurso obtenido no haga calce con
                        //ninguna regla de exclusión y se añade al Map beanClasses
                }
                for (Map.Entry<String, String> entry : beanClasses.entrySet()) {
                        configureBean(entry.getKey(), entry.getValue(),
                                        beanDefinitionRegistry, overrideContextBeans);
                }
```

```
	}
```

El último ciclo se encarga de llamar a la función `configureBean` que es la que efectivamente configura el bean dentro del contenedor.

```
private static void configureBean(String beanName, String beanClassName,
	BeanDefinitionRegistry beanDefinitionRegistry, boolean override) {
	String debugMessage = "Bean: " + beanName + ", beanClass: " +
                                                        beanClassName;
	if(beanDefinitionRegistry.containsBeanDefinition(beanName)){
		if(override){
			debugMessage = "Overriding " + debugMessage;
		} else {
			throw new DyboxException("Bean already exists and override
                                        is false. " + debugMessage);
		}
	}
	Logger.debugMessage(debugMessage);
	BeanDefinition beanDefinition;
	BeanDefinitionBuilder bdb;
	bdb = BeanDefinitionBuilder.rootBeanDefinition(beanClassName);
	bdb.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_NAME);
	beanDefinition = bdb.getBeanDefinition();
	beanDefinitionRegistry.registerBeanDefinition(beanName,
                                                    beanDefinition);
}
```

Esta función usa la capacidad del contenedor de SpringFramework de auto conectar los bean, que se configuran en él. Esta capacidad se puede configurar para que funcione mediante la conexión por tipo o por nombre. Según el diseño para esta clase se usó la conexión por nombre. Esto se ve en la llamada a la función `setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_NAME)`.

2.4.1.3 Ejemplos

La clase Context

Los siguientes ejemplos muestran como usar la clase `Context` dentro de una aplicación.

- Como obtener un bean desde una clase que no ha sido configurada dentro del contenedor.

```
SomeClass someClass = Context.getInstance().getBean(“someClass”,
                                    SomeClass.class);
```

- Como obtener una propiedad configurada en el archivo `application.propeties`

```
String property = Context.getInstance().getProperty("property.name",
                                    "defaultValue");
boolean booleanProperty = Context.getInstance().getProperties().
                                    getPrimitiveBolean("boolean_property");
```

La clase InitServletContextListener

El siguiente ejemplo muestra cómo agregar esta clase a la configuración de la aplicación web para que levante el contenedor web de SpringFramework.

Archivo /WEB-INF/web.xml

```
<context-param>
      <description>Specifies the Spring configuration files to be read
                                                                  </description>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/*Context.xml</param-value>
</context-param>

<listener>
      <listener-class>
            cl.dybox.framework.web.servlet.InitServletContextListener
      </listener-class>
</listener>
```

La clase AutoSpringContextConfigurer

Esta clase es usada internamente por la clase `Context` para autoconfigurar el contenedor en base a las reglas especificadas con la propiedad `app.context.auto_configurer_rules` del archivo `application.properites`. Sin embargo, esta clase puede ser usada desde la aplicación para configurar en demanda otros beans. El siguiente ejemplo muestra este caso suponiendo que la lista de reglas es provista por algún servicio de datos en una lista de strings.

```
List<String> rules =  getRulesFromDataService();
boolean overrideContextBeans = false;
AutoSpringContextConfigurer.findAndConfigureBeans(
                        rules.toArray(new String[0]),
                        Context.getInstance().getSpringContext(),
                        overrideContextBeans);
```

2.4.2 Requerimiento 3: Herramienta de log

2.4.2.1 Diseño

Para generar el diseño de esta herramienta se incluyó el análisis del código de aplicaciones realizadas con el framework anterior, con el fin de mejorar el diseño de esta herramienta. El resultado del análisis realizado fue el siguiente:

- El framework actual requiere que primero se obtenga una instancia de la clase log y por ende declarar una variable. Esto lleva a que se cometan dos errores comunes, tales como que se declare globalmente en la clase y no se use, o que se declare localmente en todas las funciones. La solución propuesta fue que las funciones de la clase que preste servicios de log sean estáticas. De esta forma, no se requiere declarar variables y se evitan los dos problemas anteriores.

- El framework actual requiere que en cada llamada al log se indique de dónde se está llamando para poder identificar la fuente en la salida del log, lo que lleva a que se declare una variable que contenga este valor. Generalmente, se expresa como el nombre de la clase más el nombre de la función de donde está llamando. Esta práctica se convierte en un problema al hacer uso de refactoring, con el que no se actualiza el valor de esta variable al modificar el nombre de la función o el de la clase. La solución propuesta fue que la herramienta de log se encargue de averiguar qué función y qué clase la llamó, evitando así tener que declarar esta variable y su posterior mantención.

En base a la suma de estos requerimientos se diseñaron las siguientes clases:

- **Logger**: es la que permite hacer uso de las distintas implementaciones de log desde la aplicación.
- **LogInterface**: interfaz que se debe implementar para cada tipo de log.
- **LogStdImp**: implementación de la interfaz `LogInterface` que permite el uso de la salida estándar y salida estándar de error para el envío de los mensajes de log.
- **Log4jImp**: implementación de la interfaz `LogInterface` que permite el uso del framework log4j para el envío de los mensajes de log.

La figura siguiente muestra el diagrama de clases asociado.

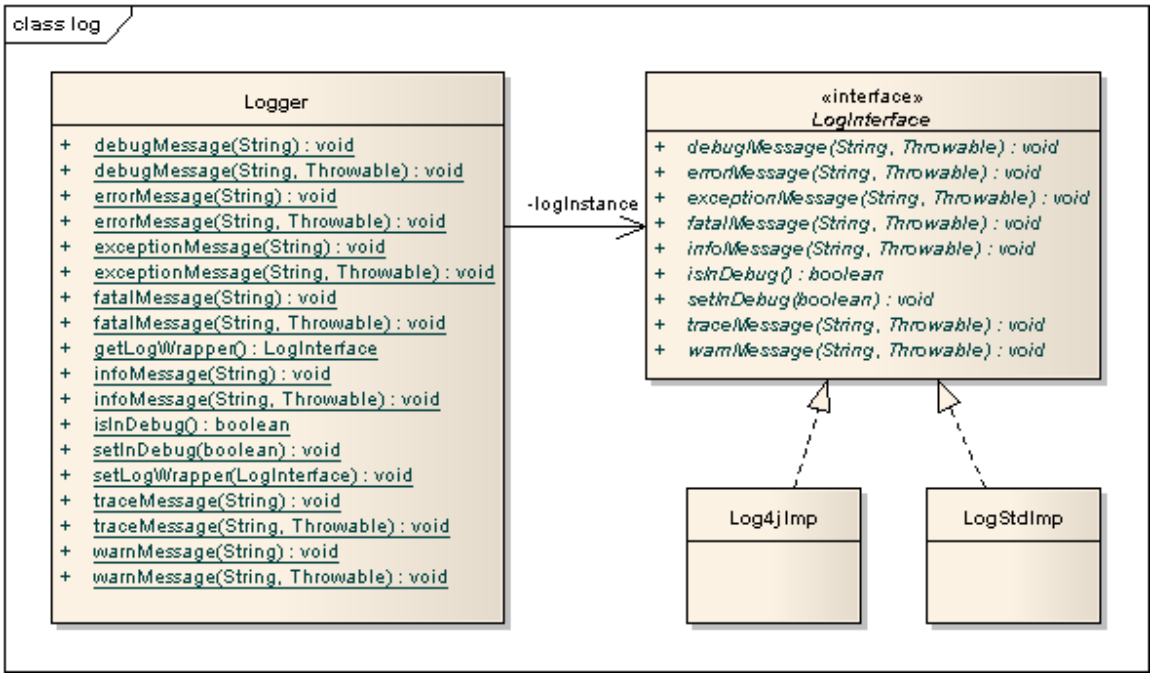


Figura 2: Diagrama de clases de la herramienta de log.

La clase Logger

Esta clase implementa el patrón Delegate[5]. Así esta clase delega todo su funcionamiento a la instancia de la interfaz `LogInterface` que se le haya configurado. Esta clase tiene las mismas funciones que define la interfaz `LogInterface`, pero con la variante de ser estáticas. De esta forma la aplicación no necesita mantener una variable para utilizar al log, simplemente usa las funciones estáticas de esta clase. La referencia a una implementación de `LogInterface` se inicializa por defecto a una instancia de `LogStdImp` para que siempre esté lista para ser usada. La forma de configurar cuál instancia usar es definiendo un bean en el contexto con el nombre "log". Este bean es buscado en la tarea "Configurar la herramienta de log del framework" de la clase Context. Si lo encuentra, entonces se reemplaza la instancia de `LogStdImp` por este bean.

[5] El patrón Delegate consiste en que un objeto delega la implementación de una determinada funcionalidad en otro, pero hacia el exterior mantiene su interfaz.

La interfaz LogInterface

Esta interfaz fue creada en base al uso del patrón Strategy para resolver el requisito de contar con implementaciones intercambiables. Este patrón permite desacoplar la implementación de la interfaz, pudiendo así intercambiar la implementación según sea necesario. Esta interfaz tiene funciones sobrecargadas que representan a cada tipo de mensaje. Existe una versión de la función que sólo recibe el mensaje y una segunda versión que además recibe una excepción. Cada implementación decidirá si requiere o no determinar quien la llamó y además ver qué hacer con la excepción que se le envía.

La clase LogStdImp

Es una implementación simple de la interfaz `LogInterface`, que según el tipo de mensaje determina si enviarlo a la salida estándar o a la salida estándar de errores. Los tipos que van a la salida estándar son: `trace`, `debug` e `info`. Los tipos que van a la salida estándar de errores son: `warn`, `exception`, `error` y `fatal`. En el caso de recibir una excepción, ésta se imprime a la salida correspondiente según el tipo de mensaje.

La clase Log4jImp

Es una implementación que ocupa al framework log4j para delegar el envío de mensajes. Los tipos de mensajes se parean uno a uno a los niveles de mensajes que define log4j. La excepción es el tipo `exception` que por no existir un par en log4j se pareó al nivel `error`.

Esta clase hace uso de la variable `production` del contexto para filtrar los mensajes. Así todos los mensajes que tengan un nivel inferior a `warn` son desechados de encontrarse la aplicación en modo producción.

Esta clase debe averiguar quien es el que la llama, ya que log4j requiere el nombre completo de una clase para obtener una instancia del objeto logger de este framework al que se le enviarán los mensajes.

La configuración de log4j permite el uso de variables globales para indicar la ubicación de los archivos generados. La siguiente línea de un archivo de configuración de log4j muestra este uso: `log4j.appender.rootLog.File=${app.log.path}/WEB-INF/logs/root.log`, donde la sección `${app.log.path}` será reemplazada por la variable global `app.log.path` que debe ser fijada antes que log4j se configure.

El valor por defecto que se le asigna a la variable global `app.log.path` es la ruta de ejecución de la aplicación. El valor de esta variable puede ser cambiado usando la propiedad `app.log.path` definida en el archivo `application.properties`. De esta forma es posible usar una ruta común para los logs de todas las aplicaciones, práctica común dentro de las empresas a las que Dybox les brinda servicios.

Sin embargo, el uso de una variable global en un entorno compartido, como es el caso de una aplicación Web, tiene la complejidad de que cada aplicación Web debería tener su propia variable para evitar la sobreescritura de la misma. Esto implica que el archivo de configuración de log4j debe ser personalizado para la aplicación usando el nombre de la variable global. Además este nombre debe ser conocido por la herramienta que configura el valor de la variable en tiempo de ejecución. Esto produce que sea otro punto más que requiere de mantención por parte del desarrollador, por lo que es un foco de posibles errores.

Para solucionar este problema se diseñó que se fije una variable global única para cada aplicación de forma dinámica. Luego el archivo de configuración de log4j se preprocesa cambiando el nombre estándar de la variable por el específico. Finalmente se entrega este archivo modificado al configurador propio de log4j. Gracias a este diseño, no es necesario modificar el archivo de configuración de log4j para cada aplicación. Por este motivo no se requiere definir ni mantener una variable específica. Esto permite poder tener un archivo de configuración de log4j que sea estándar para todas las aplicaciones. Todo este proceso ocurre dentro de la tarea “Configurar log4j” de la clase `Context`.

2.4.2.2 Implementación

En esta sección sólo se abordará la implementación de la clase `Log4jImp` por ser la de mayor complejidad.

La clase Log4jImp

Lo más complejo de abordar de esta clase fue la determinación de cuál clase y cuál función de ella fue lo que originó la llamada al log. Después de buscar en la API de Java y en Google se llegó a la conclusión de que la única forma estándar de hacerlo es mediante la obtención del `StackTrace`[6] de ejecución. La forma no estándar es usar la clase `sun.reflect.Reflection` que es propia de la implementación de Java. Esta clase permite averiguar sólo la clase del llamador mediante la función `getCallerClass(int i)` que recibe como parámetro el número del cuadro equivalente del `StackTrace.` Esta clase es usada extensamente dentro de la implementación de Java para realizar restricciones sobre el llamador de una función. Finalmente se optó por usar un esquema mixto, entre el uso de esta clase y el `StackTrace`. El uso del `StackTrace` fue requerido para saber el nombre de la función del llamador, información necesaria para ser agregada al mensaje a ser enviado al log.

El problema de usar el `StackTrace` radica en cómo obtenerlo de forma eficiente. La alternativa más directa es usar la clase Throwable que al instanciarla se encarga de poblar el `StackTrace` completo del hilo de ejecución actual dentro de ella. El problema con esta alternativa es que se vuelve muy lento a medida que la pila se hace más profunda. Esto ocurre cuando se recorre una estructura de árbol muy profunda o, en general, cuando se usa una función recursiva. Para dar solución a este problema, se investigó las posibles alternativas a la obtención de la pila. El fruto de esta investigación fue encontrar el package estándar `java.lang.management` que permite monitorear el estado de la VirtualMachine de Java. Con las clases de este package es posible obtener el `StackTrace` indicando la cantidad de cuadros que se desean. Así se soluciona el problema de la pila muy profundo, ya que

[6] El StackTrace es la traza de la pila de ejecución. Es un arreglo de Objetos del tipo StackElement donde cada uno de ellos representa un cuadro de la ejecución. Cada StackElement contiene información de la clase y la función que se ejecutó en ese cuadro.

siempre se pide una cantidad acotada de cuadros. La ejecución que permite obtener la pila es la siguiente:

```
long threadId = Thread.currentThread().getId();
int nFrames = 8; //max amount of frames to fetch from the stack
ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
StackTraceElement[] stackTraceElements = threadMXBean.getThreadInfo(threadId,
nFrames).getStackTrace();
```

Se comparó la eficiencia de obtener la clase llamadora usando esta forma versus usar la clase no estándar `sun.reflect.Reflection` y se obtuvo que usando la segunda es 3 veces más rápido. Entonces se hizo la optimización de obtener la clase llamadora mediante la segunda forma, con esta información se obtiene el `logger` de log4j asociado y se pregunta si el nivel de log que se desea usar está habilitado para este `logger`. De estarlo, se prosigue con obtener el `StackTrace` de la primera forma y obtener así el nombre de función llamadora para ser anexada junto con el mensaje a ser enviado. De no estarlo, se termina la ejecución y se ahorra el tener que pedir el `StackTrace`. El siguiente código muestra lo recientemente explicado.

