PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

ARQUITECTURA DE SOFTWARE HÍBRIDA BASADA EN SPRING FRAMEWORK Y EJB 3.

JOSÉ GUILLERMO CARTES STUARDO

INFORME FINAL DEL PROYECTO
PARA OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO CIVIL INFORMÁTICO

DICIEMBRE 2011

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

ARQUITECTURA DE SOFTWARE HÍBRIDA BASADA EN SPRING FRAMEWORK Y EJB 3.

JOSÉ GUILLERMO CARTES STUARDO

Profesor Guía: **José Miguel Rubio León**

Profesor Correferente: **Wenceslao Palma Muñoz**

Carrera: **Ingeniería de Civil Informática**

DICIEMBRE 2011

Dedicatoria

A mi Novia, Familia, profesor guía, profesores, amigos,

y todas las personas que me ayudaron a completar

esta etapa de mi vida, simplemente muchas gracias.

José Guillermo Cartes Stuardo

Índice

Resumen i

Abstract i

Lista de Figuras ii

Lista de Tablas iv

1 Descripción del Proyecto 2

1.1 Introducción 2

1.2 Definición de Objetivos 3

1.2.1 Objetivo General 3

1.2.2 Objetivos Específicos 3

1.3 Plan de Trabajo 3

1.4 Metodología 4

1.5 Motivaciones del Proyecto 5

1.6 Arquitectura de Software 5

1.6.1 Historia 5

1.6.2 Definiciones 12

1.6.3 Campos de la Arquitectura de Software 14

1.6.4 Modalidades y tendencias 17

1.6.5 Arquitectura EJB 3 v/s Arquitectura Spring Framework 20

1.6.5.1 EJB 3 20

1.6.5.2 Spring Framework 21

1.6.6 Métricas de comparación 22

1.7 Caso de estudio 24

2 Spring, un Framework de aplicación 25

2.1 Introducción e historia 25

2.2 Arquitectura de Spring 26
2.2.1 Spring Core 26
2.2.1.1 Bean Factory 26
2.2.1.2 Inversion of Control 27
2.2.1.3 Dependency Injection 28
2.2.2 Spring Context 29
2.2.2.1 Application Context 29
2.2.3 Spring AOP 29
2.2.4 Spring ORM 30
2.2.5 Spring DAO 31
2.2.5.1 DAO y JDBC 31
2.2.6 Spring Web 33
2.2.7 Spring Web MVC 33
2.2.7.1 Dispatcher Servlet 35
2.2.7.2 Handler Mappings 35
2.2.7.3 View Resolvers 36
2.2.7.4 Controladores 37
2.2.7.5 Procesamiento de formularios 39
2.3 Complementos de Spring Framework. 41
2.3.1 Hibernate. 41
2.3.1.1 Características 42
2.3.1.2 Historia 42
3 Enterprise JavaBeans (EJB) 43
3.1 Definición 43
3.2 Desarrollo basado en componentes 43

3.3 Servicios proporcionados por el contenedor EJB 44
3.4 Funcionamiento de los componentes EJB 45
3.5 La arquitectura EJB en detalle 46
3.5.1 Repaso de RMI 46
3.5.1.1 Stubs, skeletons y paso de argumentos 46
3.5.1.2 Paso de argumentos 48
3.5.2 EJB y RMI 50
3.5.3 Interfaces locales y remotas 53
3.5.4 Funcionamiento de la clase home 54
3.6 Tipos de Enterprise JavaBeans 56
3.6.1 Beans de sesión 56
3.6.1.1 Beans de sesión sin estado 56
3.6.1.2 Beans de sesión con estado 57
3.6.2 Beans de entidad 58
3.6.2.1 Diferencias con los beans de sesión 59
3.6.3 Beans dirigidos por mensajes 60
3.6.3.1 Diferencias con los beans de sesión y de entidad 60
3.7 La arquitectura de los distintos tipos de beans 61
3.7.1 Beans de sesión sin estado 61
3.7.2 Beans de sesión con estado 63
3.7.3 Beans de entidad 65
3.7.4 Beans dirigidos por mensajes 67
3.8 Despliegue de aplicaciones y beans 67
4 Arquitecturas Propuestas 70
4.1 Arquitectura de Spring propuesta 70

4.1.1 MVC ... 72

4.1.2 Negocio ... 72

4.1.3 Acceso a datos ... 73

4.2 Arquitectura de EJB3 Propuesta ... 74

5 Análisis de Resultados ... 75

5.1 Resultados EJB v/s Spring Framework ... 75

5.2 Resultados de la Arquitectura Hibrida ... 77

6 Conclusiones y Trabajo Futuro ... 80

7 Referencias ... 81

8 Anexos ... 82

Resumen

En este informe, se muestra el desarrollo de arquitecturas de 2 tecnologías Importantes: Spring Framework y EJB3. Con el objetivo de poder comparar de acuerdo a ciertas métricas, el desempeño de arquitecturas diseñadas frente a un problema en específico.

Al obtener los resultados de estas métricas, se podrán obtener las ventajas de cada una de estas tecnologías, por lo que se pretende, diseñar una nueva arquitectura integrando estas 2 tecnologías, obteniendo así, una arquitectura mucho más robusta.

Palabras Claves: Arquitectura de Software, Spring Framework, EJB 3

Abstract

In this report, we show the development of architectures of 2 major technologies: Spring Framework and EJB3. In order to compare according to certain metrics, performance architectures designed against a specific problem.

Upon obtaining the results of these measurements, it can obtain the advantages of each of these technologies, it is intended, designing a new architecture integrating these two technologies, thus obtaining a more robust architecture.

Keywords: Software Architecture, Spring Framework, EJB 3

Lista de Figuras

Figura 1: Arquitectura 3 Capas 2

Figura 2: Diagrama metodología 4

Figura 3: Diagrama general de caso de uso, para el sistema propuesto 24

Figura 4: Arquitectura de Spring 26

Figura 5: Arquitectura básica del Web MVC de Spring 34

Figura 6: Ciclo de vida de un request 34

Figura 7: Controladores que provee Spring 38

Figura 8: Representación de alto nivel del funcionamiento de los enterprise beans 45

Figura 9: Ejemplo con el objeto remoto SaludoImpl 47

Figura 10: Ejemplo de la omisión del objeto skeleton 47

Figura 11: Estructura de clases e interfaces para definir una clase remota llamada Saludo 48

Figura 12: Objeto remoto devuelto en una llamada a otro objeto 49

Figura 13: Diferencia entre EJB y RMI 50

Figura 14: Métodos remotos del objeto home 51

Figura 15: Estructura de Clases del ejemplo 52

Figura 16: Funcionamiento de la clase home 55

Figura 17: Solicitud de clientes de servicios a un bean de sesión sin estado 62

Figura 18: Ciclo de vida del contenedor EJB 63

Figura 19: Solicitud de clientes de servicios a un bean de sesión con estado 64

Figura 20: Solicitud de clientes de servicios a un bean de entidad 65

Figura 21: Pasos del ciclo de vida del bean de entidad 66

Figura 22: Pasos del ciclo de vida del bean dirigido por mensajes 67

Figura 23: Diagrama de componentes de UML, para la propuesta de Spring 71

Figura 24: Esquema de herencia de los Controllers 72

Figura 25: Diagrama de la capa de Negocio 73

Figura 26: Diagrama de componentes para la arquitectura de EJB 74

Lista de Tablas

Tabla 1: Plan de Trabajo 3

Tabla 2: Métricas para Comparar 22

Tabla 3: Descripción de Clases de un Bean 52

Tabla 4: Resultados de Comparativa 75

Tabla 5: Resultados de Integración de Tecnologías 77

1 Descripción del Proyecto

1.1 Introducción

Las aplicaciones basadas en el lenguaje de programación JAVA comprenden una parte significativa del universo de desarrollo de software. Java fue inicialmente creado por SUN Corporation y ha logrado dentro la comunidad Open-Source la aceptación y el soporte de verdaderos gigantes de las TI como BEA Systems, IBM Corporation y JBOSS (Red Hat). JAVA es especialmente popular en el desarrollo de grandes aplicaciones empresariales. Las comparativas muestran que el 25,3% de las grandes compañías usan JAVA en sus aplicaciones más importantes (octubre 2005). La plataforma Java 2 Enterprise Edition (J2EE) provee grandes oportunidades para el desarrollo de sistemas distribuidos, se utiliza en la mayoría de las aplicaciones empresariales y la tecnología Enterprise JavaBeans (EJB) es una parte muy importante de la misma. Usualmente la arquitectura de una aplicación J2EE contiene varias capas separadas como se puede apreciar en la figura 1. La capa de servidor típicamente contiene componentes de servidor con lógica de negocio, estos son manejados por un contenedor EJB (de acuerdo con la implementación de la especificación EJB). El contenedor EJB es parte del servidor de aplicaciones (usualmente el contenedor EJB y el servidor de aplicaciones no pueden ser separados y son proporcionados por el mismo proveedor). El servidor de aplicaciones provee el ciclo de vida de los componentes, así como servicios de seguridad y manejo de transacciones.

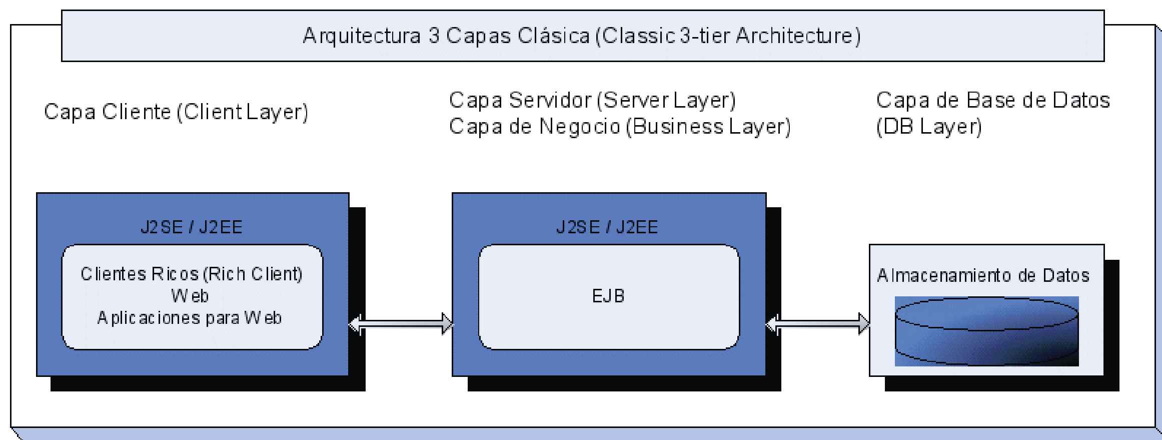

Figura 1: Arquitectura 3 Capas

Desafortunadamente, las versiones anteriores de EJB fueron demasiado complejas y nuevos componentes de manejo tecnológico aparecieron. Spring Framework, en su versión 1.0 liberada en Marzo de 2004, es un producto gratuito de carácter Open-Source, el mismo que es un contenedor liviano de componentes de negocio que pueden ser utilizados como una alternativa frente a EJB. En general, Spring Framework provee algunos servicios adicionales tales como Spring Web Model View Controller (MVC), el cual será visto en este documento.

En los capítulos 1.5 y 1.6 se verá la motivación del proyecto, además de las arquitecturas para ambas tecnologías.

En el capítulo 2 se ahondará con mayor detalle en la tecnología de Spring Framework.

En el capítulo 3 se ahondará con gran detalle en la tecnología EJB 3.

1.2 Definición de Objetivos

Para el desarrollo de este trabajo de título, es necesario tener claro cuál o cuáles serán los objetivos que se deben llevar a cabo para que éste se realice de la mejor manera, cumpliendo con todas las expectativas que se han puesto en él.

1.2.1 Objetivo General

El objetivo principal es proponer una arquitectura para desarrollo de software basada en la comparación de las arquitecturas de EJB 3 y de Spring Framework.

1.2.2 Objetivos Específicos

- Analizar las arquitecturas de Spring Framework y de EJB 3.
- Definir y diseñar un caso de estudio para aplicar las arquitecturas analizadas.
- Diseñar e implementar una arquitectura de software, basada en Spring Framework, para el caso de estudio propuesto.
- Diseñar e implementar una arquitectura de software, basada en EJB 3, para el caso de estudio propuesto.
- Validar y comparar las arquitecturas propuestas.
- Diseñar e implementar una arquitectura que integre las ventajas de las arquitecturas propuestas.
- Validar la arquitectura propuesta mediante el caso de estudio.

1.3 Plan de Trabajo

Para desarrollar este trabajo, se decidió optar por la metodología basada en prototipos incrementales (será explicado más profundamente en la sección 1.4 Metodologías). Además, en la planificación se plantea 2 iteraciones que tienen como resultado dos prototipos incrementales, que si pasan la etapa de validación y pruebas, serán integrados al sistema final.

Tabla 1: Plan de Trabajo

Tarea	Fecha Inicio	Fecha Término
Analizar la arquitectura de Spring Framework	02-08-2010	20-08-2010
Analizar la arquitectura de EJB 3	20-08-2010	03-09-2010
Definir caso de estudio	03-09-2010	24-09-2010
Delimitar el caso de estudio	24-09-2010	08-10-2010
Definir funcionalidades del caso de estudio	08-10-2010	22-10-2010
Diseñar arquitectura basa en Spring Framework para el caso en estudio	22-10-2010	05-11-2010
Diseñar arquitectura basa en EJB 3 para el caso en estudio	05-11-2010	19-11-2010
Implementar una Arquitectura basada en Spring Framework para el caso en estudio	14-03-2011	28-03-2011

Implementar una Arquitectura basada en EJB 3 para el caso en estudio	28-03-2011	11-04-2011
Validar las arquitecturas	11-04-2011	25-04-2011
Comparar las arquitecturas	25-04-2011	23-05-2011
Diseñar arquitectura integrando ambas tecnologías	23-05-2011	06-06-2011
Implementar arquitectura propuesta	06-06-2011	20-06-2011
Validar la arquitectura mediante el caso de estudio	20-06-2011	11-07-2011

1.4 Metodología

La elección de una metodología o modelo de proceso es una de las determinaciones más relevantes en un proyecto de desarrollo de software. Son importantes pues establecen las actividades necesarias para transformar los requerimientos que plantea un usuario o un cliente en un producto software de calidad. Es por esto que se debe elegir correctamente el modelo a utilizar, pues la mala elección, probablemente reduzca la calidad o la utilidad del producto de software que se va a desarrollar.

Muchos autores llegan a la conclusión de que no existe una metodología de desarrollo ideal, muchas de las empresas generan su propio modelo de desarrollo (que puede ser una mezcla o adaptación de uno o varios modelos) o simplemente no tienen ninguna metodología para los proyectos software. Se hace necesario buscar aquel que sea más conveniente y que cumpla de mejor manera con las necesidades del proyecto, o en defecto adaptar un modelo a las necesidades personales.

En esta tesis se ocupó la metodología de desarrollo basado en prototipos evolutivos para el caso del desarrollo de cada una de las arquitecturas a evaluar (EJB y Spring Framework), y por ende, de la arquitectura hibrida resultante de las dos arquitecturas mencionadas.

Los prototipos evolutivos, permiten asimilar las tecnologías que se van a ocupar, a medida que se desarrolla el trabajo de título, integrando las funcionalidades de cada iteración al sistema final. Además permiten afrontar de mejor manera las modificaciones o sugerencias que surjan durante las distintas instancias de evaluación del proyecto.

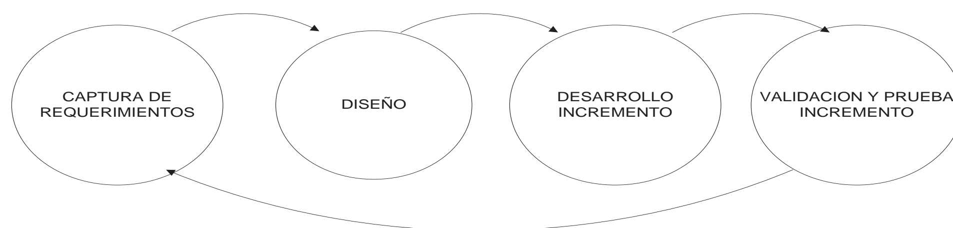

Figura 2: Diagrama metodología

Como se ve en la Figura 2, los elementos que componen la metodología son:

- Captura de Requerimientos: Esta etapa es muy importante, pues se debe tener claro los requerimientos para desarrollar cada incremento.
- Diseño: Se desarrolla diagrama general de caso de usos que permite entender de mejor manera los requerimientos. Además se realiza el diseño arquitectónico de la solución al problema.
- Desarrollo de incremento: El incremento se basara en los requerimientos que se eligieron y en el posterior diseño de la solución. Se utilizaran las distintas tecnologías que se seleccionaron en los análisis anteriores.
- Validación y prueba de incremento: El prototipo se probara y si cumple los requerimientos, se une al sistema completo. Deberá pasar un plan de pruebas acorde a los requisitos que se plantearon en la iteración.

1.5 Motivaciones del Proyecto

La arquitectura del software hoy en día es fundamental a la hora de estructurar el desarrollo de nuestro software, puesto que la elección de esta misma, puede hacer que nuestro sistema, sea más eficiente, seguro, estable, entre otras cosas. Es por esto que se han tomado 2 tecnologías que hoy en día, están predominando el comercio informático.

La idea en general, es presentar 2 arquitecturas robustas que sirvan para diferentes ámbitos y además poder obtener lo mejor de cada una de estas tecnologías e integrar en una arquitectura de mayor nivel.

En el capítulo siguiente, se muestra los conceptos asociados a las arquitecturas de software, los cuales serán la base para el desarrollo de la propuesta.

1.6 Arquitectura de Software

En los inicios de la informática, la programación se consideraba un arte y se desarrollaba como tal, debido a la dificultad que entrañaba para la mayoría de las personas, pero con el tiempo se han ido descubriendo y desarrollando formas y guías generales, con base a las cuales se puedan resolver los problemas. A estas, se les ha denominado Arquitectura de Software, porque, semejanza de los planos de un edificio o construcción, estas indican la estructura, funcionamiento e interacción entre las partes del software. En el libro "An introduction to Software Architecture", David Garlan y Mary Shaw definen que la Arquitectura es un nivel de diseño que hace foco en aspectos "más allá de los algoritmos y estructuras de datos de la computación; el diseño y especificación de la estructura global del sistema es un nuevo tipo de problema". [6]

1.6.1 Historia

Todavía no se ha escrito una historia aceptable de la Arquitectura de Software (AS). Desde que Mary Shaw o David Garlan reseñaran escuetamente la prehistoria de la especialidad a principios de los 90, los mismos párrafos han sido reutilizados una y otra vez en la literatura, sin mayor exploración de las fuentes referidas en la reseña primaria y con prisa por ir al grano, que usualmente no es de carácter histórico. En este estudio se ha optado, más bien, por inspeccionar las fuentes más de cerca, con el objeto de señalar las supervivencias y las re-semantizaciones que han experimentado las ideas fundadoras en la AS contemporánea, definir con mayor claridad el contexto, entender que muchas contribuciones que pasaron por

complementarias han sido en realidad antagónicas y comprender mejor por qué algunas ideas que surgieron hace cuatro décadas demoraron un cuarto de siglo en materializarse.

Esta decisión involucra algo más que el perfeccionamiento de la lectura que pueda hacerse de un conjunto de acontecimientos curiosos. Las formas divergentes en que se han interpretado dichas ideas, después de todo, permiten distinguir corrientes de pensamiento diversas, cuyas diferencias distan de ser triviales a la hora de plasmar las ideas en una metodología. Todo lo que parece ser un simple elemento de juicio, no pocas veces implica una disyuntiva. Situar las inflexiones de la breve historia de la AS en un contexto temporal, asimismo, ayudará a comprender mejor cuáles son sus contribuciones perdurables y cuáles sus manifestaciones contingentes al espíritu de los tiempos y a las modas tecnológicas que se han ido sucediendo.

Si bien la AS acostumbra remontar sus antecedentes al menos hasta la década de 1960, su historia no ha sido tan continua como la del campo más amplio en el que se inscribe, la ingeniería de software. Después de las tempranas inspiraciones del legendario Edsger Dijkstra, de David Parnas y de Fred Brooks, la AS quedó en estado de vida latente durante unos cuantos años, hasta comenzar su expansión explosiva con los manifiestos de Dewayne Perry de AT&T Bell Laboratories de New Jersey y Alexander Wolf de la Universidad de Colorado. Puede decirse que Perry y Wolf fundaron la disciplina, y su llamamiento fue respondido en primera instancia por los miembros de lo que podría llamarse la escuela estructuralista de Carnegie Mellon; David Garlan, Mary Shaw, Paul Clements, Robert Allen.

Se trata entonces de una práctica joven, de apenas unos doce años de trabajo constante, que en estos momentos experimenta una nueva ola creativa en el desarrollo cabal de sus técnicas en la obra de Rick Kazman, Mark Klein, Len Bass y otros metodólogos en el contexto del SEI, en la misma universidad. A comienzos del siglo XXI comienzan ya a discernirse tendencias, cuyas desavenencias mutuas todavía son leves, al menos una en el sur de California (Irvine y LosÁngeles) con Nenad Medvidovic, David Rosenblum y Richard Taylor, otra en el SRI de Menlo Park con Mark Moriconi y sus colegas y otra más vinculada a las recomendaciones formales de la IEEE y los trabajos de Rich Hilliard. Hoy se percibe también un conjunto de posturas europeas que enfatizan mayormente cuestiones metodológicas vinculadas con escenarios y procuran inscribir la arquitectura de software en el ciclo de vida, comenzando por la elicitación de los requerimientos. Antes de Perry y Wolf, empero, se formularon ideas que serían fundamentales para la disciplina ulterior. Comencemos entonces por el principio, aunque siempre cabrá la posibilidad de discutir cuál puede haber sido el momento preciso en el que todo comenzó.

Cada vez que se narra la historia de la arquitectura de software (o de la ingeniería de software, según el caso), se reconoce que en un principio, hacia 1968, Edsger Dijkstra, de la Universidad Tecnológica de Eindhoven en Holanda y Premio Turing 1972, propuso que se establezca una estructuración correcta de los sistemas de software antes de lanzarse a programar, escribiendo código de cualquier manera. Dijkstra, quien sostenía que las ciencias de la computación eran una rama aplicada de las matemáticas y sugería seguir pasos formales para descomponer problemas mayores, fue uno de los introductores de la noción de sistemas operativos organizados en capas que se comunican sólo con las capas adyacentes y que se superponen "como capas de cebolla". Inventó o ayudó a precisar además docenas de

conceptos: el algoritmo del camino más corto, los stacks, los vectores, los semáforos, los abrazos mortales. De sus ensayos arranca la tradición de hacer referencia a "niveles de abstracción" que ha sido tan común en la arquitectura subsiguiente. Aunque Dijkstra no utiliza el término arquitectura para describir el diseño conceptual del software, sus conceptos sientan las bases para lo que luego expresarían Niklaus Wirth como stepwise refinement y DeRemer y Kron como programming-in-the large (o programación en grande), ideas que poco a poco irían decantando entre los ingenieros primero y los arquitectos después.

En la conferencia de la NATO de 1969, un año después de la sesión en que se fundara la ingeniería de software, P. I. Sharp formuló estas sorprendentes apreciaciones comentando las ideas de Dijkstra: "Pienso que tenemos algo, aparte de la ingeniería de software, algo de lo que hemos hablado muy poco pero que deberíamos poner sobre el tapete y concentrar la atención en ello. Es la cuestión de la arquitectura de software. La arquitectura es diferente de la ingeniería. Como ejemplo de lo que quiero decir, echemos una mirada a OS/360. Partes de OS/360 están extremadamente bien codificadas. Partes de OS, si vamos al detalle, han utilizado técnicas que hemos acordado constituyen buena práctica de programación. La razón de que OS sea un amontonamiento amorfo de programas es que no tuvo arquitecto. Su diseño fue delegado a series de grupos de ingenieros, cada uno de los cuales inventó su propia arquitectura. Y cuando esos pedazos se clavaron todos juntos no produjeron una tersa y bella pieza de software".

Sharp continúa su alegación afirmando que con el tiempo probablemente llegue a hablarse de "la escuela de arquitectura de software de Dijkstra" y se lamenta que en la industria de su tiempo se preste tan poca o ninguna atención a la arquitectura. La frase siguiente también es extremadamente visionaria: "Lo que sucede es que las especificaciones de software se consideran especificaciones funcionales. Sólo hablamos sobre lo que queremos que haga el programa. Es mi creencia que cualquiera que sea responsable de la implementación de una pieza de software debe especificar más que esto. Debe especificar el diseño, la forma; y dentro de ese marco de referencia, los programadores e ingenieros deben crear algo. Ningún ingeniero o programador, ninguna herramienta de programación, nos ayudará, o ayudará al negocio del software, a maquillar un diseño feo. El control, la administración, la educación y todas las cosas buenas de las que hablamos son importantes; pero la gente que implementa debe entender lo que el arquitecto tiene en mente".

Nadie volvió a hablar del asunto en esa conferencia, sin embargo. Por unos años, "arquitectura" fue una metáfora de la que se echó mano cada tanto, pero sin precisión semántica ni consistencia pragmática. En 1969 Fred Brooks Jr y Ken Iverson llamaban arquitectura a la estructura conceptual de un sistema en la perspectiva del programador. En 1971, C. R. Spooner tituló uno de sus ensayos "Una arquitectura de software para los 70s", sin que la mayor parte de la historiografía de la AS registrara ese antecedente.

En 1975, Brooks, diseñador del sistema operativo OS/360 y Premio Turing 2000, utilizaba el concepto de arquitectura del sistema para designar "la especificación completa y detallada de la interfaz de usuario" y consideraba que el arquitecto es un agente del usuario, igual que lo es quien diseña su casa [Bro75], empleando una nomenclatura que ya nadie aplica de ese modo. En el mismo texto, identificaba y razonaba sobre las estructuras de alto nivel y reconocía la importancia de las decisiones tomadas a ese nivel de diseño. También distinguía

entre arquitectura e implementación; mientras aquella decía qué hacer, la implementación se ocupa de cómo. Aunque el concepto de AS actual y el de Brooks difieren en no escasa medida, el texto de Brooks "The mythical man-month" sigue siendo, un cuarto de siglo más tarde, el más leído en ingeniería de software. Se ha señalado que Dijkstra y Brooks, el primero partidario de un formalismo matemático y el segundo de considerar las variables humanas, constituyen dos personalidades opuestas, que se sitúan en los orígenes de las metodologías fuertes y de las heterodoxias ágiles, respectivamente; pero eso será otra historia.

Una novedad importante en la década de 1970 fue el advenimiento del diseño estructurado y de los primeros modelos explícitos de desarrollo de software. Estos modelos comenzaron a basarse en una estrategia más orgánica, evolutiva, cíclica, dejando atrás las metáforas del desarrollo en cascada que se inspiraban más bien en la línea de montaje de la ingeniería del hardware y la manufactura. Surgieron entonces las primeras investigaciones académicas en materia de diseño de sistemas complejos o "intensivos". Poco a poco el diseño se fue independizando de la implementación, y se forjaron herramientas, técnicas y lenguajes de modelado específicos.

En la misma época, otro precursor importante, David Parnas, demostró que los criterios seleccionados en la descomposición de un sistema impactan en la estructura de los programas y propuso diversos principios de diseño que debían seguirse a fin de obtener una estructura adecuada. Parnas desarrolló temas tales como módulos con ocultamiento de información, estructuras de software y familias de programas, enfatizando siempre la búsqueda de calidad del software, medible en términos de economías en los procesos de desarrollo y mantenimiento. Aunque Dijkstra, con sus frases lapidarias y memorables, se ha convertido en la figura legendaria por excelencia de los mitos de origen de la AS, Parnas ha sido sin duda el introductor de algunas de sus nociones más esenciales y permanentes.

En 1972, Parnas publicó un ensayo en el que discutía la forma en que la modularidad en el diseño de sistemas podía mejorar la flexibilidad y el control conceptual del sistema, acortando los tiempos de desarrollo. Introdujo entonces el concepto de ocultamiento de información (information hiding), uno de los principios de diseño fundamentales en diseño de software aún en la actualidad. La herencia de este concepto en la ingeniería y la arquitectura ulterior es inmensa, y se confunde estrechamente con la idea de abstracción. En la segunda de las descomposiciones que propone Parnas comienza a utilizarse el ocultamiento de información como criterio. "Los módulos ya no se corresponden con las etapas de procesamiento. Cada módulo en la segunda descomposición se caracteriza por su conocimiento de una decisión de diseño oculta para todos los otros módulos. Su interfaz o definición se escoge para que revele tan poco como sea posible sobre su forma interna de trabajo". Cada módulo deviene entonces una caja negra para los demás módulos del sistema, los cuales podrán acceder a aquél a través de interfaces bien definidas, en gran medida invariables. Es fácil reconocer en este principio ideas ya presentadas por Dijkstra en su implementación del "THEMultiprogramming System". Pero la significación del artículo de 1972 radica en la idea de Parnas de basar la técnica de modularización en decisiones de diseño, mientras que los "niveles de abstracción" de Dijkstra involucraban más bien (igual que su famosa invectiva en contra del Go-to) técnicas de programación.

El concepto de ocultamiento se fue mezclando con encapsulamiento y abstracción, tras algunos avatares de avance y retroceso. Los arquitectos más escrupulosos distinguen entre encapsulamiento y ocultamiento, considerando a aquél como una capacidad de los lenguajes de programación y a éste como un principio más general de diseño. De hecho, Parnas no hablaba en términos de programación orientada a objeto, sino de módulos y sub-rutinas, porque el momento de los objetos no había llegado todavía. El pensamiento de Parnas sobre familias de programas, en particular, anticipa ideas que luego habrían de desarrollarse a propósito de los estilos de arquitectura; una familia de programas es un conjunto de programas (no todos los cuales han sido construidos o lo serán alguna vez) a los cuales es provechoso o útil considerar como grupo. Esto evita el uso de conceptos ambiguos tales como "similitud funcional" que surgen a veces cuando se describen dominios. Por ejemplo, los ambientes de ingeniería de software y los juegos de video no se consideran usualmente en el mismo dominio, aunque podrían considerarse miembros de la misma familia de programas en una discusión sobre herramientas que ayuden a construir interfaces gráficas, que casualmente ambos utilizan.

Una familia de programas puede enumerarse en principio mediante la especificación del árbol de decisión que se atraviesa para llegar a cada miembro de la familia. Las hojas del árbol representan sistemas ejecutables. El concepto soporta la noción de derivar un miembro de la familia a partir de uno ya existente. El procedimiento consiste en seguir hacia atrás el árbol hasta que se alcanza un nodo (punto de decisión) genealógicamente común a ambos, y luego proceder hacia abajo hasta llegar al miembro deseado. El concepto también soporta la noción de derivar varios miembros de la familia de un punto de decisión común, aclarando la semejanza y las diferencias entre ellos.

Las decisiones iniciales son las que más probablemente permanecerán constantes entre los miembros; las decisiones más tardías (cerca de las hojas) en un árbol prudentemente estructurado deberían representar decisiones susceptibles de cambiarse trivialmente, tales como los valores de tiempo de compilación o las constantes de tiempo de carga. La significación del concepto de familia de programas para la AS es que ella corresponde a las decisiones cerca del tope del árbol de decisión de Parnas. Es importante considerar que el árbol de Parnas es topdown no sólo porque se construye y recorre de lo general a lo particular, sino porque sus raíces se encuentran hacia arriba (o a la izquierda si el modelo es horizontal). No menos esencial es tener en cuenta que el árbol se ofreció como alternativa a métodos de descomposición basados en diagramas de flujo, por los que la AS no ha mostrado nunca demasiada propensión.

Decía Parnas que las decisiones tempranas de desarrollo serían las que probablemente permanecerían invariantes en el desarrollo ulterior de una solución. Esas "decisiones tempranas" constituyen de hecho lo que hoy se llamarían decisiones arquitectónicas. Como escriben Clements y Northrop en todo el desenvolvimiento ulterior de la disciplina permanecería en primer plano esta misma idea: "La estructura es primordial (structure matters), y la elección de la estructura correcta ha de ser crítica para el éxito del desarrollo de una solución". Ni duda cabe que "la elección de la estructura correcta" sintetiza, como ninguna otra expresión, el programa y la razón de ser de la AS.

En la década de 1980, los métodos de desarrollo estructurado demostraron no escalar suficientemente y fueron dejando el lugar a un nuevo paradigma, el de la programación orientada a objetos. En teoría, parecía posible modelar el dominio del problema y el de la solución en un lenguaje de implementación. La investigación que llevó a lo que después sería el diseño orientado a objetos puede remontarse incluso a la década de 1960 con Simula, un lenguaje de programación de simulaciones, el primero que proporcionaba tipos de datos abstractos y clases, y después fue refinada con el advenimiento de Smalltalk. Paralelamente, hacia fines de la década de 1980 y comienzos de la siguiente, la expresión arquitectura de software comienza a aparecer en la literatura para hacer referencia a la configuración morfológica de una aplicación.