```
String callerClassName = findCallerClassName();
org.apache.log4j.Logger logger =
      org.apache.log4j.Logger.getLogger(callerClassName);
//check if logger is enabled for this level
if(!logger.isEnabledFor(level)){
      return;
}

private String findCallerClassName() {
      Class<?> clazz;
      String className;
      //start from 2 because on 0 and 1 this is the callerClass.
      //clases are compared by name to avoid ClassLoaders related problems.
      for (int i = 2; (clazz = Reflection.getCallerClass(i)) != null; i++) {
            className = clazz.getName();
            if (!LOGGER_NAME.equals(className) &&
                                    !THIS_NAME.equals(className)) {
                  return className;
            }
      }
      return null;
}
```

Finalmente, el resultado obtenido fue satisfactorio al poder determinar de forma eficiente la clase y la función llamadora.

2.4.2.3 Ejemplos

El siguiente ejemplo expone el caso en que la clase originalmente pertenecía al package `cl.company.process`, pero después se aplicó refactoring y se cambió al package `cl.company.batch` dejando desactualizado el mensaje del log.

Versión antigua:

```
package cl.company.batch
Public class Process{
	public void processData(){
		LogManager logger = LogManager.getInstance();
		logger.debugMessage(“cl.company.process.Process.processData”,
“someMessage”);
	}
}
```

Resultado: `[cl.company.process.Process.processData]someMessage`

Versión nueva:

```
package cl.company.batch
Public class Process{
	public void processData(){
		Logger.debugMessage(“someMessage”);
	}
}
```

Resultado: `[cl.company.batch.Process.processData]someMessage`

Se puede ver la simpleza lograda en la nueva versión al disminuir la cantidad de líneas de código y parámetros necesarios. Además de entregar precisamente la función que envía el mensaje.

2.4.3 Requerimiento 4: Se debe evitar la creación de instancias de clases innecesariamente

2.4.3.1 Diseño

Para evitar que se creen instancias innecesarias de clases se debe configurar todo lo que sea posible dentro del contenedor de aplicación de SpringFramework. De esta forma se

delega al contenedor el manejo del ciclo de vida según cómo se configure cada bean. Cada bean tiene una propiedad denominada "scope" que determina el ciclo de vida del bean. Los scopes más comunes son: `singleton` y `prototype`. El primero indica que sólo se creará una instancia de este bean a lo largo del ciclo de vida de la aplicación. El segundo indica que se creará una nueva instancia cada vez que se pida este bean. Si un bean no indica esta propiedad en la configuración, entonces se supone que tiene scope `singleton`. Así, se tiene control total de cuáles clases se creará una sola instancia y de cuáles muchas instancias. En general, las clases de la capa lógica y DAO funcionan bien con el scope `singleton`, mientras que en la capa de presentación se da un mix entre `singleton` y las que requieren de `prototype` o alguno similar. Lo anterior se debe a que hay objetos que se asocian con el ciclo de vida de una sesión http o a uno más corto como es el caso de un `request`. De esta manera se puede evitar el uso de constructores por parte de la aplicación, delegando al contenedor la responsabilidad de entregar una instancia ya creada o una nueva del objeto requerido.

Para las clases que sean imposibles de configurar dentro del contenedor, se crearon las funciones `containsBean`, `getBean` y `getBeansOfType` en la clase `Context` para permitirles interactuar con él. Con estas funciones se puede consultar si existe un bean con un determinado nombre, obtener un bean por nombre y obtener todos los beans que sean compatibles con un determinado tipo.

2.4.3.2 Implementación

La implementación de las funciones `containsBean`, `getBean` y `getBeansOfType` se preocupan de delegar su funcionamiento al contenedor de SpringFramework que cuenta con estas mismas funciones. Aquí se usó el patrón Facade para evitar que la aplicación se infecte de código proveniente de SpringFramework innecesariamente. En ellas se atrapan las excepciones que arroja el contenedor y son envueltas en excepciones `BeanNotFoundDyboxException` agregando información extra en el mensaje.

El siguiente código muestra la implementación de la función `getBean`.

```
public <T> T getBean(String beanName, Class<T> beanClass, boolean throwException) {
	Object bean = null;
```

```
	try {
		if (getSpringContext() != null)
			bean = getSpringContext().getBean(beanName, beanClass);
	} catch (Exception e) {
		if (getSpringWebContext() == null) {
			String message = "Error when getting bean named: " + beanName +
													", from SpringContext.";
			if (throwException)
				throw new BeanNotFoundDyboxException(message, e);
			else
				Logger.warnMessage(message, e);
		}
	}
	try {
		if (bean == null && getSpringWebContext() != null)
			bean = getSpringWebContext().getBean(beanName, beanClass);
	} catch (Exception e) {
		String message = "Error when getting bean named: " + beanName +
													", from SpringContext.";
		if (throwException)
			throw new BeanNotFoundDyboxException(message, e);
		else
			Logger.errorMessage(message, e);
	}
	return (T) bean;
}
```

El parámetro `boolean throwException`, sirve para indicar si se desea que se lance o sólo se envíe al log la excepción producida al intentar obtener un bean. Esto permite diseñar aplicaciones que pueden requerir un bean con prioridad media, así si falla la obtención del bean, sólo se retorna `null` y se envía la excepción al log como un `warning` y la aplicación decidirá que hacer en este caso. Existe una sobrecarga de esta función que sólo recibe los 2 primeros parámetros. Esta versión sobrecargada llama a la versión de 3 parámetros con el último fijado a verdadero. Así, se logra que por defecto se lance la excepción si no se ha especificado lo contrario, generando el comportamiento de fallar rápidamente en caso de que exista algún problema con el bean. Este sería el caso de requerir el bean con alta prioridad.

2.4.3.3 Ejemplos

Los siguientes tres ejemplos muestran tres funciones de una clase que no ha sido configurada en el contenedor y que tienen el comportamiento de requerir un bean con prioridad alta, media y baja.

```
public void failFast(){
	SomeClass someClass = Context.getInstance().getBean("someClass", SomeClass.class);
	someClass.doSomething();
}
public void mediumPriority(){
	SomeClass someClass = Context.getInstance().getBean("someClass", SomeClass.class,
                                                                    false);
	if(someClass == null) {
		someClass = new SomeClass();
	}
	someClass.doSomething();
}
public void lowPriority(){
	SomeClass someClass;
	String beanName = "someClass";
	Context context = Context.getInstance();
	if (context.containsBean(beanName)) {
		someClass = context.getBean(beanName, SomeClass.class, false);
	} else {
		someClass = new SomeClass();
	}
	someClass.doSomething();
}
```

2.4.4 Requerimiento 5: Soporte para la publicación de una funcionalidad como Servicios Web

2.4.4.1 Diseño

Dado que para poder realizar esta funcionalidad se usó el framework CXF, el diseño asociado a este requerimiento sólo se remitió a definir el estándar de nombres a usar. Se definió que se crearía el package **ws** al mismo nivel que los de la capa lógica y de acceso a datos. Dentro de este package se almacenarán las interfaces e implementaciones de ellas para dar vida a cada servicio Web. De ser necesario, se crearán subpackages que agrupen servicios por algún criterio afín. Los nombres de las interfaces deben terminar en **WS** y las clases que las implementan en **WSImp**.

La forma de disponibilizar servicios que se usó fue usando el estándar JAX-WS [49]. Este método requiere que se genere una interfaz de servicio con Annotations propios del estándar y una implementación de esta interfaz que también debe llevar Annotations. Luego, se configura cada servicio conformado por la interfaz y el implementador dentro del archivo de configuración xml de CXF. Este archivo es de tipo de configuración de un contenedor de SpringFramework. CXF ocupa un contenedor Web de SpringFramework para su configuración. Se puede especificar en la configuración del servicio las referencias a los beans de la capa lógica directamente gracias a que es posible vincular un contenedor Web con uno normal. El proceso de vinculación se describe en el diseño de la clase `InitServlerContexListner`.

Se espera que la funcionalidad a disponibilizar como Servicio Web se encuentre implementada en la capa lógica de la aplicación. La implementación del Servicio Web sólo debe actuar como un comunicador entre el servicio y la capa lógica, preocupándose de las posibles conversiones de datos y del manejo de excepciones.

2.4.4.2 Implementación

La implementación de este requerimiento fue lo que motivó a que se buscara la forma de vincular el contenedor Web con el normal. Este vínculo permite evitar tener que agregar código extra en la implementación del servicio para obtener la instancia del bean de la capa lógica desde el otro contenedor.

CXF usa por defecto el log provisto por el JDK de Java. Como se decidió usar log4j como framework de log para las componentes, fue necesario buscar la forma de configurar CXF para que use log4j en vez del provisto por el JDK. Para este fin, CXF cuenta con un mecanismo de configuración que consiste en agregar un archivo llamado `org.apache.cxf.Logger` en el classpath. Este archivo debe contener en su primera línea el nombre de la clase completa que debe usar como log. Para indicarle que use log4j se debe escribir el valor `org.apache.cxf.common.logging.Log4jLogger`.

Para agregar CXF a una aplicación Web, se debe configurar un servlet en el archivo web.xml y que se cargue un contenedor Web de SpringFramework con su configuración.

2.4.4.3 Ejemplos

El siguiente ejemplo muestra todo lo necesario para disponibilizar una función presente en la capa lógica como Servicio Web.

Se debe agregar lo siguiente al archivo `web.xml`.

```
<servlet>
	<servlet-name>CXFServlet</servlet-name>
	<servlet-class>
		org.apache.cxf.transport.servlet.CXFServlet
	</servlet-class>
	<load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
	<servlet-name>CXFServlet</servlet-name>
	<url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

No es necesario agregar nada más a este archivo, porque se da por hecho que esta aplicación Web ya tiene configurado el uso de la clase `InitServlerContexListner` en él.

Se debe crear el archivo `cxfContext.xml` dentro la misma carpeta donde reside el archivo `web.xml` con el siguiente contenido.

```
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:jaxws="http://cxf.apache.org/jaxws"
	xsi:schemaLocation="
		http://www.springframework.org/schema/beans
			http://www.springframework.org/schema/beans/spring-beans.xsd
		http://cxf.apache.org/jaxws
			http://cxf.apache.org/schemas/jaxws.xsd">

	<import resource="classpath:META-INF/cxf/cxf.xml" />
	<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
	<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

	<bean id="standardServiceFactory"
	class="org.apache.cxf.jaxws.support.JaxWsServiceFactoryBean" scope="prototype">
		<property name="properties">
			<map>
				<entry key="faultStackTraceEnabled" value="true" />
			</map>
		</property>
	</bean>
```

```
	<!– De aquí en adelante se deben agregar los servicios -->

	<jaxws:server serviceClass="cl.company.ws.SomeServiceWS" address="/SomeService">
		<jaxws:serviceBean>
			<bean class="cl.company.ws.SomeServiceWSImp">
				<property name="someLogicApp" ref="someLogicApp" />
			</bean>
		</jaxws:serviceBean>
		<jaxws:serviceFactory>
			<ref bean="standardServiceFactory" />
		</jaxws:serviceFactory>
	</jaxws:server>
</beans>
```

Interfaz `cl.company.ws.SomeServiceWS`.

```
@WebService
public interface SomeServiceWS {
	@WebMethod(action="someMethod")
	String someMethod ( @WebParam(name="someParam")  String someParam );
}
```

Implementación `cl.company.ws.SomeServiceWSImp`.

```
@WebService(endpointInterface="cl.company.ws.SomeServiceWS")
public class SomeServiceWSImp {
	private SomeLogicApp someLogicApp;
	public String someMethod ( String someParam) {
		String ret = null;
		try{
			someParam = someParam.trim().toLowerCase();
			ret = getSomeLogicApp().someMethod(someParam);
		} catch (Exception e) {
			String message = “Exception excecuting someMethod logic call.”;
			Logger.exceptionMessage(message, e);
			ret = message + “ ” + StringUtils.getSimpleStackTrace(e);
		}
		return ret;
	}

	... //getters y setters de someLogicApp
}
```

En caso de producirse una excepción, se envía un mensaje al log y se construye una respuesta acorde a la situación. La función `getSimpleStackTrace` de la clase `StringUtils` entrega un string con los mensajes de todas las excepciones encadenadas como causa separados por el string `", caused by "`.