Mientras se considera que la ingeniería de software se fundó en 1968, cuando F.L. Bauer usó ese sintagma por primera vez en la conferencia de la OTAN de Garmisch, Alemania, la AS, como disciplina bien delimitada, es mucho más nueva de lo que generalmente se sospecha. El primer texto que vuelve a reivindicar las abstracciones de alto nivel, reclamando un espacio para esa reflexión y augurando que el uso de esas abstracciones en el proceso de desarrollo pueden resultar en "un nivel de arquitectura de software en el diseño" es uno de Mary Shaw seguido por otro llamado "Larger scale systems require higher level abstractions". Se hablaba entonces de un nivel de abstracción en el conjunto; todavía no estaban en su lugar los elementos de juicio que permitieran reclamar la necesidad de una disciplina y una profesión particulares.

El primer estudio en que aparece la expresión "arquitectura de software" en el sentido en que hoy lo conocemos es sin duda el de Perry y Wolf; ocurrió tan tarde como en 1992, aunque el trabajo se fue gestando desde 1989. En él, los autores proponen concebir la AS por analogía con la arquitectura de edificios, una analogía de la que luego algunos abusaron, otros encontraron útil y para unos pocos ha devenido inaceptable. El artículo comienza diciendo exactamente: "El propósito de este paper es construir el fundamento para la arquitectura de software. Primero desarrollaremos una intuición para la arquitectura de software recurriendo a diversas disciplinas arquitectónicas bien definidas. Sobre la base de esa intuición, presentamos un modelo para la arquitectura de software que consiste en tres componentes: elementos, forma y razón (rationale). Los elementos son elementos ya sea de procesamiento, datos o conexión. La forma se define en términos de las propiedades de, y las relaciones entre, los elementos, es decir, restricciones operadas sobre ellos. La razón proporciona una base subyacente para la arquitectura en términos de las restricciones del sistema, que lo más frecuente es que se deriven de los requerimientos del sistema. Discutimos los componentes del modelo en el contexto tanto de la arquitectura como de los estilos arquitectónicos...."

La declaración, como puede verse, no tiene una palabra de desperdicio: "Cada idea cuenta, cada intuición ha permanecido viva desde entonces". Los autores prosiguen reseñando el progreso de las técnicas de diseño en la década de 1970, en la que los investigadores pusieron en claro que el diseño es una actividad separada de la implementación y que requiere notaciones, técnicas y herramientas especiales. Los resultados de esta investigación comienzan ahora (decían en 1992) a penetrar el mercado en la forma de herramientas de ingeniería asistida por computadoras, CASE. Pero uno de los resultados del uso de estas herramientas ha sido que se produjo la absorción de las herramientas de diseño por los lenguajes de implementación. Esta integración ha tendido a esfumar, si es que no a confundir, la diferencia

entre diseño e implementación. En la década de 1980 se perfeccionaron las técnicas descriptivas y las notaciones formales, permitiéndonos razonar mejor sobre los sistemas de software. Para la caracterización de lo que sucederá en la década siguiente ellos formulan esta otra frase que ha quedado inscripta en la historia mayor de la especialidad: "La década de 1990, creemos, será la década de la arquitectura de software. Usamos el término "arquitectura" en contraste con "diseño", para evocar nociones de codificación, de abstracción, de estándares, de entrenamiento formal (de los arquitectos de software) y de estilo. Es tiempo de re-examinar el papel de la arquitectura de software en el contexto más amplio del proceso de software y de su administración, así como señalar las nuevas técnicas que han sido adoptadas".

Considerada como disciplina por mérito propio, la AS ha de ser, para ellos, beneficiosa como marco de referencia para satisfacer requerimientos, una base esencial para la estimación de costos y administración del proceso y para el análisis de las dependencias y la consistencia del sistema.

Dando cumplimiento a las profecías de Perry y Wolf, la década de 1990 fue sin duda la de la consolidación y diseminación de la AS en una escala sin precedentes. Las contribuciones más importantes surgieron en torno del instituto de ingeniería de la información de la Universidad Carnegie Mellon, antes que de cualesquiera organismos de industria. En la misma década, demasiado pródiga en acontecimientos, surge también la programación basada en componentes, que en su momento de mayor impacto impulsó a algunos arquitectos mayores, como Paul Clements, a afirmar que la AS promovía un modelo que debía ser más de integración de componentes pre-programados que de programación.

Un segundo gran tema de la época fue el surgimiento de los patrones, cristalizada en dos textos fundamentales, el de la Banda de los Cuatro en 1995 y la serie POSA desde 1996. El primero de ellos promueve una expansión de la programación orientada a objetos, mientras que el segundo desenvuelve un marco ligeramente más ligado a la AS. Este movimiento no ha hecho más que expandirse desde entonces. El originador de la idea de patrones fue Christopher Alexander, quien incidentalmente fue arquitecto de edificios; Alexander desarrolló en diversos estudios de la década de 1970 temas de análisis del sentido de los planos, las formas, la edificación y la construcción, en procura de un modelo constructivo y humano de arquitectura, elaborada de forma que tenga en cuenta las necesidades de los habitantes. El arquitecto debe ser un agente del usuario.

A lo largo de una cadena de intermediarios y pensadores originales, las ideas llegaron por fin a la informática diez años más tarde. Si bien la idea de arquitectura implícita en el trabajo actual con patrones está más cerca de la implementación y el código, y aunque la reutilización de patrones guarda estrecha relación con la tradición del diseño concreto orientado a objetos, algunos arquitectos emanados de la escuela de Carnegie Mellon formalizaron un acercamiento con esa estrategia. Tanto en los patrones como en la arquitectura, la idea dominante es la de re-utilización. A impulsos de otra idea mayor de la época (la crisis del software), la bibliografía sobre el impacto económico de la re-utilización alcanza hoy magnitudes de cuatro dígitos.

Como quiera que sea, la AS de este período realizó su trabajo de homogeneización de la terminología, desarrolló la tipificación de los estilos arquitectónicos y elaboró lenguajes de

descripción de arquitectura (ADLs), temas que en este estudio se tratan en documentos separados.

Uno de los acontecimientos arquitectónicos más importantes del año 2000 fue la hoy célebre tesis de Roy Fielding que presentó el modelo REST, el cual establece definitivamente el tema de las tecnologías de Internet y los modelos orientados a servicios y recursos en el centro de las preocupaciones de la disciplina. En el mismo año se publica la versión definitiva de la recomendación IEEE Std 1471, que procura homogeneizar y ordenar la nomenclatura de descripción arquitectónica y homologa los estilos como un modelo fundamental de representación conceptual.

En el siglo XXI, la AS aparece dominada por estrategias orientadas a líneas de productos y por establecer modalidades de análisis, diseño, verificación, refinamiento, recuperación, diseño basado en escenarios, estudios de casos y hasta justificación económica, redefiniendo todas las metodologías ligadas al ciclo de vida en términos arquitectónicos. Todo lo que se ha hecho en ingeniería debe formularse de nuevo, integrando la AS en el conjunto. La producción de estas nuevas metodologías ha sido masiva, y una vez más tiene como epicentro el trabajo del Software Engineering Institute en Carnegie Mellon. La aparición de las metodologías basadas en arquitectura, junto con la popularización de los métodos ágiles en general y Extreme Programming en particular, han causado un reordenamiento del campo de los métodos, hasta entonces dominados por las estrategias de diseño "de peso pesado". Después de la AS y de las tácticas radicales, las metodologías nunca volverán a ser las mismas.

La semblanza que se ha trazado no es más que una visión selectiva de las etapas recorridas por la AS. Los lineamientos de ese proceso podrían dibujarse de maneras distintas, ya sea enfatizando los hallazgos formales, las intuiciones dominantes de cada período o las diferencias que median entre la abstracción cualitativa de la arquitectura y las cuantificaciones que han sido la norma en ingeniería de software.

1.6.2 Definiciones

No es novedad que ninguna definición de la AS es respaldada unánimemente por la totalidad de los arquitectos. El número de definiciones circulantes alcanza un orden de tres dígitos, amenazando llegar a cuatro. De hecho, existen grandes compilaciones de definiciones alternativas o contrapuestas, como la colección que se encuentra en el SEI (http://www.sei.cmu.edu/architecture/definitions.html), a la que cada quien puede agregar la suya. En general, las definiciones entremezclan despreocupadamente (1) el trabajo dinámico de estipulación de la arquitectura dentro del proceso de ingeniería o el diseño (su lugar en el ciclo de vida), (2) la configuración o topología estática de sistemas de software contemplada desde un elevado nivel de abstracción y (3) la caracterización de la disciplina que se ocupa de uno de esos dos asuntos, o de ambos.

Una definición reconocida es la de Clements: "La AS es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista

arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión y la supresión o diferimiento del detalle inherente a la mayor parte de las abstracciones".

En una definición semejante, hay que aclararlo, la idea de "componente" no es la de la correspondiente tecnología de desarrollo (COM, CORBA Component Model, EJB), sino la de elemento propio de un estilo. Un componente es una cosa, una entidad, a la que los arquitectos prefieren llamar "componente" antes que "objeto", por razones que se verán en otros documentos de esta serie pero que ya es obvio imaginar cuáles han de ser.

A despecho de la abundancia de definiciones del campo de la AS, existe en general acuerdo de que ella se refiere a la estructura a grandes rasgos del sistema, estructura consistente en componentes y relaciones entre ellos. Estas cuestiones estructurales se vinculan con el diseño, pues la AS es después de todo una forma de diseño de software que se manifiesta tempranamente en el proceso de creación de un sistema; pero este diseño ocurre a un nivel más abstracto que el de los algoritmos y las estructuras de datos. En el que muchos consideran un ensayo seminal de la disciplina, Mary Shaw y David Garlan sugieren que dichas cuestiones estructurales incluyen organización a grandes rasgos y estructura global de control; protocolos para la comunicación, la sincronización y el acceso a datos; la asignación de funcionalidad a elementos del diseño; la distribución física; la composición de los elementos de diseño; escalabilidad y rendimiento; y selección entre alternativas de diseño.

En una definición tal vez demasiado amplia, David Garlan establece que la AS constituye un puente entre el requerimiento y el código, ocupando el lugar que en los gráficos antiguos se reservaba para el diseño. La definición "oficial" de AS se ha acordado que sea la que brinda el documento de IEEE Std 1471-2000, adoptada también por Microsoft, que reza así: "La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución".

Aunque las literaturas de ambos campos rara vez convergen, entendemos que es productivo contrastar esa definición con la de ingeniería de software, conforme al estándar IEEE 610.12.1990: "La aplicación de una estrategia sistemática, disciplinada y cuantificable al desarrollo, aplicación y mantenimiento del software; esto es, la aplicación de la ingeniería al software".

Obsérvese entonces que la noción clave de la arquitectura es la organización (un concepto cualitativo o estructural), mientras que la ingeniería tiene fundamentalmente que ver con una sistematicidad susceptible de cuantificarse.

Antes el número y variedad de definiciones existentes de AS, Mary Shaw y David Garlan proporcionaron una sistematización iluminadora, explicando las diferencias entre definiciones en función de distintas clases de modelos. Destilando las definiciones y los puntos de vista implícitos o explícitos, los autores clasifican los modelos de esta forma:

- Modelos estructurales: Sostienen que la AS está compuesta por componentes, conexiones entre ellos y (usualmente) otros aspectos tales como configuración, estilo, restricciones, semántica, análisis, propiedades, racionalizaciones, requerimientos,

necesidades de los participantes. El trabajo en esta área está caracterizada por el desarrollo de lenguajes de descripción arquitectónica (ADLs).

- Modelos de framework: Son similares a la vista estructural, pero su énfasis primario radica en la (usualmente una sola) estructura coherente del sistema completo, en vez de concentrarse en su composición. Los modelos de framework a menudo se refieren a dominios o clases de problemas específicos. El trabajo que ejemplifica esta variante incluye arquitecturas de software específicas de dominios, como CORBA, o modelos basados en CORBA, o repositorios de componentes específicos, como PRISM.
- Modelos dinámicos: Enfatizan la cualidad conductual de los sistemas. "Dinámico" puede referirse a los cambios en la configuración del sistema, o a la dinámica involucrada en el progreso de la computación, tales como valores cambiantes de datos.
- Modelos de proceso: Se concentran en la construcción de la arquitectura, y en los pasos o procesos involucrados en esa construcción. En esta perspectiva, la arquitectura es el resultado de seguir un argumento (script) de proceso. Esta vista se ejemplifica con el actual trabajo sobre programación de procesos para derivar arquitecturas.
- Modelos funcionales: Una minoría considera la arquitectura como un conjunto de componentes funcionales, organizados en capas que proporcionan servicios hacia arriba. Es tal vez útil pensar en esta visión como un framework particular.

Ninguna de estas vistas excluye a las otras, ni representa un conflicto fundamental sobre lo que es o debe ser la AS. Por el contrario, representan un espectro en la comunidad de investigación sobre distintos énfasis que pueden aplicarse a la arquitectura: sobre sus partes constituyentes, su totalidad, la forma en que se comporta una vez construida, o el proceso de su construcción. Tomadas en su conjunto, destacan más bien un consenso.

Independientemente de las discrepancias entre las diversas definiciones, es común entre todos los autores el concepto de la arquitectura como un punto de vista que concierne a un alto nivel de abstracción. Revisamos las diversas definiciones del concepto de abstracción en un apartado específico más adelante en este estudio.

Es casi seguro que la percepción de la AS que prevalece entre quienes no han tenido contacto con ella, así como los estereotipos dominantes sobre su naturaleza, o los nombres que se escogerían como sus personalidades más importantes, difieren sustancialmente de lo que es el caso en el interior de la especialidad. Este sería acaso un ejercicio digno de llevarse a cabo alguna vez.

1.6.3 Campos de la Arquitectura de Software

La AS es hoy en día un conjunto inmenso y heterogéneo de áreas de investigación teórica y de formulación práctica, por lo que conviene aunque más no sea enumerar algunos de sus campos y sus focos. Una semblanza semejante (de la que aquí se proporciona sólo un rudimento) dudosamente debería ser estática. La AS comenzó siendo una abstracción descriptiva puntual que en los primeros años no investigó de manera sistemática ni las relaciones que la vinculaban con los requerimientos previos, ni los pasos metodológicos a dar luego para comenzar a componer el diseño. Pero esa sincronicidad estructuralista no pudo sostenerse. Por el contrario, daría la impresión que, a medida que pasa el tiempo, la AS tiende a redefinir todos y cada uno de los aspectos de la disciplina madre, la ingeniería de software,

sólo que a un mayor nivel de abstracción y agregando una nueva dimensión reflexiva en lo que concierne a la fundamentación formal del proceso.

Hay unas pocas caracterizaciones en torno de las áreas que componen el territorio. David Garlan y Dewayne Perry, en su introducción al volumen de abril de 1995 de IEEE Transactions on Software Engineering dedicado a la AS, en el cual se delinean las áreas de investigación más promisorias, enumeran las siguientes:

- Lenguajes de descripción de arquitecturas
- Fundamentos formales de la AS (bases matemáticas, caracterizaciones formales de propiedades extra-funcionales tales como mantenibilidad, teorías de la interconexión, etcétera).
- Técnicas de análisis arquitectónicas
- Métodos de desarrollo basados en arquitectura
- Recuperación y reutilización de arquitectura
- Codificación y guía arquitectónica
- Herramientas y ambientes de diseño arquitectónico
- Estudios de casos

Fundamental en la concepción de Clements y Northrop [CN96] es el criterio de reusabilidad como uno de los aspectos que más hacen por justificar la disciplina misma. Según estos autores, el estudio actual de la AS puede ser visto como un esfuerzo ex post facto para proporcionar un almacén estructurado de este tipo de información reutilizable de diseño de alto nivel propio de una familia (en el sentido de Parnas). De tal manera, las decisiones de alto nivel inherentes a cada miembro de una familia de programas no necesitan ser re-inventadas, re-validadas y re-descriptas.

Un razonamiento arquitectónico es además un argumento sobre las cuestiones estructurales de un sistema. Se diría también que el concepto de estilo es la encarnación principal del principio de reusabilidad en el plano arquitectónico.

Paul Clements define cinco temas fundamentales en torno de los cuales se agrupa la disciplina:

- Diseño o selección de la arquitectura: Cómo crear o seleccionar una arquitectura en base de requerimientos funcionales, de rendimiento o de calidad.
- Representación de la arquitectura: Cómo comunicar una arquitectura. Este problema se ha manifestado como el problema de la representación de arquitecturas utilizando recursos lingüísticos, pero el problema también incluye la selección del conjunto de información a ser comunicada.
- Evaluación y análisis de la arquitectura: Cómo analizar una arquitectura para predecir cualidades del sistema en que se manifiesta. Un problema semejante es cómo comparar y escoger entre diversas arquitecturas en competencia.
- Desarrollo y evolución basados en arquitectura: Cómo construir y mantener un sistema dada una representación de la cual se cree que es la arquitectura que resolverá el problema correspondiente.

- Recuperación de la arquitectura: Cómo hacer que un sistema legacy evolucione cuando los cambios afectan su estructura; para los sistemas de los que se carezca de documentación confiable, esto involucra primero una "arqueología arquitectónica" que extraiga su arquitectura.

Mary Shaw considera que en el 2001 los campos más promisorios de la AS siguen teniendo que ver con el tratamiento sistemático de los estilos, el desarrollo de lenguajes de descripción arquitectónica, la formulación de metodologías y (ahora) el trabajo con patrones de diseño. Se requieren todavía modelos precisos que permitan razonar sobre las propiedades de una arquitectura y verificar su consistencia y completitud, así como la automatización del proceso de análisis, diseño y síntesis. Sugiere que debe aprenderse una lección a partir de la experiencia de la ingeniería de software, la cual no obstante haberse desenvuelto durante treinta años no ha logrado plasmar un conjunto de paradigmas de investigación comparable al de otras áreas de las ciencias de la computación. Estima que la AS se encuentra ya en su fase de desarrollo y extensión, pero que tanto las ideas como las herramientas distan de estar maduras.

Un campo que no figura en estas listas pero sobre el cual se está trabajando intensamente es en el de la coordinación de los ADLs que sobrevivan con UML 2.0 por un lado y con XML por el otro.

Ningún lenguaje de descripción arquitectónica en el futuro próximo (excepto los que tengan un nicho técnico muy particular) será viable si no satisface esos dos requisitos.
Los ejercicios que pueden hacerse para precisar los campos de la AS son incontables. Ahora que la AS se ha abismado en el desarrollo de metodologías, hace falta, por ejemplo, establecer con más claridad en qué difieren sus elaboraciones en torno del diseño, del análisis de requerimientos o de justificación económica de las llevadas a cabo por la ingeniería de software.

Asimismo, se está esperando todavía una lista sistemática y exhaustiva que describa los dominios de incumbencia de la disciplina, así como un examen del riesgo de duplicación de esfuerzos entre campos disciplinarios mal comunicados, una situación que a primera vista parecería contradictoria con el principio de reusabilidad.

1.6.4 Modalidades y tendencias

En la década de 1990 se establece definitivamente la AS como un dominio todavía hoy separado de manera confusa de ese marco global que es la ingeniería y de esa práctica puntual que es el diseño. Aunque no hay un discurso explícito y autoconsciente sobre escuelas de AS, ni se ha publicado un estudio reconocido y sistemático que analice las particularidades de cada una, en la actualidad se pueden distinguir a grandes rasgos unas seis corrientes. Algunas distinciones están implícitas por ejemplo en, pero la bibliografía sobre corrientes y alternativas prácticamente no existe y la que sigue habrá de ser por un tiempo una de las pocas propuestas contemporáneas sobre el particular.

Ahora bien, articular una taxonomía de estrategias no admite una solución simple y determinista. En distintos momentos de su trayectoria, algunos practicantes de la AS se mueven ocasionalmente de una táctica a otra, o evolucionan de un punto de vista más genérico a otro más particular, o realizan diferentes trabajos operando en marcos distintos. Además, con la excepción del "gran debate metodológico" entre métodos pesados y ligeros, las discusiones entre las distintas posturas rara vez se han manifestado como choques frontales entre ideologías irreconciliables, por lo que a menudo hay que leer entre líneas para darse cuenta que una afirmación cualquiera es una crítica a otra manera de ver las cosas, o trasunta una toma definida de posición. Fuera de la metodología, el único factor reconocible de discordia ha sido, hasta la fecha, la preeminencia de UML y el diseño orientado a objetos. Todo lo demás parece ser más o menos negociable. La división preliminar de escuelas de AS [5] que se proponen es la siguiente:

1. Arquitectura como etapa de ingeniería y diseño orientada a objetos. Es el modelo de James Rumbaugh, Ivar Jacobson, Grady Booch, Craig Larman y otros, ligado estrechamente al mundo de UML y Rational. No cabe duda que se trata de una corriente específica; Rumbaugh, Jacobson y Booch han sido llamados "Los Tres Amigos"; de lo que sí puede dudarse es que se trate de una postura arquitectónica. En esta postura, la arquitectura se restringe a las fases iniciales y preliminares del proceso y concierne a los niveles más elevados de abstracción, pero no está sistemáticamente ligada al requerimiento que viene antes o a la composición del diseño que viene después. Lo que sigue al momento arquitectónico es business as usual, y a cualquier configuración, topología o morfología de las piezas del sistema se la llama arquitectura. En esta escuela, si bien se reconoce el valor primordial de la abstracción (nadie después de Dijkstra osaría oponerse a ello) y del ocultamiento de información promovido por Parnas, estos conceptos tienen que ver más con el encapsulamiento en clases y objetos que con la visión de conjunto arquitectónica. Para este movimiento, la arquitectura se confunde también con el modelado y el diseño, los cuales constituyen los conceptos dominantes. En esta corriente se manifiesta predilección por un modelado denso y una profusión de diagramas, tendiente al modelo metodológico CMM o a UPM; no hay, empero, una prescripción formal. Importa más la abundancia y el detalle de diagramas y técnicas disponibles que la simplicidad de la visión de conjunto. Cuando aquí se habla de estilos, se los confunde con patrones arquitectónicos o de diseño. Jamás se hace referencia a los lenguajes de descripción arquitectónica, que representan uno de los assets reconocidos de la AS; sucede como si la disponibilidad

de un lenguaje unificado de modelado los tornara superfluos. La definición de arquitectura que se promueve en esta corriente tiene que ver con aspectos formales a la hora del desarrollo; esta arquitectura es isomorfa a la estructura de las piezas de código. Una definición típica y demostrativa sería la de Grady Booch; para él, la AS es "la estructura lógica y física de un sistema, forjada por todas las decisiones estratégicas y tácticas que se aplican durante el desarrollo". Otras definiciones revelan que la AS, en esta perspectiva, concierne a decisiones sobre organización, selección de elementos estructurales, comportamiento, composición y estilo arquitectónico susceptibles de ser descriptas a través de las cinco vistas clásicas del modelo 4+1 de Kruchten.

2. Arquitectura estructural, basada en un modelo estático de estilos, ADLs y vistas. Constituye la corriente fundacional y clásica de la disciplina. Los representantes de esta corriente son todos académicos, mayormente de la Universidad Carnegie Mellon en Pittsburgh: Mary Shaw, Paul Clements, David Garlan, Robert Allen, Gregory Abowd, John Ockerbloom. Se trata también de la visión de la AS dominante en la academia, y aunque es la que ha hecho el esfuerzo más importante por el reconocimiento de la AS como disciplina, sus categorías y herramientas son todavía mal conocidas en la práctica industrial. En el interior del movimiento se pueden observar distintas divisiones. En principio se pueden reconocer tres modalidades en cuanto a la formalización; los más informales utilizan descripciones verbales o diagramas de cajas, los de talante intermedio se sirven de ADLs y los más exigentes usan lenguajes formales de especificación como CHAM y Z. En toda la corriente, el diseño arquitectónico no sólo es el de más alto nivel de abstracción, sino que además no tiene por qué coincidir con la configuración explícita de las aplicaciones; rara vez se encontrarán referencias a lenguajes de programación o piezas de código, y en general nadie habla de clases o de objetos. Mientras algunos participantes excluyen el modelo de datos de las incumbencias de la AS (Shaw, Garlan, etc.), otros insisten en su relevancia (Medvidovic, Taylor). Todo estructuralismo es estático; hasta fines del siglo XX, no está muy claro el tema de la posición del modelado arquitectónico en el ciclo de vida.

3. Estructuralismo arquitectónico radical. Se trata de un desprendimiento de la corriente anterior, mayoritariamente europeo, que asume una actitud más confrontativa con el mundo UML. En el seno de este movimiento hay al menos dos tendencias, una que excluye de plano la relevancia del modelado orientado a objetos para la AS y otra que procura definir nuevos meta-modelos y estereotipos de UML como correctivos de la situación. Dado que pueden existir dudas sobre la materialidad de esta corriente como tal, proporcionamos referencias bibliográficas masivas que corroboran su existencia. Pasaremos revista a los argumentos más fuertes en contra de UML emanados de este movimiento en el capítulo correspondiente del documento sobre lenguajes de descripción arquitectónica.

4. Arquitectura basada en patrones. Si bien reconoce la importancia de un modelo emanado históricamente del diseño orientado a objetos, esta corriente surgida hacia 1996 no se encuentra tan rígidamente vinculada a UML en el modelado, ni a CMM en la metodología. El texto sobre patrones que esta variante reconoce como referencia es la serie POSA de Buschmann y otros y secundariamente el texto de la Banda de los

Cuatro. La diferencia entre ambos textos sagrados de la comunidad de patrones no es menor; en el primero, la expresión "Software Architecture" figura en el mismo título; el segundo se llama Design Patterns: Elements of reusable Object-Oriented software y su tratamiento de la arquitectura es mínimo. En esta manifestación de la AS prevalece cierta tolerancia hacia modelos de procesos tácticos, no tan macroscópicos, y eventualmente se expresa cierta simpatía por las ideas de Martin Fowler y las premisas de la programación extrema. El diseño consiste en identificar y articular patrones preexistentes, que se definen en forma parecida a los estilos de arquitectura.

5. Arquitectura procesual. Desde comienzos del siglo XXI, con centro en el SEI y con participación de algunos (no todos) los arquitectos de Carnegie Mellon de la primera generación y muchos nombres nuevos de la segunda: Rick Kazman, Len Bass, Paul Clements, Felix Bachmann, Fabio Peruzzi, Jeromy Carrière, Mario Barbacci, Charles Weinstock. Intenta establecer modelos de ciclo de vida y técnicas de elicitación de requerimientos, brainstorming, diseño, análisis, selección de alternativas, validación, comparación, estimación de calidad y justificación económica específicas para la arquitectura de software. Toda la documentación puede encontrarse ordenada en el SEI, pero no se mezcla jamás con la de CMM, a la que redefine de punta a punta. Otras variantes dentro de la corriente procesual caracterizan de otras maneras de etapas del proceso: extracción de arquitectura, generalización, reutilización.

6. Arquitectura basada en escenarios. Es la corriente más nueva. Se trata de un movimiento predominantemente europeo, con centro en Holanda. Recupera el nexo de la AS con los requerimientos y la funcionalidad del sistema, ocasionalmente borroso en la arquitectura estructural clásica. Los teóricos y practicantes de esta modalidad de arquitectura se inscriben dentro del canon delineado por la arquitectura procesual, respecto de la cual el movimiento constituye una especialización. En esta corriente suele utilizarse diagramas de casos de uso UML como herramienta informal u ocasional, dado que los casos de uso son uno de los escenarios posibles. Los casos de uso no están orientados a objeto. Los autores vinculados con esta modalidad han sido, aparte de los codificadores de ATAM, CBAM, QASAR y demás métodos del SEI, los arquitectos holandeses de la Universidad Técnica de Eindhoven, de la Universidad Brije, de la Universidad de Groningen y de Philips Research: Mugurel Ionita, Dieter Hammer, Henk Obbink, Hans de Bruin, Hans van Vliet, Eelke Folmer, Jilles van Gurp, Jan Bosch. La profusión de holandeses es significativa; la Universidad de Eindhoven es, incidentalmente, el lugar en el que surgió lo que P. I. Sharp proponía llamar la escuela arquitectónica de Dijkstra.

En todos los simposios han habido intentos de fundación de otras variedades de AS, tales como una arquitectura adaptativa inspirada en ideas de la programación genética o en teorías de la complejidad, la auto-organización, el caos y los fractales, una arquitectura centrada en la acción que recurre a la inteligencia artificial heideggeriana o al postmodernismo, y una arquitectura epistemológicamente reflexiva que tiene a Popper o a Kuhn entre sus referentes; consideramos preferible omitirlas, porque por el momento ni su masa crítica es notoria ni su mensaje parece sustancial. Pero hay al menos un movimiento cismático digno de tomarse más en serio; a esta altura de los acontecimientos es muy posible que pueda hablarse también de una anti-arquitectura, que en nombre de los métodos

heterodoxos se opone tanto al modelado orientado a objetos y los métodos sobre documentados impulsados por consultoras corporativas como a la abstracción arquitectónica que viene de la academia. El Manifiesto por la Programación Ágil, en efecto, valoriza:

- Los individuos y las interacciones por encima de los procesos y las herramientas
- El software que funciona por encima de la documentación exhaustiva
- La colaboración con el cliente por encima de la negociación contractual
- La respuesta al cambio por encima del seguimiento de un plan

Habrá oportunidad de desarrollar el tratamiento de estas metodologías en otros documentos de la serie. Hay muchas formas, por cierto, de organizar el panorama de las tendencias y las escuelas. Discernir la naturaleza de cada una puede tener efectos prácticos a la hora de comprender antagonismos, concordancias, sesgos, disyuntivas, incompatibilidades, fuentes de recursos. Nadie ha analizado mejor los límites de una herramienta como UML o de los métodos ultra-formales, por ejemplo, que los que se oponen a su uso por razones teóricas precisas, por más que éstas sean interesadas. Un buen ejercicio sería aplicar vistas y perspectivas diferentes a las que han regido esta clasificación informal, y proponer en función de otros criterios o de métodos más precisos de composición y referencias cruzadas, taxonomías

1.6.5 Arquitectura EJB 3 v/s Arquitectura Spring Framework

1.6.5.1 EJB 3

Una de las metas de la arquitectura EJB es la de poder escribir de manera fácil aplicaciones de negocio orientadas a objetos y distribuidas, basadas en el lenguaje de programación JAVA. Desafortunadamente, las versiones 1.0 a 2.1 de EJB fueron demasiado complejas y no alcanzaron esta meta. El propósito de EJB 3 es el de proveer el soporte de la arquitectura de EJB y al mismo tiempo reducir la complejidad para el desarrollo de aplicaciones empresariales. Para simplificar la arquitectura EJB se realizaron los siguientes cambios:

- Se introduce las anotaciones de metadatos (metadata annotations) [1] las mismas que pueden ser usadas en combinación con el descriptor de despliegue (deployment descriptor) ó separadas del mismo, para anotar aplicaciones EJB (especificar tipos de componentes, comportamiento, etc.), como una manera de encapsular dependencias del ambiente de trabajo y recursos.
- Se elimina el requerimiento de especificar una interfaz "home"
- En los entreprise beans se elimina la necesidad de implementar una interfaz específica (javax.ejb.EnterpriseBean)
- Se simplifican los tipos de entreprise beans (Los entity beans fueron removidos)
- La existencia de interceptores reemplaza la necesidad de implementar interfaces tipo callback1.
- Los valores por defecto se emplean lo menos posible (se usa la aproximación de configuración por excepción).
- Se reducen los requerimientos para el manejo de excepciones

Como contrapunto se introducen en EJB 3 las anotaciones de metadatos y de interceptores como las siguientes:

- La persistencia de entidades (Entity Persistence) fue simplificada y soportada para modelar dominios de negocio medianos a grandes, además ahora es posible proveer contenedores EJB 3 livianos que pueden ser usados en una capa cliente fuera de la caja del servidor de aplicaciones.
- Se mejora en EJB QL el soporte para consultas y sentencias SQL nativas
- Se provee de un servicio de temporizador (Timer Service) manejado por el contenedor EJB el mismo que permite ejecutar Enterprise Beans en eventos de tiempo específicos.
- En EJB3 se puede usar AOP a través de interceptores. [9]

En el capítulo 3 se entrará en detalle de esta tecnología.

1.6.5.2 Spring Framework

El principal objetivo de Spring Framework es el constituirse en una alternativa sencilla y fácil ante EJB. La simplificación del desarrollo de aplicaciones y de sus respectivas pruebas (testing) es una de las claves del éxito de Spring. Este Framework se sustenta en dos características básicas en su núcleo: Inversión de Control (Inversion of Control IoC) y la Programación Orientada a Aspectos (Aspect-orient programming).

Usualmente, los objetos obtienen las referencias de otros objetos requeridos por si mismos (tal como en EJB 2.0 los beans obtienen los recursos necesarios usando JNDI). La inversión de control permite inyectar las dependencias en un bean al momento de su creación usando un manejador externo. El bean sólo necesita definir la propiedad requerida en su código así como el método de establecimiento (set() method). La fuente primaria de la inyección de dependencias es un archivo de configuración en formato XML. Por ejemplo productService necesita realizar alguna operación sobre customerService, la referencia de customerService debe ser inyectada en la propiedad customer de com.article.ProductService en la siguiente figura se muestra un ejemplo de cómo se mapearía esta configuración en el respectivo archivo XML.

```
<beans>
<bean id=”customerService” class=”com.article.CustomerServiceImpl”/>
<bean id=”productService” class=”com.article.ProductServiceImpl” >
<property name=”customer” ref=”customerService”/>
</bean>
</beans>
```

La Programación Orientada a Aspectos (Aspect-Orient Programming AOP) permite implementar la mayoría de los servicios comunes (como manejo de transacciones, seguridad, logging, etc.) que pueden ser aplicados en múltiples componentes. En el caso del uso de AOP no se requiere ningún conocimiento acerca de cómo han sido enmascarados (wrapped2) los servicios. AOP es usada en Spring [8] para:

- Proveer servicios de aplicación (enterprise services) declarativos. Ejemplo declarar el manejo de transacciones

- Permitir a los usuarios la facilidad de implementar sus propios aspectos personalizados

Spring provee un número de servicios adicionales que son basados en IoC y AOP. Estos servicios deben ser comparados con sus equivalentes en EJB para poder tener un buen criterio de evaluación.