Finalmente, el servicio quedará disponible en la dirección `http://server:port/AppContext/services/SomeService` y su `wsdl` en `http://server:port/AppContext/services/SomeService?wsdl`.

2.4.5 Requerimiento 6: Capacidad de intercambiar la capa de presentación

2.4.5.1 Diseño

Como se expuso en la sección 1.1, el framework antiguo está muy acoplado al uso de XML/XSLT en la capa de presentación. Este acoplamiento se produce en varios puntos de la aplicación.

El primer punto de acoplamiento es que las clases que pasarán a la capa de presentación como entrada deben extender de la clase `XMLEntity` para que puedan ser convertidas a xml. Esta clase provee el método `getElement` que se encarga de transformar la instancia actual de la clase a un nodo `Element` del framework JDOM. La transformación a un nodo xml se realiza según la configuración específica para el tipo de la clase en un archivo de configuración. En el archivo de configuración se debe especificar la estructura xml que se desea generar para las instancias de la clase. Estos requerimientos necesarios para lograr que una instancia de una clase sea convertida a xml son foco de muchos errores durante el desarrollo. Los errores más comunes son no actualizar la configuración cuando la clase cambia, no hacer que la clase extienda de `XMLEntity`, o escribir mal el nombre de un atributo o nombre de función. Además de todo lo anterior, el convertidor a xml no detecta los ciclos, por lo que es responsabilidad del desarrollador generar una configuración que evite este problema. Lo anterior redunda en agregar más responsabilidad sobre el desarrollador y hacer más complejo el desarrollo. El archivo de configuración para la siguiente clase sería:

```
package cl.company.entities;
public class Employee {
	private long id;
	private int age;
	private String name;
```

```
        private String charge;
        private Employee bos;
        ... //funciones getters y setters
}
<entities>
        <entity class="cl.company.entities.Employee">
                <employee id="getId">
                        <name>getName</name>
                        <age>getAge</age>
                        <charge>getCharge</charge>
                        <bos>getBos</bos>
                </employee>
        </entity>
</entities>
```

Donde `getId`, `getName`, `getAge`, `getCharge` y `getBos`, hacen referencia a las funciones a invocar sobre la instancia de esta clase para obtener el valor que se le dará a cada elemento. Haciendo un análisis simple sobre el archivo de configuración, se puede ver que es totalmente factible generarlo automáticamente.

El diseño se basó en cómo evitar que: las clases requieran extender de `XMLEntity`, que se deba crear una configuración por cada clase y que el manejo de ciclos sea de responsabilidad del desarrollador. Para dar solución a estos tres problemas se comenzó por desacoplar el proceso de conversión a xml de las clases mismas moviendo este código a clases externas. Así, las clases a ser convertidas a xml ya no requieren extender de `XMLEntity`. Luego, se diseñó la forma de generar la configuración automáticamente mediante el uso de Reflection[7] y convenciones. De esta manera se evita tener que crear y mantener la configuración por cada clase. Finalmente, se diseñó la forma de detectar los ciclos en el proceso de conversión a xml. Todo esto derivó en la creación de las siguientes clases que implementan el patrón Abstract Factory.

- **SerializerFactory**: punto de entrada para obtener un serializador. Su función es crearlos, configurarlos y mantenerlos apropiadamente.
- **ObjectSerializer**: serializador abstracto que sirve de base para los demás serializadores.

[7] Reflection [47] es un conjunto de funciones y clases que provee Java para poder acceder a la información de un objeto sin conocer su tipo. En este caso, se ocupa Reflection para obtener una lista de todos los métodos de la clase del objeto y luego invocarlos sobre la instancia.

- **SimpleSerializer**: serializador específico para los tipos simples de Java. En este grupo quedan los strings, caracteres, números y fechas.
- **CollectionSerializer** y **MapSerializer**: serializadores específicos para todos los tipos que implementen las interfaces `Collection` y `Map` respectivamente.
- **BeanSerializer**: serializador específico para los objetos que son identificables como Java Beans.

La figura siguiente muestra el diagrama de clases asociado.

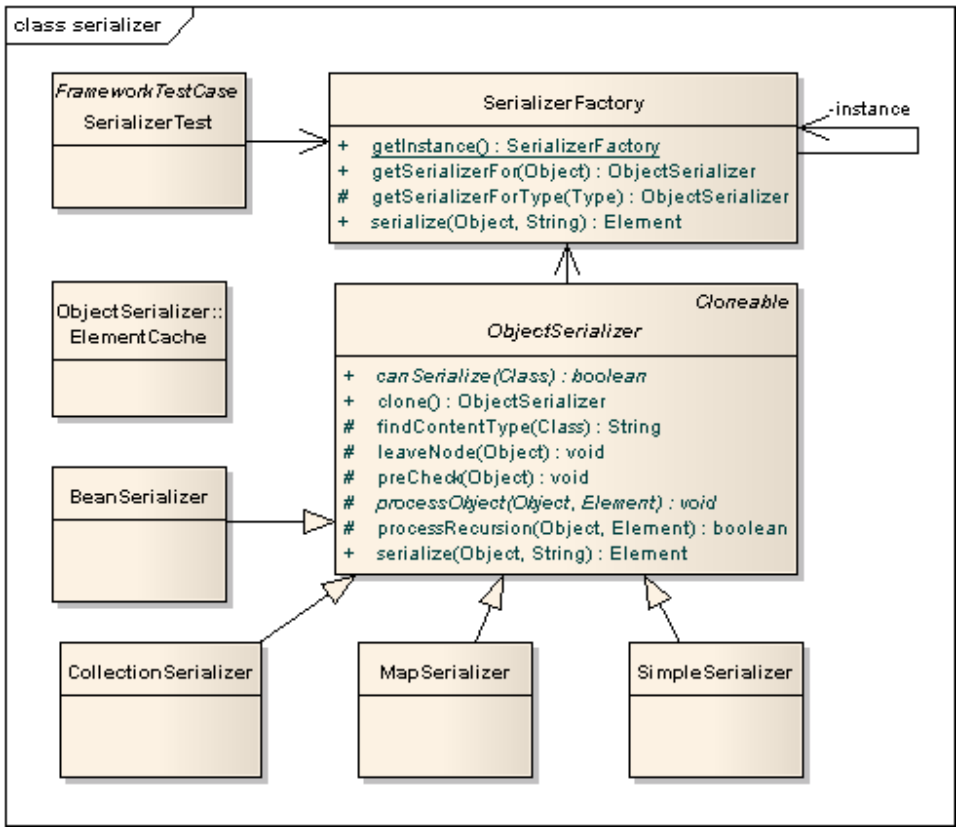


Figura 3: Diagrama de clases de la herramienta serializadora de Objetos a xml.

La clase SerializerFactory

Esta clase se diseñó usando el patrón Singleton. De esta forma, existirá una única instancia en toda la aplicación de esta clase. El proceso de selección del serializador puede ser alterado mediante configuración. Esta clase contiene una lista de serializadores

priorizados, los que serán consultados primero si pueden serializar la clase solicitada. Si un serializador responde afirmativamente, entonces será usado para realizar la serialización del tipo solicitado. Si ninguno responde afirmativamente o la lista está vacía, entonces se prosigue con el algoritmo normal de selección del serializador. De tal manera, es posible agregar nuevos serializadores y garantizar que serán usados cuando corresponda. Esta configuración puede ser realizada mediante el uso del contenedor definiendo un bean con el nombre "`serializerFactory`" y del tipo `SerializerFactory`. Otra alternativa es pedir la instancia de esta clase y setear la lista de serializadores. Esto último sólo funcionará correctamente si se hace antes de que se solicite un serializador por primera vez. Lo anterior se debe a que se mantiene un mapa entre clases y serializadores para no realizar la búsqueda y configuración en cada llamada.

El algoritmo de selección de un serializador para un tipo determinado sigue los siguientes pasos:

- Se busca en el mapa si ya existe una asociación entre un serializador y este tipo. Si se encuentra uno, entonces se termina la ejecución retornando el serializador.
- Se busca en la lista de serializadores priorizados uno que pueda serializar este tipo. Si se encuentra uno, entonces se genera una nueva instancia clonando al serializador actual. Luego, se registra la asociación en el mapa entre el clon y este tipo. Se le indica al serializador clonado que ejecute su proceso de configuración y finalmente se termina retornando el nuevo serializador.
- Se comprueba si puede ser serializado por un `SimpleSerializer`. Si se puede, entonces se genera un nuevo SimpleSerializer clonando al original. Se procede igual que en el caso anterior.
- Se comprueba si puede ser serializado por un CollectionSerializer o un MapSerializer. Si se puede, entonces se procede de igual forma que en el paso anterior.
- Finalmente, si no fue posible encontrar un serializador para este tipo, entonces será considerado como un JavaBean[8]. Entonces, se crea una nueva instancia de BeanSerializer con la que se procede a hacer lo mismo que en los pasos anteriores.

[8] Un JavaBean [48] es una clase que tiene variables de estado privadas para las cuales provee métodos de acceso públicos conocidos como `getters/setter` o accesores. Si la clase tiene una propiedad de tipo entero llamada `saldo`, entonces debe tener los métodos `void setSaldo(int value)` e `int getSaldo`.

De esta forma se garantiza que siempre se devolverá un serializador para el tipo solicitado, sin embargo, el uso del BeanSerialzer como serializador por defecto trae problemas con las clases que no respetan la definición de JavaBean. Este problema quedó abierto y debe ser resuelto en futuras versiones.

Para determinar si un serializador puede serializar un tipo determinado, se usa el método `boolean canSerialize(Class clazz)`. Este método fue definido abstracto en la clase `ObjectSerializer` de tal forma que todos los serializadores provean una implementación para él.

La clase ObjectSerializer

Esta clase posee la implementación del método central de los serializadores. Este método es `Element serialize(Object object, String elementName)`, el cual toma un objeto y lo convierte a una representación en xml mediante el uso del framework JDOM. Esta función se divide en varios métodos más pequeños, algunos de ellos abstractos, para que puedan ser sobrescritos o implementados por los serializadores y de esta forma alterar el resultado. Este patrón de diseño se conoce como Template Method[9]. Así, se evita tener que rescribir en cada serializador los pasos comunes a todos. Se crea una nueva instancia de un serializador por cada tipo a serializar para guardar información necesaria de él. De esta manera se evita la sobrecarga de obtener esta información cada vez que se quiera serializar una instancia de ese tipo. La información que se desea guardar del tipo es obtenida cuando se configura el serializador mediante la ejecución del método introspect. La información que se obtiene del tipo depende de cada serializador. La implementación provista por esta clase se encarga de generar un nombre por defecto para el tipo y crear un prototipo `Element` que se usará como base en cada serialización.

La clase SimpleSerializer

[9] El patrón Template Method [50] sirve para especificar un gran algoritmo que puede tener ciertas variaciones dependiendo del comportamiento de los subprocesos que requiera ejecutar. La combinación de las distintas variaciones de estos subprocesos, generará una nueva versión del algoritmo.

Para determinar si esta clase puede serializar un determinado tipo o no, se comprueba si es de tipo Simple. Se considerará a un tipo como Simple siempre y cuando su método `toString` retorne toda la información útil e indivisible de él. Algunos ejemplo de tipos simple son: `String`, `Long`, `Double`, `Character`, junto a sus tipos básicos `long`, `double`, `char`. El tipo `Date` no se considera simple, porque el retorno de su método `toString` es divisible en año, mes, día, hora, minutos, segundos, y aún más.

El elemento xml que devuelve este serializador tiene la forma:

```
<element_name type=”simple” class=”simple.class.simpleName.toLowerCase”
                                                    value=”simple.toString” />
```

La clase CollectionSerializer

Para determinar si esta clase puede serializar un determinado tipo o no, se comprueba si es asignable[10] a la interfaz `Collection`.

El elemento xml que devuelve este serializador tiene la forma:

```
<element_name type="collection" class="collection.class.name"
                          content_type="collection.elementes.class">
      <... child elements ...>
</element_name>
```

La clase MapSerializer

Para determinar si esta clase puede serializar un determinado tipo o no, se comprueba si es asignable con la interfaz `Map`.

El elemento xml que devuelve este serializador tiene la forma:

```
<element_name type="map" class="map.class.name"
                                content_type="map.elements.class">
      <child_element_name ... key=”map.elements.current.key”>
      ...
      </child_element_name>
      ...
</element_name>
```

[10] Se considera a un tipo asignable a otro si es posible realizar una operación de cast del primer tipo al segundo sin que se arroje una excepción. Ejemplo: `List list = (List) new ArrayList()`, en este caso el tipo `ArrayList` es asignable a la interfaz `List`.

La clase BeanSerializer

Esta clase genera la serialización del objeto invocando todos los métodos accesores que correspondan a la forma definida por el estándar JavaBean. El nombre de un método accesor comienza por un string que depende del tipo de la propiedad a la que accede más el nombre de la propiedad con la primera letra en mayúsculas. El nombre de los accesores que obtienen la propiedad siempre comienza por el string "get", excepto para las propiedades booleanas que se puede usar "is". Este serializador ocupa la fase de configuración para obtener todos los accesores que usará durante la serialización. De esta forma se gana en eficiencia al realizar este trabajo una sola vez.

El elemento xml que devuelve este serializador tiene la forma:

```
<element_name type="bean|entity" id=”bean.id” class="bean.class.name"
                                        interface="bean.class.interfaces[0]"
                                        super_class="bean.class.superclass">
      <property_name ... >
            ...
      </property_name>
      ...
</element_name>
```

Una bean será un `entity` si contiene una propiedad llamada `id`. Así, se diferencia entre las entidades de la aplicación y los beans que son sólo contenedores de valores. El atributo `id` sólo se agregará cuando el bean sea un `entity`.

La arquitectura antigua del framework también se basa en el modelo MVC. Presenta un `FrontController` que se encarga de determinar qué clases y qué funciones llamar en función de la petición del cliente. Luego, procesa el retorno de las funciones que ejecuta para producir el resultado final que le será enviado al cliente. Las clases que el `FrontController` puede ejecutar deben implementar la interfaz `Action`. Esta interfaz expone una función que recibe como parámetros un `Map` con los datos enviados por el cliente y retorna un objeto árbol xml que debe estar formado por objetos que hereden de `XMLEntity`. Con este diseño se deja totalmente acoplado el uso de xml como respuesta al `FrontController`. Éste es el segundo punto de acoplamiento.

Para determinar qué clases `Action` debe invocar el `frontController`, se usa un esquema de páginas que pueden ejecutar muchas clases `Action`. Esto se especifica en un archivo de configuración en el que se indica el nombre de la página y todas las clases `Action` que ejecuta junto con los parámetros extras que pueda necesitar para funcionar. Si bien este esquema permite la reusabilidad de las clases `Action` y da libertad de agregar y quitar funcionalidad a una pagina rápidamente agregando y quitando `Actions`, se convierte en otro gran problema de mantención de configuración. El esquema de agregar y quitar Actions a una página se complica cuando la ejecución de un `Action` determina el comportamiento del siguiente a ejecutar. Esta dependencia no queda explícita en la configuración, por lo que si se desea quitar un `Action` o cambiarlos de orden, se requiere analizar por dentro cada uno de ellos antes de poder realizar el cambio. Esto lleva a que ciertos `Action` se vuelvan dependientes de que otros se ejecuten antes que él, generando una dependencia implícita difícil de seguir. Esta dificultad radica en que el paso de parámetros entre uno y otro es indirecto por medio de la sesión. Realizar un refactoring para unir dos `Action` es una tarea compleja ya que el IDE no ve ninguna relación directa entre ellos dada la forma en que se comunican.

El rediseño del `FrontController` se basó en que no quede acoplado al uso de xml como retorno de las clases que invoque y que requiera de la menor configuración posible para funcionar. Con esto como base se diseñó que una página esté ligada a un sólo método de una clase. Las clases contenedoras de estos métodos serán los Controllers. Así, cuando se solicite la dirección `http://server:port/AppContext/employees/index.dvt`, se sabe que el método a invocar será `index` de la clase `EmployeesController`. De esta forma se logra que toda la lógica de la ejecución de la página quede expresada en lenguaje Java con todos los beneficios que esto trae en el control del flujo y la capacidad de ser probada.

La siguiente figura muestra el diagrama de secuencia de ejecución del `FrontController` frente a una petición de un navegador Web.

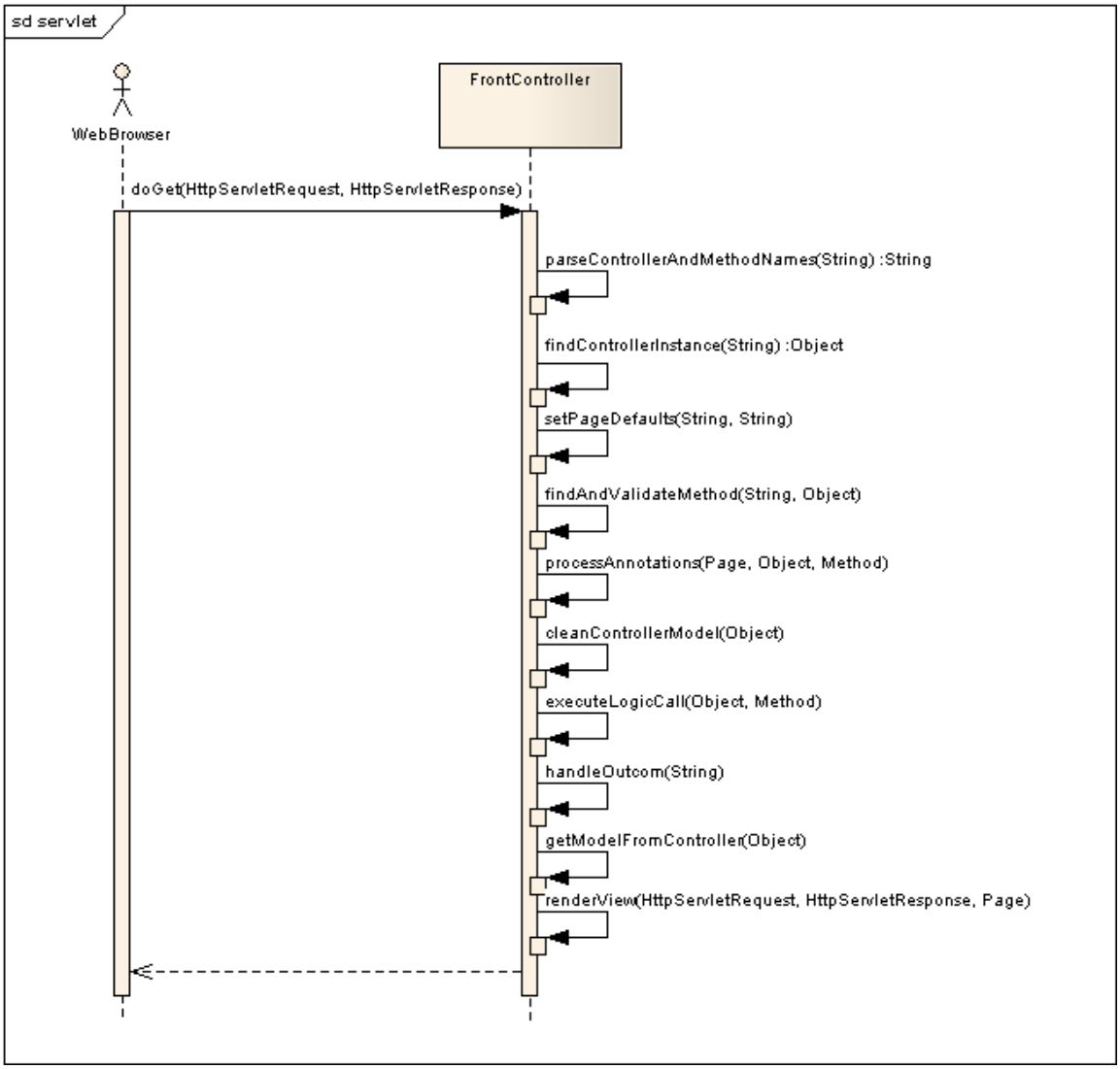


Figura 4: Diagrama de secuencia de ejecución de la clase FrontController.

La instancia del Controller se obtiene desde el contenedor usando el nombre entregado por la url más el string "`Controller`". Según el ejemplo usado, se buscará el bean de nombre `employeesController`. De esta forma se delega al contenedor la responsabilidad de crear las instancias. Esto permite que cada Controller sea configurado según el `scope` que requiera. En general, el `scope` de `singleton` será suficiente, pero en algunos casos se podría requerir uno de `session` o de `request`. La única validación que se hace sobre el bean obtenido, es que el nombre del package al que pertenece tenga el string "`controller.`". Para el ejemplo usado, la clase `EmployeesController` pertenece al package `cl.company.controller`. En el caso de que la url no especifique un Controller, entonces se

usará el nombre por defecto `defaultController` para obtener la instancia del Controller desde el contenedor. Si no se encuentra la instancia del Controller buscado, entonces se lanza una excepción indicando lo sucedido.

Para determinar la vista a utilizar se usa el nombre del método invocado más el resultado retornado por éste, siempre y cuando sea un string. Así, si el método retorna un `null` o un string vacío, entonces la vista a usar será exactamente el mismo nombre del método. Si el retorno es un string distinto de vacío, entonces este valor se añade al nombre del método con un punto entre ambos para generar el nombre de la vista. Para el ejemplo del párrafo anterior, si el método `index` retorna el string `success`, entonces la vista usada será `index.success`. Esta vista será buscada dentro del directorio con el nombre del Controller sin el string "`Controller`" al que pertenece el método. Para el ejemplo, sería dentro del directorio `employees`. Entonces, la ruta de la vista será `employees/index.success`. Lo mismo se aplica para el caso del `defaultController`, la ruta será `default/method`. Para los casos especiales donde se requiere realizar un `redirect` o un `forward`[11], se diseñó que se retorne un string de la forma "`comando:url`", donde el comando puede ser `redirect` o `forward`. Para el caso de un `redirect`, la `url` puede ser relativa o completa permitiendo que sea posible redirigir al cliente a una página externa a la aplicación. Para el caso de un `forward`, la `url` sólo puede ser relativa a la aplicación.

Para recuperar los objetos que serán pasados a la vista se diseñaron dos alternativas. La primera y más simple, es que el retorno del método invocado será el objeto a pasar a la vista. Este caso sirve cuando no se requiere hacer uso de las funcionalidades descritas en el párrafo anterior, que permite el retorno de un string desde el método. Se deberá usar la clase `StringModel` en caso que se desee retorna un string como objeto a pasar a la vista. La segunda, se da cuando el retorno es `null` o un string. Entonces se intentará ejecutar el método `getModelAll` sobre el Controller obtenido desde el contenedor. El retorno de este método será pasado a la vista. Si este método no existe en el Controller o el retorno es nulo, entonces esta ejecución no pasará objetos a la vista. Se creó la clase abstracta

[11] Un forward es una redirección que ocurre dentro del flujo de respuesta de la aplicación, sin tener que enviar información al cliente para que éste solicite la nueva url. Desde el punto de vista del cliente, es como si ninguna redirección ocurriera, sin embargo el resultado entregado por la aplicación será acorde al forward.

`AbstractController` para facilitar la implementación de los Controllers que requieran el uso del método `getModelAll` para recuperar los objetos a pasar a la vista.

Se creó la clase `Page` para mantener el estado de la ejecución de los distintos pasos que se deben realizar al procesar un request. Esta clase guarda los nombres del Controller y método actual, los objetos que serán pasados a la vista y el tipo de vista a utilizar.

Para desacoplar el uso de XML/XSLT como tecnología de presentación, se diseñó una estructura de clases que permite seleccionar el tipo de vista y tecnología de presentación a utilizar. De esta forma, se logra que se pueda usar cualquier tipo de tecnología de presentación que sea compatible con el modelo de trabajo expuesto. El patrón utilizado fue nuevamente Abstract Factory y las clases que lo implementan son:

- **ViewRendererFactory**: punto de entrada para obtener un `ViewRenderer`. Su función es crearlos y mantener apropiadamente.
- **ViewRenderer**: interfaz que presenta las funciones necesarias para realizar el trabajo de renderer.
- **XSLViewRenderer**: implementación de `ViewRenderer` que permite el uso de XML/XSLT como tecnología de presentación.
- **JSPViewRenderer**: implementación de `ViewRenderer` que permite el uso de JSP como tecnología de presentación.

La siguiente figura muestra el diagrama de clases asociado.

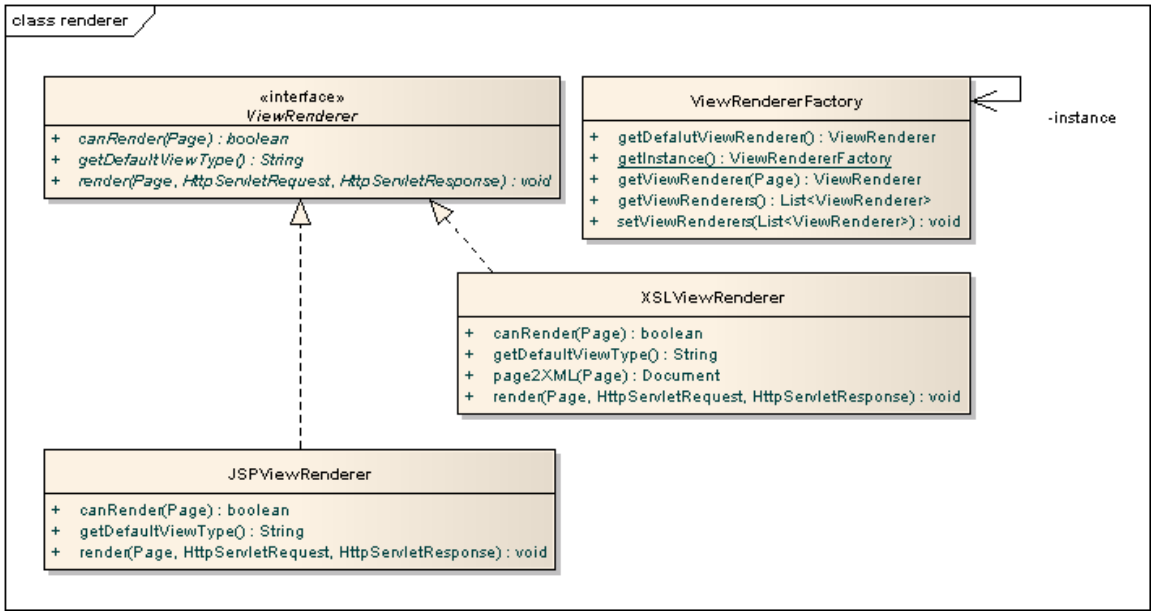


Figura 5: Diagrama de clases de la herramienta de rendering de páginas.