1.6.6 Métricas de comparación

El propósito de la comparación es mostrar diferencias entre EJB y Spring. Para llegar a esta meta se han seleccionado los siguientes criterios

1. Manejo de transacciones (Transaction Manager), la comparación debe permitir comparar las diferentes clases de implementación de transacciones que están soportadas
2. Oportunidades de criterios de transacción incluidos (Transaction Opportunities), atributos soportados, niveles de isolation3), soporte de transacciones anidadas.
3. Manejo de entidades de persistencia (Entity Persistente) que permitan evaluar funcionalidad para persistencia de objetos Object-Relational Mappings (ORM)
4. AOP (Interceptors) muestran como se provee funcionalidad para programación orientada a aspectos
5. Configuración de aplicaciones (Application Configuration), la posibilidad de instalar la configuración de la aplicación y servicios declarativos
6. Seguridad (Security) la comparación muestra como se ofrecen diferentes niveles de seguridad.
7. Flexibilidad de servicios (Service Flexibility), evaluar la posibilidad de reemplazar servicios por otros.
8. Servicios de integración (Service Integration), detecta las facilidades de integración en especial con los servidores de aplicaciones.
9. Funcionalidad adicional (Additional Functionality), describe funcionalidad adicional provista por el framework,

Cada criterio será evaluado de la siguiente forma:

Tabla 2: Métricas para Comparar

Métricas (N°)	No existe 0	Insuficiente 1	Aceptable 2	Excelente 3
1	No existe el manejo de transacciones en esta tecnología.	Las diferentes clases de implementación de transacciones son engorrosas.	El manejo de transacciones es comprensible, en general, además de la poca complejidad de la implementación de estas.	Las diferentes clases de implementación de transacciones son muy fáciles de llevar a cabo, puesto que son sencillas.
2	No tiene soporte de transacciones anidadas.	La implementación de transacciones anidadas, es confusa.	La implementación de transacciones anidadas es adecuada, a pesar de las restricciones que la aplicación impone.	La implementación de transacciones anidadas, es fácil de llevar a cabo.
3	No existe manejo	Poco entendible, el	El manejo de entidades	El manejo de entidades

	de entidades de persistencia.	manejo de entidades de persistencia.	de persistencia es adecuado a pesar de la complejidad de llevarla a cabo.	de persistencia es excelente; fácil de implementar y con grandes funcionalidades para la persistencia de objetos.
4	No provee funcionalidad para programación orientada a aspectos.	Las funcionalidades para la programación orientada a aspectos, son complejos y confusos (difícil de implementar).	Las funcionalidades para la programación orientada a aspectos, son adecuadas a pesar del nivel de complejidad de implementarlas.	Las funcionalidades para la programación orientada a aspectos, son claras y fácil de implementar.
5	No se puede instalar configuraciones a la aplicación.	Se pueden instalar configuraciones a la aplicación, pero son demasiadas confusas.	Se pueden instalar configuraciones a las aplicaciones de manera clara.	La posibilidad de instalar la configuración de la aplicación y servicios declarativos, es de manera fácil.
6	No provee niveles de seguridad.	Los niveles de seguridad provistos, son poco claros y son difíciles de llevarlos a cabo.	Los niveles de seguridad ofrecidos son adecuados en general y con cierta comodidad para el programador a la hora de implementar el servicio.	Se pueden encontrar diferentes niveles de seguridad, los cuales son fácil de llevar a cabo.
7	No se pueden reemplazar servicios por otros.	Se pueden reemplazar servicios por otros, pero el costo es caro, por su gran complejidad, y su implementación poco clara.	La flexibilidad de los servicios es clara.	La flexibilidad de servicios es grande y se pueden reemplazar servicios por otros de manera fácil.
8	No existen servicios de integración.	La integración es bastante compleja y confusa.	La integración en especial con los servidores de aplicaciones es adecuada a pesar de algunos errores e incompatibilidades.	Tiene una gran facilidad de integración en especial con los servidores de aplicaciones, por lo que hace que la integración sea fácil.
9	No se pueden visualizar ni instalar funcionalidades adicionales.	Se pueden instalar funcionalidades adicionales, pero es demasiado complejo y confuso hacerlo.	Se pueden instalar funcionalidades adicionales de forma adecuada a pesar de los errores.	La instalación de aplicaciones adicionales, es muy fácil e intuitivo de llevar a cabo.

1.7 Caso de estudio

Como caso de estudio, se tomó un sistema web, basado en jsp, el cual pueda administrar los terminales portuarios de nuestro país.

Dentro de las funcionalidades principales de este sistema, tenemos:

- Administrar terminales: debe permitir administrar la información de los terminales, de la siguiente forma:
    - Ingresar terminales con sus datos.
    - Eliminar terminales.
    - Modificar terminales.
    - Filtrar información de las terminales.
- Generar Reportes: estos reportes se generan con el filtrado de datos. Esto debe permitir:
    - Filtrar por diferentes columnas.
    - Exportar resultados a Planilla Excel.
- Cargar archivos: se pueden cargar archivos en el sistema, con los datos de los terminales.

A continuación de muestra el caso de uso de alto nivel del Sistema de Administración de Terminales Portuarios.

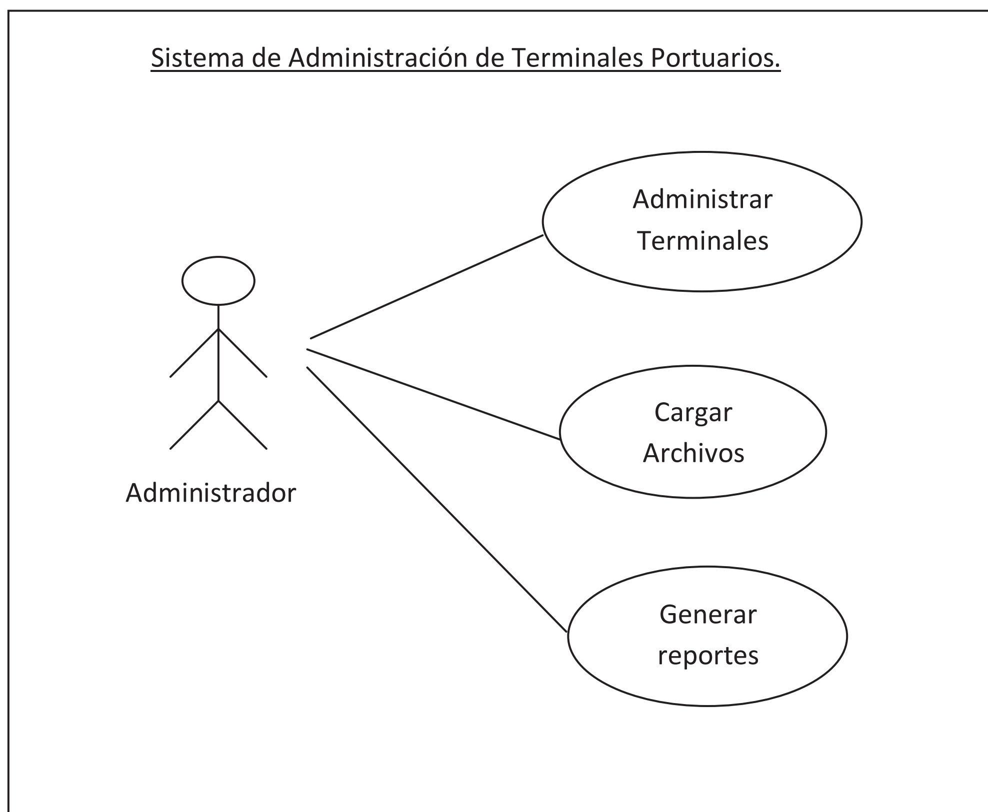

Figura 3: Diagrama general de caso de uso, para el sistema propuesto

2 Spring, un Framework de aplicación

En este capítulo se dan a conocer todos los aspectos técnicos de Spring, los conceptos básicos, como sus creadores, sus características esenciales, las partes de su arquitectura, así como los componentes que lo conforman. En algunas partes se dan pedazos de código con el fin de ejemplificar mejor cierta situación. Este capítulo es un análisis un poco más profundo de las diferentes partes que conforman la arquitectura de Spring.

2.1 Introducción e historia

Spring es un framework de aplicación desarrollado por la compañía Interface 21, para aplicaciones escritas en el lenguaje de programación Java. Fue creado gracias a la colaboración de grandes programadores, entre ellos se encuentran como principales partícipes y líderes de este proyecto Rod Johnson y Jürgen Höller. Estos dos desarrolladores, además de otros colaboradores que juntando toda su experiencia en el desarrollo de aplicaciones J2EE (Java 2 Enterprise Editions), incluyendo EJB (Enterprise JavaBeans), Servlets y JSP (Java Server Pages), lograron combinar dichas herramientas y otras más en un sólo paquete, para brindar una estructura más sólida y un mejor soporte para este tipo de aplicaciones.

Además se considera a Spring un framework lightweight, es decir liviano o ligero, ya que no es una aplicación que requiera de muchos recursos para su ejecución, además el framework completo puede ser distribuido en un archivo .jar de alrededor de 1 MB, lo cual representa muy poco espacio, y para la cantidad de servicios que ofrece es relativamente insignificante su tamaño.

Este framework se encuentra actualmente en su versión 1.2.5, aunque es una versión temprana, está adquiriendo gran auge y una gran popularidad. Una de las características que ayuda a este éxito, es que es una aplicación open source, lo cual implica que no tiene ningún costo, ni se necesita una licencia para utilizarlo, por lo tanto da la libertad a muchas empresas y desarrolladores a incursionar en la utilización de esta aplicación. Además de que está disponible todo el código fuente de este framework en el paquete de instalación.

Spring no intenta "reinventar la rueda" sino integrar las diferentes tecnologías existentes, en un sólo framework para el desarrollo más sencillo y eficaz de aplicaciones J2EE portables entre servidores de aplicación. [3]

Otro de los principales enfoques de Spring y por el cual está ganando dicha popularidad es que simplifica el desarrollo de aplicaciones J2EE, al intentar evitar el uso de EJB, ya que como menciona Craig Walls en su libro Spring in Action, "En su estado actual, EJB es complicado. Es complicado porque EJB fue creado para resolver cosas complicadas, como objetos distribuidos y transacciones remotas.". Y muchas veces aunque el proyecto no es lo suficientemente complejo, se utiliza EJB, contenedores de alto peso y otras herramientas que soportan un grado mayor de complejidad, como una solución a un proyecto. "Con Spring, la complejidad de tu aplicación es proporcional a la complejidad del problema que se está resolviendo.". Esto sin embargo no le quita crédito a EJB, ya que también ofrece a los desarrolladores servicios valiosos y útiles para resolver ciertas tareas, la diferencia radica en que Spring intenta brindar los mismos servicios pero simplificando el modelo de programación.

Spring fue creado basado en los siguientes principios:

- El buen diseño es más importante que la tecnología subyacente.
- Los JavaBeans ligados de una manera más libre entre interfaces es un buen modelo.
- El código debe ser fácil de probar.

2.2 Arquitectura de Spring

Spring es un framework modular que cuenta con una arquitectura divida en siete capas o módulos, como se muestra en la Figura 4, lo cual permite tomar y ocupar únicamente las partes que interesen para el proyecto y juntarlas con gran libertad. [4]

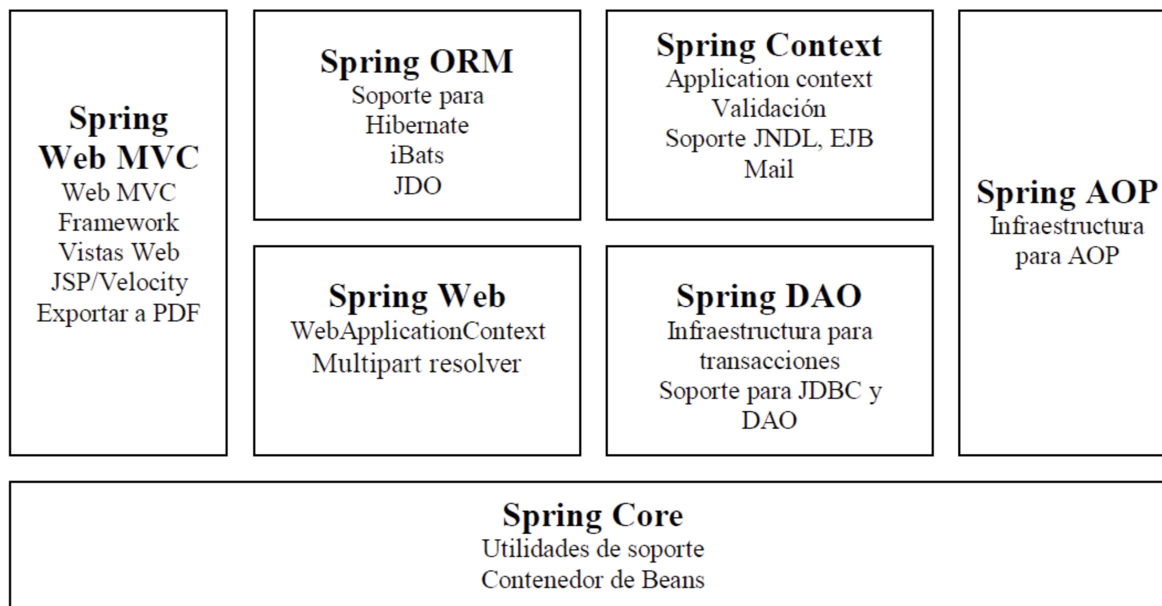

Figura 4: Arquitectura de Spring

2.2.1 Spring Core

Esta parte es la que provee la funcionalidad esencial del framework, está compuesta por el BeanFactory, el cual utiliza el patrón de Inversión de Control (Inversion of Control) y configura los objetos a través de Inyección de Dependencia (Dependency Injection). El núcleo de Spring es el paquete org.springframework.beans el cual está diseñado para trabajar con JavaBeans.

2.2.1.1 Bean Factory

Es uno de los componentes principales del núcleo de Spring. Es una implementación del patrón Factory, pero a diferencia de las demás implementaciones de este patrón, que muchas veces sólo producen un tipo de objeto, BeanFactory es de propósito general, ya que puede crear muchos tipos diferentes de Beans. Los Beans pueden ser llamados por nombre y se encargan de manejar las relaciones entre objetos.

Todas las Bean Factories implementan la interfase org.springframework.beans.factory.BeanFactory, con instancias que pueden ser accedidas a través de esta interfaz. Además también soportan objetos de dos modos diferentes:

- Singleton: Existe únicamente una instancia compartida de un objeto con un nombre particular, que puede ser regresado o llamado cada vez que se necesite. Este es el método más común y el más usado. Este modo está basado en el patrón de diseño que lleva el mismo nombre.
- Prototype: también conocido como non-singleton, en este método cada vez que se realiza un regreso o una llamada, se crea un nuevo objeto independiente.

La implementación de BeanFactory más usada es org.springframework.beans.factory.xml.XmlBeanFactory que se encarga de cargar la definición de cada Bean que se encuentra guardada en un archivo XML y que consta de:
id (que será el nombre que el que se le conocerá en las clases), clase (tipo de Bean), Singleton o Prototype (modos del Bean antes mencionados), propiedades, con sus atributos name, value y ref, además argumentos del constructor, método de inicialización y método de destrucción. A continuación se muestra un ejemplo de un Bean:

```
<beans>
	<bean id=”exampleBean” class=”eg.ExampleBean” singleton=”true”/>
	<property name="driverClassName" value="com.mysql.jdbc.Driver" />
</beans>
```

Como se muestra en el ejemplo la base de este documento XML es el elemento <beans>, y adentro puede contener uno o más elementos de tipo <bean>, para cada una de las diferentes funciones que se requieran.