La clase ViewRendererFactory

Esta clase se diseñó usando el patrón Singleton. De esta forma existirá una única instancia en toda la aplicación de esta clase. Para seleccionar el `ViewRenderer` apropiado para el `page` actual se le preguntará a cada `ViewRenderer`, que se encuentre en la lista de renderers que mantiene esta clase, si es capaz de procesar el `page`. El primero que responda positivamente se retornará como el `ViewRenderer` seleccionado. Así, se logra que el orden de la lista sea la prioridad que tiene cada render y además que se delegue al renderer la responsabilidad de determinar si es capaz de renderizar el `page`. El contrato del método de un renderer que determina si es capaz de renderizar un `page`, se diseñó para que retorne `true` cuando el `page.ViewType` sea nulo. Esto permite configurar el renderer por defecto de la aplicación poniéndolo primero en la lista de renderers.

La interfaz ViewRenderer

Esta interfaz expone las tres funciones necesarias para implementar un `ViewRenderer`. Estas funciones son: `canRender`, `getDefaultViewType` y `render`. El primer método es el encargado de responder si es capaz de renderizar a un `page` determinado. El segundo retorna el `ViewType` por defecto de este renderer y finalmente el último es el encargado de renderizar un `page` determinado.

La clase XSLViewRenderer

Esta clase se encarga de manejar el proceso de transformar los objetos entregados para ser pasados a la vista en una representación xml y luego ejecutar la transformación xslt. Para transformar los objetos a xml usa las funcionalidades provistas por la clase `SerializerFactory`. La transformación xslt la realiza usando la clase `XSLTProcessor`[12].

La clase JSPViewRenderer

Esta clase se encarga de poner en contexto a los objetos pasados para la vista, para que sea posible usar una expresión EL[13] para obtener el valor requerido desde un archivo jsp.

El algoritmo para determinar el archivo de vista a utilizar resultó ser el mismo para ambos renderers. Este algoritmo consiste en formar el string: `view_directory/viewType/controllerName/methodName.methodReturn.viewType`, que corresponderá al archivo buscado. Para el ejemplo: `employees/index.success`, el archivo de vista resultante para el renderer xsl será: `/WEB-INF/views/xsl/employees/index.success.xsl`.

Una de las mayores ventajas del framework anterior en el uso de varios `Actions` por cada página, es que se le puede agregar nueva funcionalidad a la página agregando un nuevo `Action` a su configuración y realizando los cambios necesarios en la vista. En general, esta capacidad se usa para agregar `Actions` genéricos que traen entidades para su posterior uso en la vista. El ejemplo más típico es que se configura un `Action` genérico que trae una lista de entidades según el tipo y la fuente de datos especificada, las que después son usadas para rellenar un `combobox`. Este tipo de comportamiento se agregó en el nuevo desarrollo incorporando el uso del patrón de diseño Decorator. Para decorar un método se

[12] Esta clase se heredó del framework antiguo. Se encarga de realizar transformaciones xslt.
[13] EL [51] se define como Expression Language. Permite obtener el valor de una expresión usando el lenguaje permitido por esta. Para obtener el valor de la propiedad edad de un JavaBean Persona, se podría hacer de la forma: ${persona.edad}

debe usar el Annotation `Decorate` en el cual se especifica el nombre del bean decorador a usar y se le entregan los parámetros necesarios. Para especificar varios decoradores se creó el Annotation `Decoraters` que tiene un arreglo de `Decorate`. Con este diseño es posible mantener la misma funcionalidad que la versión anterior. Los decoradores deben implementar la interfaz `Decorator` que expone el método `Object decorate(String[] parameters)`. El arreglo de parámetros que se le entrega es el mismo que se configura en el Annotation `Decorate` y el retorno es agregado a los objetos a pasar a la vista.

La siguiente figura muestra el diagrama de clases asociado.

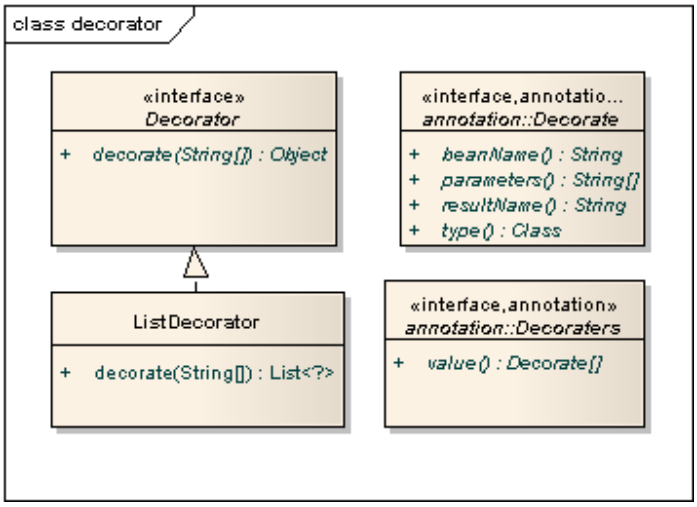


Figura 6: Diagrama de clases de la funcionalidad de Decorator.

Todo este diseño permitió que la configuración necesaria para ejecutar una página sea muy poca. Se deben registrar en el contenedor los Controllers y los Decorators a usar en toda la aplicación y con las Annotations Decorate o Decoraters especificar los Decorators que se necesite ejecutar en cada método asociado a una página. Gracias a la función de autoconfiguración que se diseñó para el contenedor, detallada en la sección 2.4.1, es posible realizar la configuración de los objetos en el contenedor con sólo unas pocas reglas de inclusión/exclusión de clases. Sólo será necesario configurar a mano en el contenedor los Controllers y Decorators que requieran de un `scope` distinto al de `singleton` o que necesiten configuración específica distinta a la por defecto.

2.4.5.2 Implementación

Las únicas clases que serán abordadas en esta sección son algunas de la funcionalidad de serialización a xml, ya que son las únicas que tienen una mayor complejidad.

Las clases del serializador a XML

La mayor complejidad que tuvo la implementación de estas clases fue lograr que el proceso de serialización fuera rápido y eficiente. La primera implementación que se hizo, fue relativamente simple y rápida de implementar. Consistía de una sola clase con varios métodos que se encargaban de serializar los distintos tipos aceptados. Esta implementación extraía poca información de los objetos y era muy desordenada y difícil de extender. En velocidad, era un poco más lento que el serializador del framework antiguo. En cantidad de información extraída eran similares. Luego, se pasó al esquema actual, pero sin optimización de ningún tipo. Este nuevo esquema permite que sea extensible y configurable la forma de serializar los objetos a xml. Sin embargo, en sus primeras versiones fue muchísimo más lento que el serializador anterior, aunque con la ventaja de extraer mucha información nueva, como por ejemplo, mostrar el tipo de la propiedad de un bean cuando su valor es nulo. La optimización consistió en hacer los serializadores específicos a cada tipo a serializar, para que de esta forma se pueda extraer toda la información posible del tipo al momento de configurar el serializador. La información que se guarda para el caso del BeanSerializer es una lista de métodos que representan los accesores a sus propiedades, el tipo de cada propiedad incluyendo sus valores genéricos[14], las interfaces que implementa y su superclase. De esta forma al serializar el objeto toda esta información ya es conocida. Como el mismo serializador se reutiliza para todas las instancias del tipo, entonces el costo de obtener toda esa información es constante y sólo se paga la primera vez. Con estos cambios se logró un serializador que es sólo un poco más lento que el del framework anterior, pero que genera más del doble de información y que es configurable y extensible.

Según las pruebas realizadas en el computador del autor para serializar 1000 JavaBean con cuatro atributos cada uno inicializados al azar, se obtuvieron los siguientes

[14] Para la propiedad `List<Employee> employees`, su tipo es `List` y su tipo genérico es `Employee`.

resultados para el serializador antiguo y nuevo. Los valores son el promedio de 10 ejecuciones.

Serializador	Primera vez	Repetitiva
Antiguo	72 ms.	35 ms.
Nuevo	80 ms.	57 ms.

Tabla 1: Tiempos de ejecución de los serializadores de Objetos a xml.

Se puede notar el mayor tiempo requerido en la primera serialización donde ambos realizan trabajo extra. El antiguo lee la información desde el archivo de configuración y el nuevo ejecuta la función `introspect`. En la ejecución repetitiva se ve que el nuevo versus el antiguo guardan una relación de 1.5 veces más lento aproximadamente. Estos valores se consideran satisfactorios considerando que el nuevo evita la creación de la configuración, entrega mucha más información y que es configurable.

El siguiente código muestra la función `instrospect` de la clase `BeanSerailizer`, la que se encarga de recuperar toda la información posible del tipo.

```
public void instrospect() {
	super.instrospect();
	Class[] interfaces = clazz.getInterfaces();
	if(interfaces.length > 0){
		rootPrototype.setAttribute("interface", interfaces[0].getName());
	}
	Class superClass = clazz.getSuperclass();
	if(superClass != null && superClass != Void.TYPE && superClass != Object.class){
		rootPrototype.setAttribute("super_class", superClass.getName());
	}
	List<Method> getterMethods = ReflectionController.findMatchingMethodsByName(clazz,
                                                                "(get.*)|(is.*)", 0);

	Method getter;
	String getterName;
	for (int i=0; i < getterMethods.size(); i++ ) {
		getter = getterMethods.get(i);
		getterName = getter.getName();
		if(getterName.equals("getId")){
			setElementType("entity");
			rootPrototype.setAttribute("type", getElementType());
		}else if(getterName.equals("getClass")){
			getterMethods.remove(i);
			i--;
			continue;
		}
		getters.add(new GetterInfo(getter));
		Class returnClass = getter.getReturnType();
		Type returnType = getter.getGenericReturnType();
		ObjectSerializer objectSerializer;
```

```
                if(!returnClass.getName().equals(clazz.getName()))
                        objectSerializer =
serializerFactory.getSerializerForType(returnType);
                else
                        objectSerializer = this;
                defaultSerializers.add(objectSerializer);
        }
}
```

Para resolver el problema de los ciclos, se usó un mapa de nodos visitados asociados con un entero. Cada vez que se va a visitar un nodo se revisa si es posible hacerlo según la restricción de recursividad máxima configurada. De ser posible, se suma uno en el mapa y se visita al nodo. De no ser posible, se visita al nodo en modo especial para que se agregue sólo la definición del nodo pero no su contenido y se le agrega el atributo `max_deep_reach=true`. Cada vez que se deja un nodo, se le resta uno en el mapa. De esta forma, es posible resolver el problema de los ciclos limpiamente.

El siguiente código muestra el método `serialize` que corresponde al template de la clase `ObjectSerializer`. En él se implementa la resolución de ciclos.

```
public Element serialize(Object object, String name) {
        preCheck(object);
        boolean maxDeepReached = false;
        if (StringUtils.isEmpty(name)) {
                if (object != null)
                        name = StringUtils.underscorize(object.getClass().getSimpleName());
                else
                        name = defaultName;
        }
        Element root = (Element) rootPrototype.clone();
        root.setName(name);
        if(object == null || !Hibernate.isInitialized(object)){
                return root;
        }
        maxDeepReached = processRecursion(object, root);
        try{
                if(!maxDeepReached)
                        processObject(object, root);
        }finally{
                if(!maxDeepReached)
                        leaveNode(object);
        }
        return root;
}
```

Si se pide que se serialize una instancia de un `Collection` o `Map` directamente, sin pasar por un JavaBean que lo contenga, entonces la única forma de obtener el tipo de los objetos que lleva dentro es inspeccionándolos. Sólo es posible determinar el tipo de los objetos que lleva dentro usando la información provista por Generics cuando se trata de una

propiedad de un JavaBean. El siguiente código muestra el método `findContentType` de la clase `ObjectSerializer` que se encarga de obtener esta información para este caso.

```
protected static String findContentType(Class clazz) {
	Logger.debugMessage("input Class: " + clazz);
	Class contentTypeClass = null;
	// 1. assign from superClass if different than null and Object.class
	contentTypeClass = clazz.getSuperclass();
	if (Object.class.equals(contentTypeClass)) {
		contentTypeClass = null;
	}
	// 2. assign from first implemented interface diferent from the pattern java*
	if (contentTypeClass == null) {
		Class[] interfaces = clazz.getInterfaces();
		if (interfaces.length > 0 && !interfaces[0].getName().startsWith("java")) {
			contentTypeClass = interfaces[0];
		}
	}
	// 3. assign the input type;
	if (contentTypeClass == null) {
		contentTypeClass = clazz;
	}
	return contentTypeClass.getName();
}
```