Para cargar dicho XML se le manda un InputStream al constructor de XmlBeanFactory de la siguiente manera:

```
BeanFactory fact = new XmlBeanFactory(new FileInputStream(“bean.xml”));
```

Esto sin embargo no quiere decir que sea instanciado directamente, una vez que la definición es cargada, únicamente cuando se necesite el Bean se creará una instancia dependiendo de sus propiedades. Para tomar un Bean de un Factory simplemente se usa el método getBean(), mandándole el nombre del Bean que se desea obtener :

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

2.2.1.2 Inversion of Control

“Inversion of Control se encuentra en el corazón de Spring”. BeanFactory utiliza el patrón de Inversión de Control o como se le conoce IoC, que es una de las funcionalidades más importantes de Spring. Esta parte se encarga de separar del código de la aplicación que se está desarrollando, los aspectos de configuración y las especificaciones de dependencia del framework. Todo esto configurando los objetos a través de Inyección de Dependencia o Dependency Injection, que se explicará más adelante. [10]

Una forma sencilla de explicar el concepto de IoC es el “principio Hollywood”: “No me llames, yo te llamaré a ti”. Traduciendo este principio a términos de este trabajo, en lugar de que el código de la aplicación llame a una clase de una librería, un framework que utiliza IoC llama al código. Es por esto que se le llama “Inversión”, ya que invierte la acción de llamada a alguna librería externa.

2.2.1.3 Dependency Injection

Es una forma de Inversión de Control, que está basada en constructores de Java, en vez de usar interfaces específicas del framework. Con este principio en lugar de que el código de la aplicación utilicé el API del framework para resolver las dependencias como: parámetros de configuración y objetos colaborativos, las clases de la aplicación exponen o muestran sus dependencias a través de métodos o constructores que el framework puede llamar con el valor apropiado en tiempo de ejecución, basado en la configuración.

Todo esto se puede ver de una forma de push y pop, el contenedor hace un push de las dependencias para ponerlas dentro de los objetos de la aplicación, esto ocurre en tiempo de ejecución. La forma contraria es tipo pull, en donde los objetos de la aplicación jalan las dependencias del ambiente. Además los objetos de la Inyección de Dependencia nunca cargan las propiedades ni la configuración, el framework es totalmente responsable de leer la configuración.

Spring soporta varios tipos de Inyección de Dependencia, pero en si estos son los dos más utilizados:

- Setter Injection: en este tipo la Inyección de Dependencia es aplicada por medio de métodos JavaBeans setters, que a la vez tiene un getter respectivo.
- Constructor Injection: esta Inyección es a través de los argumentos del constructor.

A continuación se muestra un ejemplo tomado del libro de Java Development with Spring framework del autor Rob Johnson, en el cual se muestra como un objeto es configurado a través de Inyección de Dependencia. Se tiene una interfaz Service y su implementación ServiceImpl. Supóngase que la ServiceImpl tiene dos dependencias: un int que tiene configura un timeout y un DAO (Data Access Object). Con el método SetterInjection se puede configurar ServiceImpl utilizando las propiedades de un JavaBean para satisfacer estas 2 dependencias. [11]

```
public class ServiceImpl implements Service {
	private int timeout;
	private AccountDao accountDao;
public void setTimeout(int timeout) {
this.timeout = timeout;
}
public void setAccountDao(AccountDao accountDao) {
	this.accountDao = accountDao;
}
```

Con Constructor Injection se le da las dos propiedades al constructor:

```
public class ServiceImpl implements Service {
       private int timeout;
       private AccountDao accountDao;
public ServiceImpl(int timeout, AccountDao accountDao) {
       this.timeout = timeout;
       this.accountDao = accountDao;
}
```

La clave de la innovación de la Inyección de Dependencia es que trabaja con sintaxis pura de Java, no es necesaria la dependencia del API del contenedor.

2.2.2 Spring Context

El módulo BeanFactory del núcleo de Spring es lo que lo hace un contenedor, y el módulo de contexto es lo que hace un framework.

En sí Spring Context es un archivo de configuración que provee de información contextual al framework general. Además provee servicios enterprise como JNDI, EJB, email, validación y funcionalidad de agenda.

2.2.2.1 Application Context

ApplicationContext es una sub-interfaz de BeanFactory, ya que org.springframework.context.ApplicationContext es una subclase de BeanFactory.

En si todo lo que puede realizar una BeanFactory también lo puede realizar ApplicationContext. En sí agrega información de la aplicación que puede ser utilizada por todos los componentes. Además brinda las siguientes funcionalidades extra:

- Localización y reconocimiento automático de las definiciones de los Beans.
- Cargar múltiples contextos.
- Contextos de herencia.
- Búsqueda de mensajes para encontrar su origen.
- Acceso a recursos.
- Propagación de eventos, para permitir que los objetos de la aplicación puedan publicar y opcionalmente registrarse para ser notificados de los eventos.
- Agrega soporte para internacionalización (i18n).

En algunos casos es mejor utilizar ApplicationContext ya que obtienes más funciones a un costo muy bajo, en cuanto a recursos se refiere. Ejemplo de un ApplicationContext:

```
ApplicationContext ct = new FileSystemXmlApplicationContext (“c:\bean.xml”);
ExampleBean eb = (ExampleBean) ct.getBean(“exampleBean”);
```

2.2.3 Spring AOP

Aspect-oriented programming, o AOP, es una técnica que permite a los programadores modularizar ya sea las preocupaciones crosscutting, o el comportamiento que corta a través de las divisiones de responsabilidad, como logging, y manejo de transacciones. El núcleo de construcción es el aspect, que encapsula comportamiento que afectan a diferentes clases, en módulos que pueden ser reutilizados. AOP se puede utilizar para:

- Persistencia
- Manejo de transacciones
- Seguridad
- Logging
- Debugging

AOP es un enfoque diferente y un poco más complicado de acostumbrarse en comparación con OOP (Object Oriented Programming). Rob Johnson prefiere referirse a AOP como un complemento en lugar de como un rival o un conflicto.

Spring AOP es portable entre servidores de aplicación, funciona tanto en servidores Web como en contenedores EJB. Spring AOP soporta las siguientes funcionalidades:

- Intercepción: se puede insertar comportamiento personalizado antes o después de invocar a un método en cualquier clase o interfaz.
- Introducción: Especificando que un advice (acción tomada en un punto particular durante la ejecución de un programa) debe causar que un objeto implemente interfaces adicionales.
- Pointcuts dinámicos y estáticos: para especificar los puntos en la ejecución del programa donde debe de haber intercepción.

Spring implementa AOP utilizando proxies dinámicos. Además se integra transparentemente con los BeanFactory que existen. En el ejemplo siguiente se muestra como definir un proxy AOP:

```
<bean id="myTest"class="org.springframework.aop.Framework.ProxyFactoryBean">
	<property name="proxyInterfaces">
		<value>org.springframework.beans.ITestBean</value>
	</property>
	<property name="interceptorNames">
		<list>
			<value>txInterceptor</value>
			<value>target</value>
		</list>
	</property>
</bean>
```

2.2.4 Spring ORM

En lugar de que Spring proponga su propio módulo ORM (Object-Relational Mapping), para los usuarios que no se sientan confiados en utilizar simplemente JDBC, propone un módulo que soporta los frameworks ORM más populares del mercado, entre ellos:

- Hibernate (2.1 y 3.0): es una herramienta de mapeo O/R open source muy popular, que utiliza su propio lenguaje de query llamada HQL.
- iBATIS SQL Maps (1.3 y 2.0). una solución sencilla pero poderosa para hacer externas las declaraciones de SQL en archivos XML.
- Apache OJB (1.0): plataforma de mapeo O/R con múltiples APIs para clientes.
- Entre otros como JDO (1.0 y 2.0) y Oracle TopLink.

Todo esto se puede utilizar en conjunto con las transacciones estándar del framework. Spring e Hibernate es una combinación muy popular. Algunas de las ventajas que brinda Spring al combinarse con alguna herramienta ORM son:

- Manejo de sesión: Spring hace de una forma más eficiente, sencilla y segura la forma en que se manejan las sesiones de cualquier herramienta ORM que se quiera utilizar.
- Manejo de recursos: se puede manejar la localización y configuración de los SessionFactories de Hibernate o las fuentes de datos de JDBC por ejemplo. Haciendo que estos valores sean más fáciles de modificar.
- Manejo de transacciones integrado: se puede utilizar una plantilla de Spring para las diferentes transacciones ORM.
- Envolver excepciones: con esta opción se pueden envolver todas las excepciones para evitar las molestas declaraciones y los catch en cada segmento de código necesarios.
- Evita limitarse a un solo producto: Si se desea migrar o actualizar a otra versión de un ORM distinto o del mismo, Spring trata de no crear una dependencia entre la herramienta ORM, el mismo Spring y el código de la aplicación, para que cuando sea necesario migrar a un nuevo ORM no sea necesario realizar tantos cambios.
- Facilidad de prueba: Spring trata de crear pequeños pedazos que se puedan aislar y probar por separado, ya sean sesiones o una fuente de datos (datasource).

2.2.5 Spring DAO

El patrón DAO (Data Access Object) es uno de los patrones más importantes y usados en aplicaciones J2EE, y la arquitectura de acceso a los datos de Spring provee un buen soporte para este patrón.

2.2.5.1 DAO y JDBC

Existen dos opciones para llevar a cabo el acceso, conexión y manejo de bases de datos: utilizar alguna herramienta ORM o utilizar el template de JDBC (Java Database Connectivity) que brinda Spring. La elección de una de estas dos herramientas es totalmente libre y en lo que se debe basar el desarrollador para elegir es en la complejidad de la aplicación. En caso de ser una aplicación sencilla en la cual únicamente una clase hará dicha conexión, entonces la mejor opción sería el Spring JDBC o en caso contrario que se requiera un mayor soporte y sea más robusta la aplicación se recomienda utilizar una herramienta ORM.

El uso de JDBC muchas veces lleva a repetir el mismo código en distintos lugares, al crear la conexión, buscar información, procesar los resultados y cerrar la conexión. El uso de las dos tecnologías mencionadas anteriormente ayuda a mantener simple este código y evitar que sea tan repetitivo, además minimiza los errores al intentar cerrar la conexión con alguna base datos. Este es un ejemplo de como se hace la inserción de datos en una base de datos utilizando JDBC tradicional:

```
public void insertPerson(Person person) throws SQLException {
//Se declaran recursos
Connection conn = null;
PreparedStatement stmt = null;
try {
```

```
        //Se Abre una conexión
        conn = dataSource.getConnection();
        //Se crea la declaraciòn
        stmt = conn.prepareStatement("insert into person (" +
        "id, firstName, lastName) values (?, ?, ?)");
        //Se Introducen los parámetros
        stmt.setInt(0, person.getId().intValue());
        stmt.setString(1, person.getFirstName());
        stmt.setString(2, person.getLastName());
        //Se ejecuta la instrucciòn
        stmt.executeUpdate();
}
//Se atrapan las excepciones
catch(SQLException e) {
LOGGER.error(e);
}
finally {
        //Se limpian los recursos
        try { if (stmt != null) stmt.close(); }
        catch(SQLException e) { LOGGER.warn(e); }
        try { if (conn != null) conn.close(); }
        catch(SQLException e) { LOGGER.warn(e); }
        }
}
```

De todo este código únicamente el 20% es para un propósito específico, el otro 80% es de propósito general en cada una de las diferentes operaciones de acceso a una base de datos. Esto no quiere decir que no sea importante si no que al contrario es lo que le da la estabilidad y robustez al guardado y recuperación de información.

Las clases bases que Spring provee para la utilización de los DAO son abstractas y brindan un fácil acceso a recursos comunes de base de datos. Existen diferentes implementaciones para cada una de las tecnologías de acceso a datos que soporta Spring.

Para JDBC existe la clase JdbcDaoSupport que provee métodos para accesar al DataSource y al template pre-configurado que se mencionó anteriormente, JdbcTemplate.

Únicamente se extiende la clase JdbcDaoSupport y se le da una referencia al DataSource actual.

"En el ejemplo que se muestra a continuación la clase base es JdbcTemplate, que es la clase central que maneja la comunicación con la base de datos y el manejo de excepciones.". Esta clase provee varios métodos que realizan la carga pesada y resultan bastante convenientes, como el método execute() que aparece en el ejemplo, el cual recibe un comando de SQL como su único parámetro. Para este ejemplo se supone que se tiene una tabla ya creada dentro de una base de datos utilizando el siguiente comando SQL:

```
create table mytable (id integer, name varchar(100))
```

Ahora se quiere agregar algunos datos de prueba:

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
public class MinimalTest extendí TestCase{
        private DriverManagerDataSource dataSource;
        public void setup(){
        dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
```

```
        dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        JdbcTemplate Jt = new JdbcTemplate(dataSource);
        Jt.execute("delete from mytable");
        Jt.execute("insert into mytable (id,name) values(1,`John´)");
        Jt.execute("insert into mytable (id,name) values(2,`Jane´)");
}
public void testSomething(){
// Aquí va el codigo de prueba que se puede aplicar
}
```

Como se pudo apreciar en el ejemplo anterior no existe manejo de excepciones, ya que el JdbcTemplate se encarga de atrapar todas las SQLExceptions y las convierte a una subclase de DataAccessException, que es la clase que se encuentra en el nivel superior de la jerarquía de las excepciones de acceso a datos. También el JdbcTemplate se encarga de controlar la conexión con la base de datos. Y se puede evitar configurar el DataSource en DAO, únicamente se tiene que establecer a través de la inyección de dependencia.

2.2.6 Spring Web

El módulo web de Spring se encuentra en la parte superior del módulo de contexto, y provee el contexto para las aplicaciones web. Este módulo también provee el soporte necesario para la integración con el framework Struts de Yakarta.

Este módulo también se encarga de diversas operaciones web como por ejemplo: las peticiones multi-parte que puedan ocurrir al realizar cargas de archivos y la relación de los parámetros de las peticiones con los objetos correspondientes (domain objetcs o Business objects).

2.2.7 Spring Web MVC

Spring brinda un MVC (Model View Controller) para web bastante flexible y altamente configurable, pero esta flexibilidad no le quita sencillez, ya que se pueden desarrollar aplicaciones sencillas sin tener que configurar muchas opciones.

Para esto se puede utilizar muchas tecnologías ya que Spring brinda soporte para JSP, Struts, Velocity, entre otros.

El MVC de Spring presenta una arquitectura Tipo 2, para mayor información consultar el Capítulo 2 de este documento de tesis.

El Web MVC de Spring presenta algunas similitudes con otros frameworks para web que existen en el mercado, pero son algunas características que lo vuelven único:

- Spring hace una clara división entre controladores, modelos de JavaBeans y vistas.
- El MVC de Spring está basado en interfaces y es bastante flexible.
- Provee interceptores (interceptors) al igual que controladores.
- Spring no obliga a utilizar JSP como única tecnología View también se puede utilizar otras.

- Los Controladores son configurados de la misma manera que los demás objetos en Spring, a través de IoC.
- Los web tiers son más sencillos de probar que en otros frameworks.
- El web tiers se vuelve una pequeña capa delgada que se encuentra encima de la capa de business objects.

La arquitectura básica de Spring MVC está ilustrada en la Figura 5.

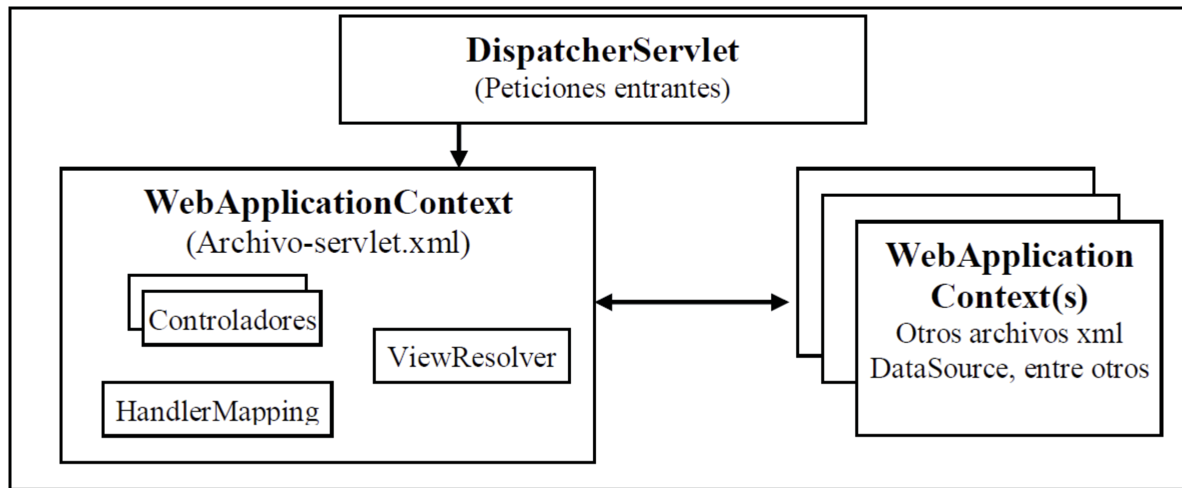

Figura 5: Arquitectura básica del Web MVC de Spring

Para intentar comprender cada parte de la arquitectura del Web MVC de Spring se presenta en la Figura 6, el ciclo de vida de una petición o request:

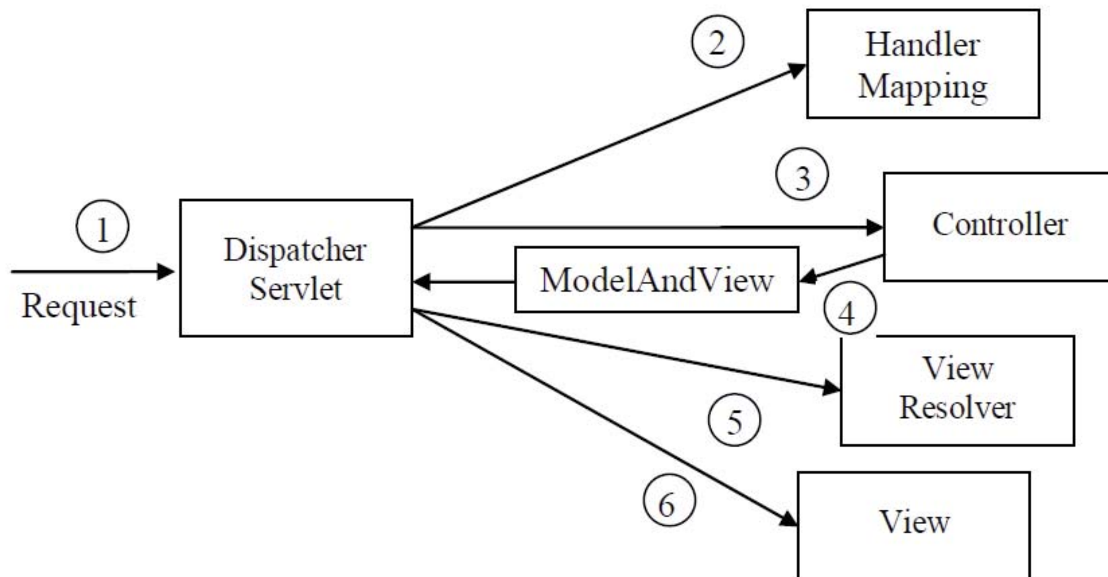

Figura 6: Ciclo de vida de un request

1. El navegador manda un request y lo recibe un DispatcherServlet.
2. Se debe escoger que Controller manejará el request, para esto el HandlerMapping mapea los diferentes patrones de URL hacia los controladores, y se le regresa al DispatcherServlet el Controller elegido.
3. El Controller elegido toma el request y ejecuta la tarea.
4. El Controller regresa un ModelAndView al DispatcherServlet.
5. Si el ModelAndView contiene un nombre lógico de un View se tiene que utilizar un ViewResolver para buscar ese objeto View que representará el request modificado.
6. Finalmente el DispatcherServlet despacha el request al View.

Spring cuenta con una gran cantidad de controladores de los cuales se puede elegir dependiendo de la tarea, entre los más populares se encuentra: Controller y AbstractController para tareas sencillas; el SimpleFormController ayuda a controlar formularios y el envío de los mismos, MultiActionController ayuda a tener varios métodos dentro un solo controlador a través del cual se podrán mapear las diferentes peticiones a cada uno de los métodos correspondientes.

2.2.7.1 Dispatcher Servlet

Para configurar el DispatcherServlet como el servlet central, se tiene que hacer como cualquier servlet normal de una aplicación web, en el archivo de configuración web.xml (Deployment Descriptor).