2.4.5.3 Ejemplos

Serialización a XML

El siguiente código muestra parte de la prueba de unidad para el serializador de JavaBean.

```
public SerializerTest() {
	serializerFactory = SerializerFactory.getInstance();
	Format format = Format.getPrettyFormat();
	 format.setEncoding("ISO-8859-1");
	 xmlOut = new XMLOutputter(format);

	empleado = new Empleado();
	empleado.setRut("123");
	empleado.setEdad(25);
	empleado.setId(159L);
	empleado.setNombre("EL super Empleado");

	jefe = new Empleado();
	jefe.setRut("456");
	jefe.setEdad(50);
	jefe.setId(753L);
	jefe.setNombre("Jefecito");

	subordinados = new ArrayList<Persona>();
	Empleado subordinado = new Empleado();
	subordinado.setEdad(23);
```

```
        subordinado.setNombre("n1");
        subordinados.add(subordinado);
        subordinado = new Empleado();
        subordinado.setEdad(222);
        subordinado.setNombre("n2");
        subordinados.add(subordinado);

        companeros = new HashMap<String, Serializable>();
        companeros.put("integer", new Integer(127));
        Empleado companero = new Empleado();
        companero.setEdad(23);
        companero.setNombre("n3");
        companeros.put("n3", companero);
        companero = new Empleado();
        companero.setEdad(222);
        companero.setNombre("n4");
        companeros.put("n4",companero);

}

public void testBeanSerializer() throws Exception {
        empleado.setJefe(jefe);
        empleado.setCompaneros(companeros);
        companeros.put("loop", empleado);
        ObjectSerializer objectSerializer =
                serializerFactory.getSerializerForType(empleado.getClass());
        assertNotNull(objectSerializer);
        assertEquals(BeanSerializer.class, objectSerializer.getClass());
        element = objectSerializer.serialize(empleado, "empleado");
        assertNotNull(element);
        assertTrue(element.getChildren().size() > 0);
        xmlOut.output(element, System.out);
}
```

El resultado de esta ejecución es:

```
<empleado type="entity" class="cl.company.entities.Empleado"
   interface="cl.company.entities.Persona" super_class="cl.company.entities.EmpleadoBase"
   id="159">
  <info type="simple" class="string" value="cl.company.entities.Empleado@1edd9b3" />
  <subordinados type="collection" class="java.util.List" />
  <jefe type="entity" class="cl.company.entities.Empleado"
    interface="cl.company.entities.Persona"
    super_class="cl.company.entities.EmpleadoBase" id="753">
    <info type="simple" class="string" value="cl.company.entities.Empleado@1875da7" />
    <subordinados type="collection" class="java.util.List" />
    <jefe type="entity" class="cl.company.entities.Empleado"
      interface="cl.company.entities.Persona"
      super_class="cl.company.entities.EmpleadoBase" />
    <edad type="simple" class="integer" value="50" />
    <nombre type="simple" class="string" value="Jefecito" />
    <companeros type="map" class="java.util.Map" />
    <rut type="simple" class="string" value="456" />
  </jefe>
  <edad type="simple" class="integer" value="25" />
  <nombre type="simple" class="string" value="EL super Empleado" />
  <companeros type="map" class="java.util.HashMap" content_type="java.io.Serializable">
    <empleado type="entity" class="cl.company.entities.Empleado"
      interface="cl.company.entities.Persona"
      super_class="cl.company.entities.EmpleadoBase" id="null" key="n3">
```

```
      <info type="simple" class="string" value="cl.company.entities.Empleado@7ab40c" />
      <subordinados type="collection" class="java.util.List" />
      <jefe type="entity" class="cl.company.entities.Empleado"
        interface="cl.company.entities.Persona"
        super_class="cl.company.entities.EmpleadoBase" />
      <edad type="simple" class="integer" value="23" />
      <nombre type="simple" class="string" value="n3" />
      <companeros type="map" class="java.util.Map" />
      <rut type="simple" class="string" />
    </empleado>
    <integer type="simple" class="integer" value="127" key="integer" />
    <empleado type="entity" class="cl.company.entities.Empleado"
      interface="cl.company.entities.Persona"
      super_class="cl.company.entities.EmpleadoBase" id="null" key="n4">
      <info type="simple" class="string" value="cl.company.entities.Empleado@1c2a1ed" />
      <subordinados type="collection" class="java.util.List" />
      <jefe type="entity" class="cl.company.entities.Empleado"
        interface="cl.company.entities.Persona"
        super_class="cl.company.entities.EmpleadoBase" />
      <edad type="simple" class="integer" value="222" />
      <nombre type="simple" class="string" value="n4" />
      <companeros type="map" class="java.util.Map" />
      <rut type="simple" class="string" />
    </empleado>
    <empleado type="entity" class="cl.company.entities.Empleado"
      interface="cl.company.entities.Persona"
      super_class="cl.company.entities.EmpleadoBase" max_deep_reach="true" key="loop" />
  </companeros>
  <rut type="simple" class="string" value="123" />
</empleado>
```

La salida producida para este mismo JavaBean por el primer serializador que se desarrolló y que es muy similar a la producida por el serializador de la versión antigua es:

```
<empleado class="cl.company.entities.Empleado" id="159">
  <info>cl.company.entities.Empleado@1edd9b3</info>
  <subordinados />
  <jefe class="cl.company.entities.Empleado">
    <info>cl.company.entities.Empleado@1875da7</info>
    <subordinados />
    <jefe />
    <edad>50</edad>
    <nombre>Jefecito</nombre>
    <companeros />
    <rut>456</rut>
  </jefe>
  <edad>25</edad>
  <nombre>EL super Empleado</nombre>
  <companeros class="java.util.HashMap">
    <entry>
      <key class="java.lang.String">n3</key>
      <value class="cl.company.entities.Empleado" id="null">
        <info>cl.company.entities.Empleado@7ab40c</info>
        <subordinados />
        <jefe />
        <edad>23</edad>
        <nombre>n3</nombre>
        <companeros />
        <rut />
      </value>
```

```
    </entry>
    <entry>
      <key class="java.lang.String">integer</key>
      <value class="java.lang.Integer">127</value>
    </entry>
    <entry>
      <key class="java.lang.String">n4</key>
      <value class="cl.company.entities.Empleado" id="null">
        <info>cl.company.entities.Empleado@1c2a1ed</info>
        <subordinados />
        <jefe />
        <edad>222</edad>
        <nombre>n4</nombre>
        <companeros />
        <rut />
      </value>
    </entry>
    <entry>
      <key class="java.lang.String">loop</key>
                <... 72 lineas omitidas ...>
                </entry>
                <entry>
                  <key class="java.lang.String">loop</key>
                  <value class="cl.company.entities.Empleado">max recursive deep reached</value>
                </entry>
              </companeros>
              <rut>123</rut>
            </value>
          </entry>
        </companeros>
        <rut>123</rut>
      </value>
    </entry>
  </companeros>
  <rut>123</rut>
</empleado>
```

Haciendo una comparación directa entre ambos resultados, resulta evidente que el del nuevo es más compacto y añade mucha más información útil.

Funcionamiento del FrontController

El siguiente ejemplo muestra todas las funcionalidades para crear páginas usando la nueva versión del FrontController.

El Controller es `ExampleController` y extiende de `AbstractController` para usar la facilidad del manejo de los objetos que serán pasados a la vista usando los métodos `setModel` y `getModelAll`.

```
public class ExampleController extends AbstractController {

       public String complex(){
              Empleado jefe = new Empleado();
              jefe.setNombre("Jorge");
              jefe.setEdad(45);
```

```
		Empleado arquitecto = new Empleado();
		arquitecto.setNombre("nombre");
		arquitecto.setEdad(25);
		arquitecto.setJefe(jefe);

		List<Empleado> empleados = new ArrayList<Empleado>();
		empleados.add(jefe);
		empleados.add(arquitecto);

		setModel("arquitecto", arquitecto);
		setModel("empleados", empleados);
		return null;
	}

	@Decoraters({
		@Decorate(beanName="listDecorator",
parameters={"cl.company.entities.Empleado", "empleadosDAO"}),
		@Decorate(beanName="listDecorator", resultName="empleados",
parameters="cl.company.entities.Empleado")
	})
	public String decoratorts(){
		Empleado jefe = new Empleado();
		jefe.setNombre("Jorge");
		jefe.setEdad(45);

		Empleado arquitecto = new Empleado();
		arquitecto.setNombre("nombre");
		arquitecto.setEdad(25);
		arquitecto.setJefe(jefe);

		List<Empleado> empleados = new ArrayList<Empleado>();
		empleados.add(jefe);
		empleados.add(arquitecto);
		setModel("empleados", empleados);
		return "success";
	}

	public String redirect(){
		return "redirect:/empleados/index.dvt";
	}

	public String forward(){
		return "forward:/empleados/index2.dvt";
	}

	public Integer integer(){
		return new Integer(1);
	}

	public StringModel string(){
		return new StringModel("Un String de retorno");
	}
}
```

La url base para el ejemplo será: `http://localhost:8081/FrameworkDevelopment`, a ésta debemos añadirle el nombre del Controller y método que queramos ejecutar. Para

ejecutar la página `decorators` del Controller `ExampleController`, debemos usar la url: `http://localhost:8081/FrameworkDevelopment/example/decorators.dvt`.

2.4.6 Requerimiento 8: Herramienta para desarrollar usando JUnit

2.4.6.1 Diseño

El framework JUnit provee la clase abstracta `TestCase` para generar las pruebas de unidad. Entonces el diseño consistió en generar una clase que extienda a `TestCase` y que se preocupe de inicializar la configuración obteniendo la instancia de la clase `Context`. La clase `TestCase` provee de dos funciones para controlar el flujo de ejecución de cada prueba. Estas funciones son `setUp` y `tearDown`, las que son ejecutadas antes y después de cada prueba respectivamente. En un principio, se pensó en usar estas funciones para obtener una instancia de la clase `Context` en la fundón `setUp` y luego, destruirla en la función `tearDown`, para que en cada prueba se cuente con una configuración recién levantada. Sin embargo, esta idea se desechó porque levantar la configuración una y otra vez es demasiado lento teniendo en cuenta que una aplicación puede llegar a tener cientos de pruebas. Finalmente se decidió que la configuración sólo se levantaría una vez para todas las pruebas.

La clase generada se llama `FrameworkTestCase` y todas las pruebas de unidad de la aplicación a de desarrollar con el framework deben ser hechas extendiendo de esta clase.

2.4.6.2 Implementación

Esta clase extiende a `junit.framework.TestCase` y para obtener la configuración sólo una vez se obtiene la instancia de `Context` en el constructor. Además, se agregó la propiedad `context` de tipo `Context` a esta clase para que sea directo obtener la instancia de este singleton dentro de las pruebas de unidad. En el siguiente código se ve lo relevante de esta clase.

```
public class FrameworkTestCase extends TestCase {
	private Context context;

	public FrameworkTestCase() {
		super();
```

```
		init();
	}

	public FrameworkTestCase(String name) {
		super(name);
		init();
	}

	private void init(){
		// Starts the framework
		// Ask first if the framework was already initialized to only send
		// the info message once.
		if(!Context.isReady()){
			setContext(Context.getInstance());
			Logger.infoMessage("New Test Suit Initialized");
		}else {
			setContext(Context.getInstance());
		}
	}
	...
}
```

2.4.6.3 Ejemplo

La siguiente prueba unidad corresponde a la de la clase `AutoSpringContextConfigurer`.

```
public class AutoSpringContextConfigurerTest extends FrameworkTestCase {

	public void testConfigureBeans() throws Exception {
		BeanDefinitionRegistry beanDefinitionRegistry =
				(BeanDefinitionRegistry)
				((ClassPathXmlApplicationContext)
				getContext().getSpringContext()).getBeanFactory();

		AutoSpringContextConfigurer.findAndConfigureBeans(new String[]{
			"cl.dybox.framework.test.**",
			"!cl.**.dao.*",
			"!cl.dybox.framework.test.entities.*Base"
			}, beanDefinitionRegistry, false);

		assertTrue(
			beanDefinitionRegistry.containsBeanDefinition("empleadosDAO")
		);
		BeanDefinition bd =
			beanDefinitionRegistry.getBeanDefinition("empleadosDAO");

		assertTrue(bd.getBeanClassName().endsWith("EmpleadosDAOImp"));

		assertTrue(
			beanDefinitionRegistry.containsBeanDefinition("empleadosApp")
		);

		assertTrue(
```

```
                !beanDefinitionRegistry.containsBeanDefinition("empleadoBase")
            );

            CompaniaApp companiaApp = getContext().getBean("companiaApp");

            assertNotNull(companiaApp.getEmpleadosApp());

            //Mailer is already configured in the mailContext.xml as mailer
            assertNotNull(companiaApp.getMailer());
        }
}
```

2.5 Resumen de la convención generada

A lo largo del proceso de desarrollo de cada uno de los requisitos se fue generando la convención a usar por sobre la configuración. En esta sección se resumirá la convención generada durante todo el desarrollo. Aquí se indicarán los valores por defecto para cada una de las configuraciones posibles. Estos valores por defecto son la convención.

Archivo de configuración `application.properties`

- `app.production`: false
- `app.context.load`: local
- `app.log.path`: ruta de ejecución de la aplicación
- `app.context.auto_configurer_rules`: null

Carga de configuración del contenedor SpringFramework

- Los nombres de los archivos xml deben ser de la forma `*Context.xml` y estar dentro de la carpeta `context` o de alguna de sus subcarpetas.
- Los nombres de las propiedades de los beans a configurar en el contenedor deben tener un nombre significativo que lo relacione directamente con otra clase por nombre. Por ejemplo: Si una clase de la lógica necesita enviar mail, entonces debería tener una propiedad llamada `mailer` que es el nombre de la herramienta para el envío de mail del framework. La clase de esta herramienta es `Mailer`. De igual forma se requiere de algún DAO, debería tener la propiedad con el nombre de la interfaz del DAO.
- Se espera que para el caso de las clases que implementen una interfaz de lógica de la aplicación, éstas tengan el nombre de la interfaz más "`Imp`". En el caso de existir más de un

tipo de implementación, se deben crear subpackages que agrupen dichas implementaciones. Ejemplo: para la interfaz `EmpleadosDAO` su implementación se debe llamar `EmpleadosDAOImp`. Si existieran variaciones, como por ejemplo una implementación maqueta, entonces esta implementación debería ir en el subpackage `mock` relativo a la interfaz.