```
<servlet>
<servlet-name>training</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>ejemplo</servlet-name>
<url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Entonces el DispatcherServlet buscará como está indicado por el tag <servletname> el contexto de aplicación (application Context) correspondiente con el nombre que se haya puesto dentro de ese tag acompañado de la terminación –servlet.xml, en este caso buscará el archivo ejemplo-servlet.xml. En donde se pondrán las definiciones y como su nombre lo indica el contexto de la aplicación dentro de diferentes beans, con sus correspondientes propiedades y atributos, para el caso del Web MVC, se pondrán los diferentes ViewResolver a utilizar, los controladores y sus propiedades, el HandlerMapping, así como los diferentes beans que sean necesarios.

2.2.7.2 Handler Mappings

Existen diversas maneras en que el DispatcherServlet pueda determinar y saber que controlador es el encargado de procesar un request, y a que bean del Application Context se lo puede asignar. Esta tarea la lleva a cabo el Handler Mapping o el manejador de mapeo, existen 2 tipos principales que son los que mas se usan:

- BeanNameUrlHandlerMapping: mapea el URL hacia el controlador en base al nombre del bean del controlador. Ejemplo de cómo se declara:

Esta es la declaración principal del Handler Mapping:

```
<bean id="beanNameUrlMapping" class="org.springframework.web.
servlet.handler.BeanNameUrlHandlerMapping"/>
```

A continuación se escribe cada uno de los diferentes URLs que se vayan a utilizar en la aplicación, poniéndolo en el atributo name del bean y en el atributo class la clase del Controlador que vaya a procesar ese dicho request, y finalmente se ponen las diferentes propiedades que utilizará dicho controlador.

```
<bean name="/ejemplo.htm" class="web.ExampleController">
<property name="Servicio">
<ref bean="Servicio"/>
</property>
</bean>
```

Así cuando el usuario entre a una página con el URL /ejemplo.htm el request que haga será dirigido hacia el controlador ExampleController

- SimpleUrlHandlerMapping: mapea el URL hacia el controlador basando en una colección de propiedades declarada en el applicationContext. Ejemplo de cómo declarar un HandlerMapping de este tipo:

```
<bean id="simpleUrlMapping" class=
"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/ejemplo.htm">ExampleController</prop>
<prop key="/login.htm">LoginController</prop>
</props>
</property>
</bean>
```

En este Handler Mapping se declara una lista de propiedades en las cuales se pondrá cada uno de los URLs como una propiedad, con el URL como atributo key y como valor el nombre del bean del controlador que será responsable de procesar la petición o request. Por ejemplo el URL /login.htm será responsabilidad del controlador con el nombre del bean LoginController.

Se pueden trabajar con múltiples Handler Mappings por si son necesarios en diferentes situaciones, únicamente se le tiene que agregar la propiedad “order” para poder asignarle en que orden el DispatcherServlet va a considerarlas para poder invocarlas.

2.2.7.3 View Resolvers

En el Spring MVC una vista o View es un bean que transforma los resultados para que sean visibles para el usuario y los pueda interpretar de una mejor forma. En sí un View Resolver es cualquier bean que implemente la interfaz org.springframework.web.servlet.ViewResolver. . Esto quiere decir que un View Resolver es

el encargado de resolver el nombre lógico que regresa un controlador en un objeto ModelAndView, a un nombre de archivo físico que el navegador podrá desplegarle al usuario junto con los resultados procesados. Spring MVC cuenta con cuatro View Resolvers diferentes:

- InternalResourceViewResolver: Resuelve los nombres lógicos en un archivo tipo View que es convertido utilizando una plantilla de archivos como JSP, JSTL o Velocity
- BeanNameViewResolver: Resuelve los nombres lógicos de las vistas en beans de tipo View en el applicationContext del DispatcherServlet
- ResourceBundleViewResolver: Resuelve los nombres lógicos de las vistas en objetos de tipo View contenidos en un ResourceBundle o un archivo con extensión .properties.
- XMLViewResolver: Resuelve los nombres los lógicos de las vistas que se encuentran en un archivo XML separado.

El View Resolver más utilizado es el InternalResourceViewResolver, y se especifica en el web applicationContext de nuestra aplicación de la siguiente manera:

```
<bean id="viewResolver" class="org.springframework.web.
servlet.view.InternalResourceViewResolver">
       <property name="viewClass">
       <value>org.springframework.web.servlet.view.JstlView</value>
       </property>
       <property name="prefix"><value>/WEB-INF/jsp/</value></property>
       <property name="suffix"><value>.jsp</value></property>
</bean>
```

En este bean de id viewResolver, se especifica de que clase se quiere que se implemente, para este caso será el tipo de View Resolver que se quiere de los 4 mencionados anteriormente. Después vienen tres propiedades:

- viewClass: sirve para especificar que tipo de plantilla se quiere usar para desplegar la vista, en este caso se utilizó JSTL (Java Standard Tag Library), que es una de las plantillas más comunes para este tipo de tarea, también se puede seleccionar Velocity o Tiles para desplegar la información
- prefix: el prefijo que antecederá al nombre lógico de la vista, generalmente es la ruta donde se encontrarán los JSP, Ejemplo: /WEB-INF/jsp
- suffix: el sufijo que tendrán nuestras vistas, ya que la mayoría de los nombres físicos que se utilizan son JSPs se le asigna la extensión .jsp.

Todo esto es con el afán de tener una programación que no sea tan dependiente, ya que si se quisiera cambiar de carpeta todas las vistas, lo único que se tendría que modificar sería el prefix del viewResolver.

2.2.7.4 Controladores

Si el DispatcherServlet es el corazón del Spring MVC los controladores son los cerebros. Existen varios controladores cada uno especializado para una tarea en particular, claro que como en todos los casos existen algunos que son de uso general. Para poder crear un controlador basta con implementar la interfaz del Controlador deseado, sobrescribir los

métodos que sean necesarios para procesar la petición. Esta situación ayuda a poder modularizar la aplicación ya que con la combinación de diversos controladores que sean enfocados a una tarea en particular se puede concentrarse en cada parte de aplicación aislada y tener una mejor estructura que ayudará a detectar más fácilmente las fallas y errores que puedan surgir.

Existe una variedad de controladores, como se muestra en la Figura 7, los cuales poseen una jerarquía. Spring brinda libertad al desarrollador de escoger que tipo de controlador desea implementar, no lo limita como en algunos otros frameworks.

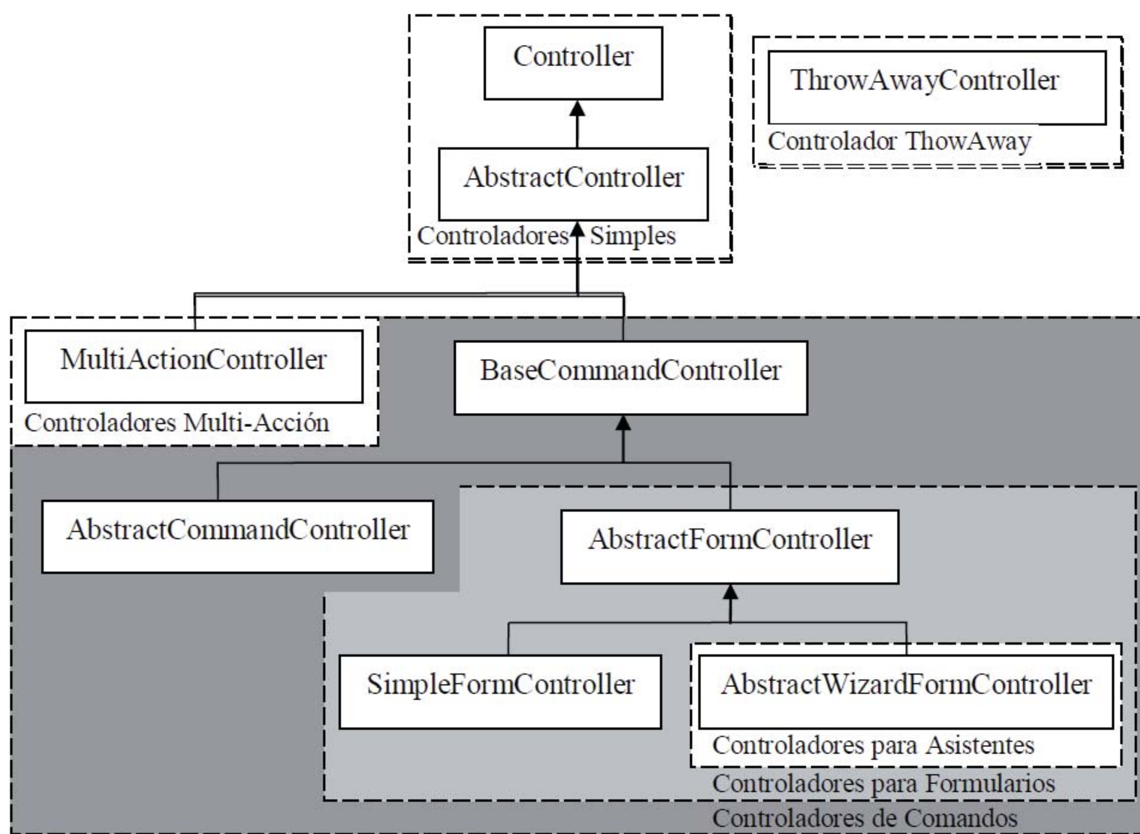

Figura 7: Controladores que provee Spring

La manera en que trabaja cada uno de dichos controladores es similar, cada uno tiene su método específico para procesar las peticiones que haga el usuario, pero todos regresan un objeto tipo ModelAndView, que es como su nombre lo dice el modelo y la vista, ya que se está compuesto de 2 o más atributos. El principal es el nombre de la vista a la que se va a regresar el modelo para que sea desplegado, y los demás atributos pueden ser parámetros que se le agregan por medio del método .addObject("nombre_parámetro",valor_parámetro). Además el Modelo puede estar formado por un solo objeto o por un Map de Objetos, los cuales se identificarán en la vista con el mismo nombre que se haya mandado dentro del Modelo que se le dio al objeto ModelAndView que se esté regresando para ser procesado.

El siguiente ejemplo muestra el código básico de un controlador.

```
public class MoviesController extends AbstractController {
//Método que controla el Request
public ModelAndView handleRequestInternal(
HttpServletRequest request, HttpServletResponse response)
throws Exception {
       //Se recupera una lista de todas las peliculas
       List movies = MovieService.getAllMovies();
       //Se manda a la vista movieList el objeto movies, con el nombre
       lógico //movies
       return new ModelAndView("movieList", "movies", movies);
       }
}
```

El objeto ModelAndView contiene el nombre lógico de la vista, el cual el framework se encarga a través del View Resolver de convertir ese nombre lógico al nombre físico que es el que buscará el servidor web.

2.2.7.5 Procesamiento de formularios

Una de las increíbles facilidades que brinda Spring es la de llenar, recibir y procesar formularios, a través de los 3 diferentes controladores que brinda Spring para el manejo de formas o formularios. Ya que con estos controladores, se facilita el procesamiento, llenado de la forma, almacenamiento (si es necesario) de los datos, así como desplegar errores y confirmaciones. Para el usuario todo este proceso es totalmente transparente ya que el formulario que este llenando tendrá el mismo formato y los mismos elementos que cualquier otro formulario normal.

Existen varios pasos a seguir para poder utilizar un controlador de formularios, uno de ellos rellenar el archivo .jsp con tags que provee la librería spring.tld. Esta librería se debe de combinar de la siguiente manera: En el Deployment Descriptor de la aplicación se pone el siguiente tag:

```
<taglib>
<taglib-uri>/spring</taglib-uri>
<taglib-location>/WEB-INF/spring.tld</taglib-location>
</taglib>
```

Una vez realizada esta acción se configura el applicationContext de la aplicación con el bean que describirá toda la información que el controlador necesitará.

Para el control de las formas, se recomienda crear un Objeto que tenga atributos, cada uno de ellos con sus respectivos métodos set y get, para que a través del patrón de Inyección de Dependencia se pueda poner la información dentro de ese objeto y sea más fácil de manipular y almacenar. A este objeto se le conoce como “objeto comando” o command object.

Estas son algunas de las propiedades que el controlador necesita, no todas son obligatorias:

- formView: nombre lógico de la vista que contiene la forma.
- successView: nombre lógico de la vista que se desplegará una vez que la forma sea llenada con éxito.

- commandClass: la clase a la que pertenecerá el Objeto comando u objeto base sobre el cual se va a trabajar.
- commandName: nombre lógico que se le quiere dar al objeto comando, con el cual se le reconocerá en la vista.
- validator: este será el bean de referencia de la clase que extiende a la interfaz org.springframework.validation.Validator que se encarga después de que se hizo la vinculación (binding) de validar que todos los campos cumplan con los requisitos que le desarrollador desee y regresará los errores y las violaciones, y en que campo fue donde se cometieron, así como el mensaje de error que se debe desplegar.