- Clases que implementen una interfaz de la lógica de la aplicación deberían ser configuradas para usar el nombre de la interfaz. Ejemplo: para la interfaz `EmpleadosDAO` y la implementación `EmpleadosDAOImp`, el bean debiera llamarse "`empleadosDAO`" y no "`empleadosDAOImp`".

Herramienta de log

La clase `Logger` inicializa su instancia de `Log` a `LogStdImp` por defecto. Para cambiar esta instancia automáticamente, se debe configurar un bean con el nombre "`log`" en el contenedor. La clase `Context` se encargará de configurar la clase `Logger` usando este bean en la fase de inicialización del framework.

Herramienta para Servicios Web

Las clases asociadas a esta herramienta deben ir dentro del package `ws`. El nombre de la interfaz que representa a cada Servicio Web debe terminar en "`WS`" y el de la implementación en "`WSImp`". Estas clases deben respetar las convenciones descritas para las clases que se configurarán dentro del contenedor.

Capa de Presentación

- La extensión de las urls a usar deben ser "`dvt`". Ejemplo:

`http://localhost/AppContext/controller/method.dvt`

- El tipo de vista por defecto es xslt.
- Los nombres de las clases Controller deben terminar en "`Controller`".
- Los Controllers deben respetar las convenciones descritas para las clases que se configurarán dentro del contenedor.
- En caso de usar una url sin Controller, se buscará en el contenedor el Controller llamado "`defaultController`".

- La url debe llevar el nombre del Controller sin la parte final del nombre de la clase que seria “Controller”. Ejemplo: la url `/exampleController/method.dvt` es incorrecta, la versión correcta es `/example/method.dvt`.

3 Resultado obtenido

Se obtuvo un nuevo framework que se usó para construir dos aplicaciones reales de la cartera de Dybox. La primera aplicación se encontraba en fase de pruebas finales por parte del cliente para luego ser pasada a producción. La segunda se encontraba en pleno desarrollo, contando ya con tres entregas intermedias al cliente al momento de escribir esta sección. Lamentablemente, ninguna de estas aplicaciones usó el rediseño de la capa de presentación a fondo, aunque ambas se vieron beneficiadas del desacoplamiento de la misma del resto de la arquitectura. Esto permitió que en la segunda aplicación fuese usado el MVC JSF ICEFaces [52] como capa de presentación sin ser necesaria ninguna modificación extra a la arquitectura.

Se redujo significativamente la cantidad de configuración necesaria para que la aplicación funcione. Lo anterior, gracias al uso de CoC, ID y a la funcionalidad de auto configuración implementada. En la tabla 2 se aprecia la diferencia de configuración necesaria para una pequeña aplicación que permite administrar una entidad. La medición se hizo en puntos de configuración, que son sólo las líneas de contenido de la configuración, despreciando las que sirven para dar o completar el formato en el que se escriben.

Versión/Capa	Presentación	Lógica	Acceso a Datos
Antiguo	3 Entidad x 5 atributos + 4 vistas x 3 puntos de configuración por cada una = 27	0	Igual para ambos
Nuevo	0	1 línea para el autoconfigurador	Igual para ambos
Diferencia	-27 puntos de configuración	+1 punto de configuración	Sin cambio

Tabla 2: Comparación de cantidad de configuración necesaria por capa.

Este resultado muestra la alta cantidad de configuración necesaria en la capa de presentación para el framework antiguo, versus la ausencia total de ésta para el nuevo. En

la capa lógica, se agrega un punto de configuración impuesto por la necesidad de que las instancias de los objetos de esta capa sean administradas por el contenedor, sin embargo, éste no crece proporcionalmente a la cantidad de funcionalidad de la aplicación. En la capa de acceso a datos no hay variaciones significativas como para ser medidas, dado que las mejoras en este nivel apuntaron a actualizar sólo la versión del ORM utilizado. Si a esta aplicación se le quisieran agregar nuevas funcionalidades, entonces en el framework antiguo, sería necesario seguir agregando configuración en la capa de presentación, mientras que la del nuevo se mantendría constante. Lo único que sería necesario configurar en el nuevo framework para esta capa, es el uso de los decoradores con los Annotations `Decorate` y `Decorates` cuando se requiriesen.

La herramienta de log generada, permite reducir la cantidad de líneas de código asociada al envío de mensajes al log dentro de la aplicación. Esta herramienta es capaz de identificar desde que clase y método se generó el envío, permitiendo que su API ya no requiera el paso de esta información en cada invocación. El nuevo diseño permite el intercambio de la implementación de log a usar. Se desarrollaron dos implementaciones, una que envía los mensajes a la salida estándar y otra que es un adaptador para usar el framework log4j.

El uso del contenedor de aplicaciones liviano de SpringFramework permitió evitar la creación innecesaria de instancias de objetos. El ciclo de vida de los objetos se le delega al contenedor, permitiendo configurar cada instancia entre varios ciclos de vida de diversa duración.

La integración del framework CXF permitió disponibilizar las funciones de la capa lógica como Servicios Web. Se optó por usar el estándar JAX-WS soportado por CXF para generar los servicios. La integración lograda con este framework y la adopción del estándar JAX-WS, permitió que el acoplamiento con CXF sea mínimo y que sea muy simple crear un nuevo Servicio Web.

El rediseño de la capa de presentación logrado, permite que sea posible usar cualquier tipo de tecnología de presentación. Para la presentación usando XML/XSLT, se creó una herramienta de serialización de objetos a xml, que es capaz de funcionar sin ninguna configuración especial. De todas formas, permite ser configurada y extendida agregando nuevos serializadores para tipos específicos, según se requiera. Este rediseño permitió que no sea necesario agregar más configuración al crear una nueva página Web. Gracias a que se desacopló esta capa del resto de la arquitectura del framework, es posible reemplazarla completamente.

Se actualizaron todas las componentes en las que depende el framework a las versiones más nuevas posible. La actualización permitió que el framework se beneficie de todos los arreglos y mejoras incluidas en ellas.

Se creó una herramienta que permite la ejecución de pruebas de unidad sin tener que cambiar la configuración. Esta herramienta es simple y permite que se empleen técnicas de desarrollo como el TDD.

Todos, excepto el requerimiento 6, fueron terminados por completo. El requerimiento 6, que consiste en poder intercambiar la capa de presentación, requirió que se rediseñara completamente su funcionamiento y estructura. Se rediseñó e implementó una primera versión funcional, pero no alcanzó a ser probada en una aplicación real. Al momento de escribir esta sección, se estaba estudiando en cuál de las nuevas aplicaciones a desarrollar por Dybox se probaría.

El peso del nuevo framework es de poco más de 110kb empaquetado en un archivo jar. El tamaño final es mucho mayor debido a que hay que añadirle los jars de las componentes que requiere para funcionar. Dependiendo de las componentes a usar, varía entre 12mb y 17mb. La cantidad de clases creadas por el autor fueron poco más de 60. En este número se consideran otras clases necesarias para la implementación de las funcionalidades, pero que no fueron descritas en este documento, u otras clases que implementan otros requisitos que no fueron parte de este trabajo. De estas últimas, las

funcionalidades más importantes son la capacidad de declarar un trabajo que se debe repetir en el tiempo mediante configuración, una herramienta para el envío de mail asíncrono y el manejo de cache por hilo y sincrónicos.

4 Conclusiones

Las dos aplicaciones que fueron implementadas usando el nuevo framework, fueron desarrolladas por dos personas que ya tenían experiencia trabajando con el framework antiguo. El uso de CoC junto a ID y la autoconfiguración desarrollada, permiten que la configuración necesaria sea mínima. Sin embargo, a los dos desarrolladores que usaron el nuevo framework, les costo acostumbrase a tener que configurar poco y valerse de las convenciones. Pareciera que se sienten más seguros escribiendo todo, que dejando que la convención funcione. Este comportamiento fue más evidente en el desarrollador que más tiempo llevaba usando el framework antiguo. Al pasar de los días, ambos terminaron acostumbrándose a la autoconfiguración, CoC e ID, pasando al otro extremo de querer que todo funcione automáticamente.

Otra práctica que le costo mucho dejar de lado a los desarrolladores fue el empezar toda función llamando al constructor de los objetos, cuando esto no era necesario, porque el contenedor inyectaría la instancia requerida o se debía obtener directamente pidiéndola al contenedor. A pesar de que se les explicó cómo funciona el contenedor de ID y que la creación de instancias debían ser delegadas a éste en la inmensa mayoría de los casos. Finalmente se observó que a pesar de que partían usando un constructor, sabían que debían cambiarlo en algún momento, aunque esta práctica no es la ideal, se consideró como un buen comienzo.

Para mejorar el proceso de adopción de esta nueva tecnología, se podría haber contado con un documento corto y simple que tuviera las claves del uso del framework. Con esta ayuda memoria, los desarrolladores podrían aclarar sus dudas sin tener que recurrir al autor, con lo que se podría haber obtenido mejores resultados. Al momento de escribir esta sección, un tercer desarrollador de Dybox se encontraba realizando la tarea de generar una Wiki del uso del framework. Este tercer desarrollador no tenía experiencia en el uso del framework antiguo, por lo que adoptar las convenciones y prácticas de uso del nuevo framework le fue un poco más fácil que a los 2 anteriores. La documentación que se le encargó hacer, fue una guía rápida de cómo realizar cada una de las tareas básicas y una

segunda que explicase cada punto en mayor detalle. La primera guía se encontraba en desarrollo y la segunda sin comenzar. El trabajo se realizó enseñándole al desarrollador como se debían hacer las cosas, luego él generaba un documento y el autor lo revisaba y corregía. Si los errores eran graves, se les explicaba de nuevo el punto y se le pedía que generara una nueva versión del documento.

El hecho de contar con que el nuevo framework fuera usado en dos aplicaciones reales mientras se construía, permitió validar y mejorar cada una de las funcionalidades requeridas por éstas, excepto las relativas a la capa de presentación, ya que ninguna de estas aplicaciones la utilizó directamente.

La simplificación lograda en el uso de la herramienta de log, permite que el código generado sea más simple y legible. La capacidad de detectar automáticamente de donde fue generado el mensaje, permite que el desarrollador sólo escriba el contenido del mismo delegando en la herramienta la responsabilidad de completarlo con esta información. Este rediseño logra que el proceso de depuración de la aplicación sea más simple y certero. La capacidad desarrollada de intercambiar por configuración la implementación de log a usar, permite que la aplicación este preparada para soportar nuevas tecnologías de log sin tener que ser recompilada. Con el uso de log4j como framework de log, es posible beneficiarse de todas las capacidades que éste brinda. La que fue de mayor importancia para el autor, fue la capacidad de poder definir distintos archivos de salida para cada package o clase de la aplicación. Esto permite separar el log en tantos archivos como clases existan, permitiendo generar una configuración tan fina como se desee.

La reducción de la cantidad de configuración y el soporte del nuevo framework para realizar pruebas de unidad sin tener que alterar la configuración, permitiría que el desarrollador le dedique más tiempo efectivo al trabajo en el caso uso. Esto habría que demostrarlo con algún tipo de instrumento a desarrollar en el futuro.

Con respecto a la selección de las componentes a usar, se considera un gran acierto el uso de SpringFramework como contenedor, ya que es muy extensible y relativamente

fácil de usar como desarrollador. Por otro lado, facilitó tremendamente la tarea de integración de la componente CXF para la disponibilización de Servicios Web. SpringFramework junto a CXF, permitió que todo lo referente a Servicios Web quede casi totalmente fuera del código Java, permitiendo abstraer a la aplicación de que está sirviendo un Servicio Web o de que está usando uno como cliente.

El desacoplamiento del framework del uso de XML/XSLT como tecnología de presentación, permite abrir las puertas a nuevos desarrollos que requieran del uso de tecnologías basadas en JSP. Gracias a que es posible especificar el tipo de vista a usar en cada petición del cliente, se puede tener un ambiente mixto en donde algunas páginas sean servidas usando XML/XSLT y otras JSP o algún derivado.

Otra área que sufrió una fuerte mejora fue la serialización de objetos a xml. La ausencia de configuración necesaria para realizar la serialización, el manejo de ciclos y la cantidad de información que se le añadió al xml generado, permiten que el futuro rediseño e implementación del framework XSLT tenga una buena base en donde afirmar sus cimientos. Estas mejoras también repercuten en que el desarrollador tenga menos trabajo de configuración y por ende otro punto menos donde cometer errores.

La opinión del arquitecto general, es que este nuevo framework cuenta con las bases sólidas para sustentar el crecimiento que tendrá a medida que se le añadan más funcionalidades. Sin embargo, su visto bueno oficial quedará retenido hasta que la capa de presentación se considere lista. Con esta última fase terminada, el nuevo framework estará en condiciones de ser liberado oficialmente, con lo que reemplazaría al antiguo. Al pasar a esta nueva etapa, el diseño e implementación de todo lo nuevo se abordará en conjunto con el resto de los arquitectos de Dybox.

5 Trabajo Futuro

Se debe generar un documento corto con las claves del uso del framework, en el que se expongan pequeños fragmentos de código que muestren el uso de cada una de las secciones importantes. Éste debería tener un recuadro con el resumen de las convenciones y otro con las APIs principales. Se espera que este documento facilite la adopción de esta nueva tecnología y sirva como guía en el proceso de desarrollo diario. Idealmente que no sea más de una pagina para que pueda ser impresa y mantenida por el desarrollador en su escritorio para consultas rápidas.

Se debe seguir trabajando en el desarrollo de la capa de presentación. Se debe probar en profundidad la nueva arquitectura, con casos de uso reales que permitan averiguar si el rediseño es un real aporte, o fue sólo cambiar un problema por otro. No se sabrá a ciencia cierta si el uso de un Controller con una función por página es una buena solución o es mejor cambiar a la alternativa de una clase por página. Sin embargo, es la opinión del autor que la nueva arquitectura permite más grados de libertad al incluir el uso del `redirect` y `forward` dinámico, ya que en la versión anterior sólo existe `redirect` estático.

Otra área en donde se debe enfatizar es en la creación o adopción de un instrumento que permita medir las diferencias objetivas en calidad y tiempo de desarrollo del software creado usando distintas versiones del framework. Esto es de vital utilidad para determinar si los cambios introducidos aportan alguna mejora o simplemente se está cambiando el problema de forma y no dándole una solución de fondo.

Se deben migrar las demás funcionalidades que ofrece el framework anterior al nuevo framework. Para este proceso se recomienda el análisis de la solución anterior y ver como se puede mejorar con el uso del nuevo framework. En especial, revisar la parte de configuración asociada a la funcionalidad para introducir el uso de CoC e ID, ya que el antiguo framework requiere configuración para todo.

Referencias

[1] Dybox, http://www.dybox.cl/

[2] Guo-ping, L. et al. The "Core-Periphery" pattern of the globalization of electronic commerce. ACM International Conference Proceeding Series; Vol. 113 (2005) 66 – 69.

[3] Stoltzfus, K. Motivations for implementing e-government: an investigation of the global phenomenon. ACM International Conference Proceeding Series; Vol. 89 (2005) 333 – 338.

[4] Framework, http://es.wikipedia.org/wiki/Framework

[5] Java 1.6, http://java.sun.com/javase/downloads/index.jsp

[6] Java 1.5 mejoras, http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html

[7] CoC, Convención por sobre Configuración, http://en.wikipedia.org/wiki/Convention_over_Configuration; http://www.javalobby.org/java/forums/t65305.html

[8] Inyección de Dependencia, http://www.martinfowler.com/articles/injection.html

[9] PicoContainer, http://www.picocontainer.org/

[10] SpringFramework, http://www.springframework.org/

[11] Hibernate, http://www.hibernate.org/

[12] JUnit, http://www.junit.org/

[13] AOP, http://en.wikipedia.org/wiki/Aspect-oriented_programming

[14] MVC, http://en.wikipedia.org/wiki/Model-view-controller

[15] XML, http://www.w3.org/XML/

[16] XSLT, http://www.w3.org/TR/xslt

[17] JSP, http://java.sun.com/products/jsp/

[18] JSF, http://java.sun.com/javaee/javaserverfaces/

[19] SOA, http://en.wikipedia.org/wiki/Service-oriented_architecture

[20] Web Service, http://www.w3.org/2002/ws/

[21] Java Web estándar. http://java.sun.com/j2ee/reference/whitepapers/j2ee_guide.pdf

[22] Java Servlet, http://java.sun.com/products/servlet/

[23] Java EJB, http://java.sun.com/products/ejb/

[24] John Wiley & Sons, J2EE Development Without EJB, Expert One-on-One; (2004), ISBN: 978-0764558313

[25] Contenedores livianos y pesados, http://www.ibm.com/developerworks/java/library/os-lightweight2/

[26] Revisión SpringFramework, http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework

[27] DNA, http://dna.codehaus.org/

[28] Excalibur, http://excalibur.apache.org/index.html

[29] Soto, http://www.sapia-oss.org/projects/soto/site/html/home.html

[30] ORM, http://www.agiledata.org/essays/mappingObjects.html

[31] Object-Relational Impedance Mismatch, http://www.agiledata.org/essays/impedanceMismatch.html

[32] Java Annotations, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[33] MVC, Trygve Reenskaug, http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[34] Struts, http://struts.apache.org/

[35] ¿Qué es un controlador?, http://c2.com/cgi/wiki?WhatsaControllerAnyway

[36] EDP, http://c2.com/cgi/wiki?EventDrivenProgramming

[37] Erl, Thomas, Service-oriented Architecture: Concepts, Technology, and Design; (2005), ISBN: 0-13-185858-0

[38] Axis, http://ws.apache.org/axis/

[39] Rendimiento Axis, http://atmanes.blogspot.com/2006/08/axis-1x-or-axis2.html, http://www.sosnoski.com/presents/cleansoap/results.html

[40] XFire, http://xfire.codehaus.org/

[41] Celtix, http://celtix.objectweb.org/

[42] CXF, http://cxf.apache.org/

[43] *Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems*, *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)* September, 1990, ACM por William F. Opdyke y Ralph E. Johnson

[44] TDD, http://www.agiledata.org/essays/tdd.html

[45] Properties, http://java.sun.com/docs/books/tutorial/essential/environment/properties.html

[46] Joshua Fox, When is a Singleton not a Singleton, http://java.sun.com/developer/technicalArticles/Programming/singletons/

[47] JAX-WS, https://jax-ws.dev.java.net/, http://jcp.org/en/jsr/detail?id=224

[48] Java Reflection, http://java.sun.com/docs/books/tutorial/reflect/

[49] JavaBean, http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html

[50] Template Method, http://en.wikipedia.org/wiki/Template_method_pattern

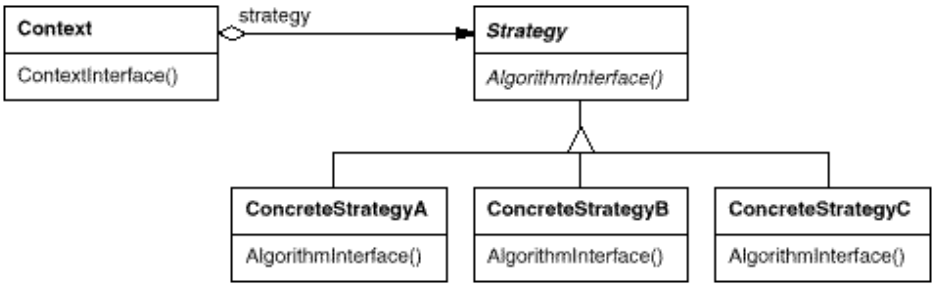
[51] Java Expression Language, http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html

[52] ICEFaces, http://www.icefaces.org/

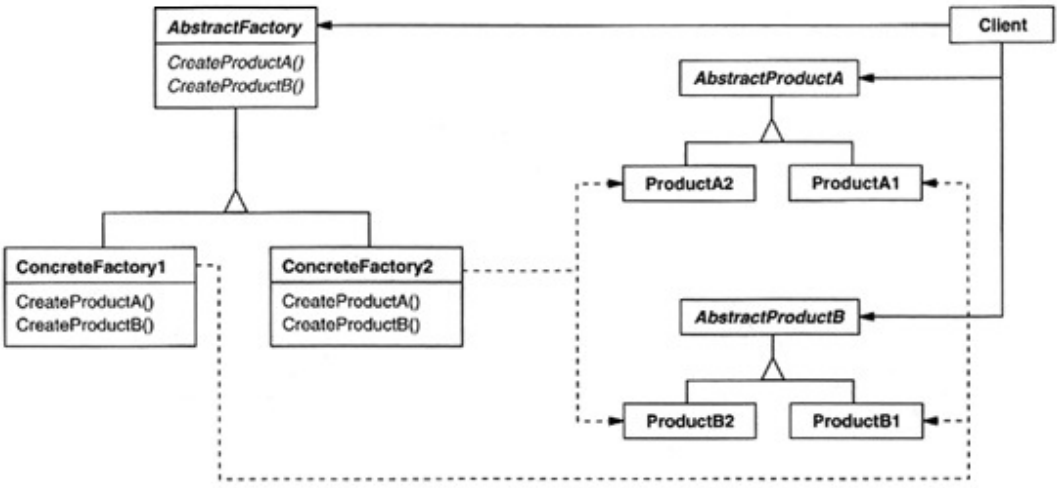
Apéndice

Patrones de Diseño usados en el desarrollo, ya sea directa o indirectamente.

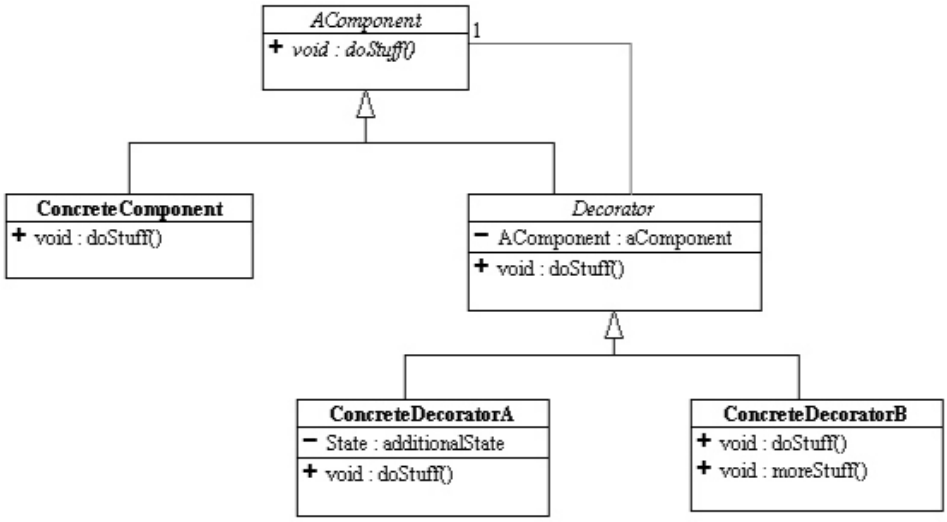
- Strategy



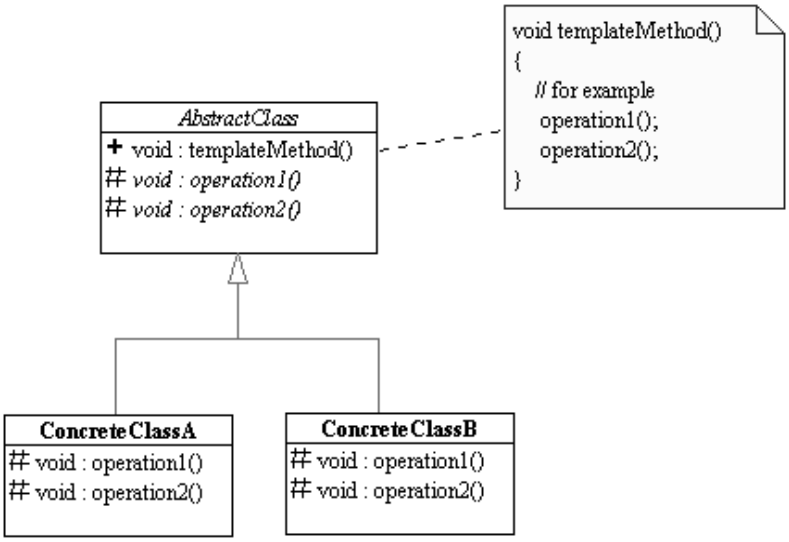
- Abstract Factory



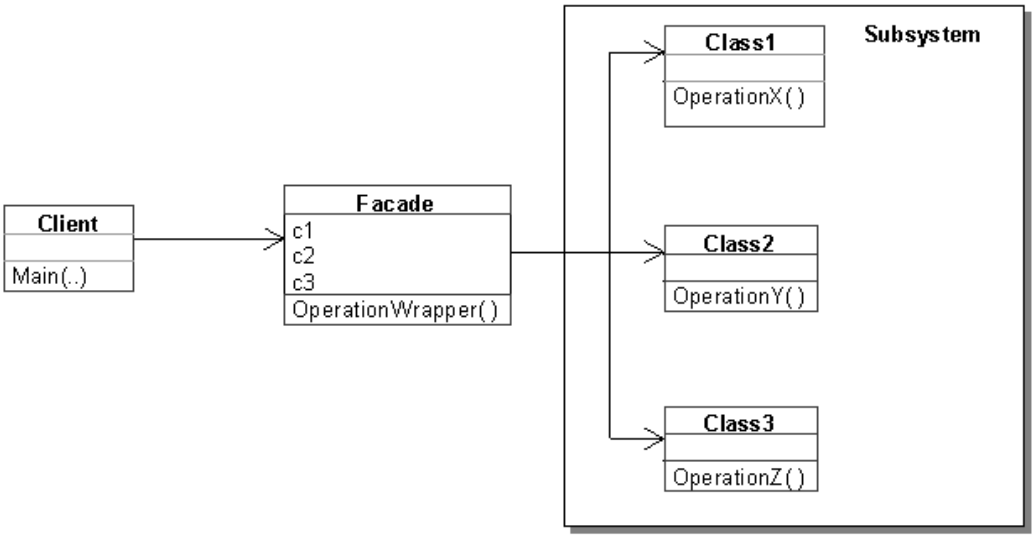
- Decorator



- Template Method



- Facade


Subsystem
Class1
OperationX()
Client
Main(..)
Facade
c1
c2
c3
OperationWrapper()
Class2
OperationY()
Class3
OperationZ()