Este es un ejemplo de un bean de configuración de un SimpleFormController:

```
<bean id="registerController" class="web.RegisterController">
<property name="formView">
<value>newStudentForm</value>
</property>
<property name="successView">
<value>studentWelcome</value>
</property>
<property name="commandClass">
<value>business.User</value>
</property>
<property name="commandName">
<value>user</value>
</property>
<property name="validator">
<ref bean="registerValidator"/>
</property>
</bean>
<bean id="registerValidator" class="validators.RegisterValidator"/>
```

Una vez realizada esta configuración se debe crear una clase para el controlador, la cual debe extender a la clase SimpleFormController, en la cual se pueden sobrescribir varios métodos como por ejemplo:

- Map referenceData(HttpServletRequest req, Object command, BindException errors): en este método se realiza el llenado de los campos que el desarrollador desee que aparezcan llenos una vez que se cargue el formulario. Estos campos vendrán llenos con los datos que se encuentren en el Map que será regresado. Este se mezclará con el modelo para que la vista los procese y se desplieguen en los campos del formulario necesarios.
- ModelAndView onSubmit(HttpServletRequest req, HttpServletResponseres, Object command, BindException errors): Este método puede tener variaciones en cuanto al número de parámetros que recibe, ya que existen formas más sencillas de este método. Estos métodos son llamados después de que ocurrió la validación y en el objeto BindException no hubo ningún error, se ejecuta esté método, en el cual se realiza la acción que se desee una vez que el usuario haya enviado el formulario, ya sea almacenar la información en una base de datos, etc.
- void doSubmitAction(HttpServletRequest req): este método es la forma más simple de los métodos para enviar los formularios ya que no se necesita crear un objeto ModelAndView, el método crea uno nuevo con la instancia successView que se haya

dado en la configuración del controlador en el applicationContext. Y se pueden llevar a cabo las mismas operaciones que en cualquier método de envío, por ejemplo almacenar en una base de datos la información del formulario.

Ahora solamente queda utilizar los tags dentro del JSP para poder vincular cada uno de los campos a un atributo del Objeto comando que se haya escogido. Para esto se utiliza el tag <spring:bind> que únicamente tiene el atributo path, que es donde se asigna a que atributo del objeto comando se va a referir ese campo del formulario. Este tag trae dentro también los errores de validación que se llegarán a dar al momento de enviar el formulario.

Esto se da gracias a un objeto status de tipo Bind-Status que a través de 3 parámetros ayudará a vincular:

- expression: La expresión usada para recuperar el nombre del campo, si por ejemplo la propiedad es Movie.name, el valor de la expresión y el nombre del campo será name.
- value: El valor del campo asociado, una vez que fue enviada la forma se utiliza para volver a poner el valor de este campo si es que hubo errores y que no se pierda toda la información que el usuario proporciono.
- errorMessages: Un arreglo de Strings que contiene los errores relacionados con este campo.

Ejemplo del código fuente de un JSP utilizando los tags de vinculación

```
<spring:bind>.
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="spring" uri="/spring" %>
…
<form method="POST" action="/registerUser.htm">
<spring:bind path="user.firstName">
Nombre:
<input type="text"
name="<c:out value="${status.expression}"/>"
value="<c:out value="${status.value}"/>">
<p>Errores:<c:out value=”${status.errorMessages}”/></p>
</spring:bind>
…
</form>
```

Una de las combinaciones más importante en cuanto a las vistas JSP es utilizar la tecnología JSTL para poder realizar ciclos, condiciones, entre otros de una manera más rápida, sencilla y eficaz para el desarrollador.

2.3 Complementos de Spring Framework.

2.3.1 Hibernate.

Hibernate es una herramienta de Mapeo objeto-relacional para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

2.3.1.1 Características

Como todas las herramientas de su tipo, Hibernate busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación; el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional). Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información Hibernate le permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la POO. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL. Hibernate genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

Hibernate ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo que una API para construir las consultas programáticamente (conocida como "criteria").

Hibernate para Java puede ser utilizado en aplicaciones Java independientes o en aplicaciones Java EE, mediante el componente Hibernate Annotations que implementa el estándar JPA, que es parte de esta plataforma.

2.3.1.2 Historia

Hibernate fue una iniciativa de un grupo de desarrolladores dispersos alrededor del mundo conducidos por Gavin King.

Tiempo después, JBoss Inc. (empresa comprada por Red Hat) contrató a los principales desarrolladores de Hibernate y trabajó con ellos en brindar soporte al proyecto.

La rama actual de desarrollo de Hibernate es la 3.x, la cual incorpora nuevas características, como una nueva arquitectura Interceptor/Callback, filtros definidos por el usuario, y opcionalmente el uso de anotaciones para definir la correspondencia en lugar de (o conjuntamente con) los archivos XML. Hibernate 3 también guarda cercanía con la especificación EJB 3.0 (aunque apareciera antes de la publicación de dicha especificación por Java Community Process) y actúa como la espina dorsal de la implementación de EJB 3.0 en JBoss.

3 Enterprise JavaBeans (EJB)

Los Enterprise JavaBeans (también conocidos por sus siglas EJB) son una de las API que forman parte del estándar de construcción de aplicaciones empresariales J2EE (ahora JEE 5.0) de Oracle Corporation (inicialmente desarrollado por Sun Microsystems). Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor que son, precisamente, los EJB:

- Comunicación remota utilizando CORBA
- Transacciones
- Control de la concurrencia
- Eventos utilizando JMS (Java messaging service)
- Servicios de nombres y de directorio
- Seguridad
- Ubicación de componentes en un servidor de aplicaciones.

La especificación de EJB define los papeles jugados por el contenedor de EJB y los EJB, además de disponer los EJB en un contenedor.

3.1 Definición

Los EJB proporcionan un modelo de componentes distribuido estándar del lado del servidor. El objetivo de los EJB es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, etc.) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes permite que éstos sean flexibles y sobre todo reutilizables.

No hay que confundir los Enterprise JavaBeans con los JavaBeans. Los JavaBeans también son un modelo de componentes creado por Oracle - Sun Microsystems para la construcción de aplicaciones, pero no pueden utilizarse en entornos de objetos distribuidos al no soportar nativamente la invocación remota (RMI) [2].

3.2 Desarrollo basado en componentes

Con la tecnología J2EE Enterprise JavaBeans es posible desarrollar componentes (enterprise beans) que luego se puede reutilizar y ensamblar en distintas aplicaciones que se tenga que hacer para la empresa. Por ejemplo, se podría desarrollar un bean Cliente que represente un cliente en una base de datos. Se podría usar después ese bean Cliente en un programa de contabilidad o en una aplicación de comercio electrónico o virtualmente en cualquier programa en el que se necesite representar un cliente. De hecho, incluso sería posible que el desarrollador del bean y el ensamblador de la aplicación no fueran la misma persona, o ni siquiera trabajaran en la misma empresa.

El desarrollo basado en componentes promete un paso más en el camino de la programación orientada a objetos. Con la programación orientada a objetos se puede reutilizar clases, pero con componentes es posible reutilizar n mayor nivel de funcionalidades e incluso

es posible modificar estas funcionalidades y adaptarlas a cada entorno de trabajo particular sin tocar el código del componente desarrollado. El contenedor de componentes se denomina contenedor EJB y es algo así como el sistema operativo en el que éstos residen. Recordar que en Java existe un modelo de programación de objetos remotos denominado RMI. Con RMI es posible enviar peticiones a objetos que están ejecutándose en otra máquina virtual Java. Podemos ver un componente EJB como un objeto remoto RMI que reside en un contenedor EJB que le proporciona un conjunto de servicios adicionales.

Cuando se esté trabajando con componentes se tendrá que dedicar tanta atención al despliegue (deployment) del componente como a su desarrollo. Se entenderá por despliegue la incorporación del componente a nuestro contenedor EJB y a nuestro entorno de trabajo (bases de datos, arquitectura de la aplicación, etc.). El despliegue se define de forma declarativa, mediante un fichero XML (descriptor del despliegue, deployment descriptor) en el que se definen todas las características del bean. [7]

El desarrollo basado en componentes ha creado expectativas sobre la aparición de una serie de empresas dedicadas a implementar y vender componentes específicos a terceros. Este mercado de componentes nunca ha llegado a tener la suficiente masa crítica como para crear una industria sostenible. Esto es debido a distintas razones, como la dificultad en el diseño de componentes genéricos capaces de adaptarse a distintos dominios de aplicación, la falta de estandarización de los dominios de aplicación o la diversidad de estos dominios. Aun así, existe un campo creciente de negocio en esta área (por ejemplo, www.componentsource.com).

3.3 Servicios proporcionados por el contenedor EJB

En el apartado anterior se ha comentado que la diferencia fundamental entre los componentes y los objetos clásicos reside en que los componentes viven en un contenedor EJB que los envuelve proporcionando una capa de servicios añadidos. ¿Cuáles son estos servicios? Los más importantes son los siguientes:

- Manejo de transacciones: apertura y cierre de transacciones asociadas a las llamadas a los métodos del bean.
- Seguridad: comprobación de permisos de acceso a los métodos del bean.
- Concurrencia: llamada simultánea a un mismo bean desde múltiples clientes.
- Servicios de red: comunicación entre el cliente y el bean en máquinas distintas.
- Gestión de recursos: gestión automática de múltiples recursos, como colas de mensajes, bases de datos o fuentes de datos en aplicaciones heredadas que no han sido traducidas a nuevos lenguajes/entornos y siguen usándose en la empresa.
- Persistencia: sincronización entre los datos del bean y tablas de una base de datos.
- Gestión de mensajes: manejo de Java Message Service (JMS).
- Escalabilidad: posibilidad de constituir clusters de servidores de aplicaciones con múltiples hosts para poder dar respuesta a aumentos repentinos de carga de la aplicación con sólo añadir hosts adicionales.
- Adaptación en tiempo de despliegue: posibilidad de modificación de todas estas características en el momento del despliegue del bean.

Se podría pensar en lo complicado que sería programar una clase a mano que implementara todas estas características. Como se suele decir, la programación de EJB es sencilla si la comparamos con lo que habría que implementar de hacerlo todo por uno mismo. Evidentemente, si en la aplicación que se está desarrollando no se va a necesitar estos servicios y va a tener un interfaz web, se podría utilizar simplemente páginas JSP y JDBC.

3.4 Funcionamiento de los componentes EJB

El funcionamiento de los componentes EJB se basa fundamentalmente en el trabajo del contenedor EJB. El contenedor EJB es un programa Java que corre en el servidor y que contiene todas las clases y objetos necesarios para el correcto funcionamiento de los enterprise beans.

En la figura siguiente se puede ver una representación de muy alto nivel del funcionamiento básico de los enterprise beans. En primer lugar, se puede ver que el cliente que realiza peticiones al bean y el servidor que contiene el bean está ejecutándose en máquinas virtuales Java distintas. Incluso pueden estar en distintos hosts. Otra cosa a resaltar es que el cliente nunca se comunica directamente con el enterprise bean, sino que el contenedor EJB proporciona un EJBObject que hace de interfaz. Cualquier petición del cliente (una llamada a un método de negocio del enterprise bean) se debe hacer a través del objeto EJB, el cual solicita al contenedor EJB una serie de servicios y se comunica con el enterprise bean. Por último, el bean realiza las peticiones correspondientes a la base de datos. [8] El contenedor EJB se preocupa de cuestiones como:

- ¿Tiene el cliente permiso para llamar al método?
- Hay que abrir la transacción al comienzo de la llamada y cerrarla al terminar.
- ¿Es necesario refrescar el bean con los datos de la base de datos?

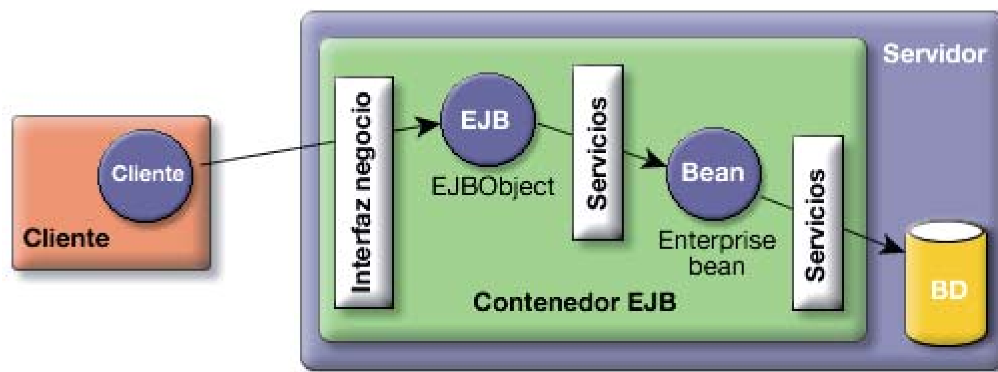

Figura 8: Representación de alto nivel del funcionamiento de los enterprise beans

Se colocará un ejemplo para poder entender mejor el flujo de llamadas. Suponer que se tiene una aplicación de bolsa y el bean proporciona una implementación de un Broker. La interfaz de negocio del Broker está compuesta de varios métodos, entre ellos, por ejemplo, los métodos compra o vende. Supongamos que desde el objeto cliente queremos llamar al método compra. Esto va a provocar la siguiente secuencia de llamadas:

1. Cliente: "Necesito realizar una petición de compra al bean Broker."
2. EJBObject: "Espera un momento, necesito comprobar tus permisos."
3. Contenedor EJB: "Sí, el cliente tiene permisos suficientes para llamar al método compra."
4. Contenedor EJB: "Necesito un bean Broker para realizar una operación de compra. Y no olvidar comenzar la transacción en el momento de instanciarlos."
5. Pool de beans: "A ver... ¿a quién de nosotros le toca esta vez?".
6. Contenedor EJB: "Ya tengo un bean Broker. Pásale la petición del cliente."

3.5 La arquitectura EJB en detalle

Los contenidos que vamos a ver en este apartado son:

- RMI: stubs y skeletons, paso de argumentos
- EJB y RMI
- Interfaces remotas y locales
- Funcionamiento de la clase home
- La arquitectura de los distintos tipos de beans

3.5.1 Repaso de RMI

3.5.1.1 Stubs, skeletons y paso de argumentos

En este apartado se repasara algunos conceptos fundamentales para entender el funcionamiento de la arquitectura EJB; los stubs y skeletons, los objetos remotos y el paso de parámetros y devolución de resultados en las llamadas remotas.

RMI (Remote Method Invocation) define la forma de comunicación remota entre objetos Java situados en máquinas virtuales distintas. Supongamos que un objeto cliente quiere hacer una petición a un objeto remoto situado en otra JVM (Máquina Virtual Java, Java Virtual Machine). RMI pretende hacer trasparente la presencia de la red, de forma que cuando se escriba el código de los objetos clientes y remotos no se tenga que tratar con la complicación de gestionar la comunicación física por la red.

Para ello, RMI proporciona al cliente un objeto proxy (llamado stub) que recibe la petición del objeto cliente y la transforma en algo que se puede enviar por la red hasta el objeto remoto. Este stub se hace cargo de todos los aspectos de bajo nivel (streams y sockets). En el lado del servidor, un objeto similar (llamado skeleton) recibe la comunicación, la desempaqueta y lanza el método correspondiente del objeto remoto al que sirve. Al igual que la petición, se deben empaquetar los argumentos de la llamada. El programador sólo debe definir el código del método en el objeto remoto. Los objetos stub y skeleton los construye el compilador de RMI de forma automática.

La siguiente figura muestra un ejemplo con el objeto remoto SaludoImpl:

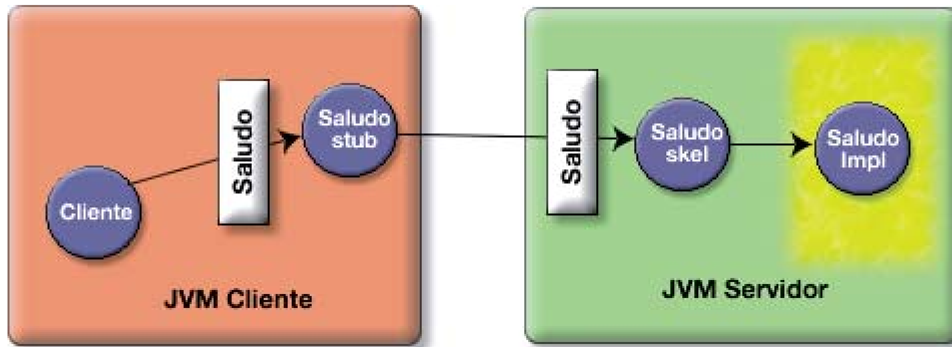

Figura 9: Ejemplo con el objeto remoto SaludoImpl

El objeto remoto SaludoImpl está sombreado para indicar que en él se encuentra la implementación de los métodos remotos.

Una vez realizada la llamada, el stub queda en espera (y el objeto cliente que ha llamado al método correspondiente del stub) hasta recibir la respuesta del skeleton (respuesta que debe proporcionar el método invocado en el objeto remoto). Si la respuesta no se recibe, el stub lanza una excepción que debe capturar el objeto cliente.

Es posible que en el servidor no exista un objeto skeleton por cada objeto remoto sino que, para hacer la arquitectura más eficiente, se pueda definir un objeto genérico que distribuya las peticiones a los objetos remotos con algún mecanismo de identificación de la petición y de caché de objetos remotos. Por eso en las siguientes imágenes que se muestren no aparecerá este objeto.

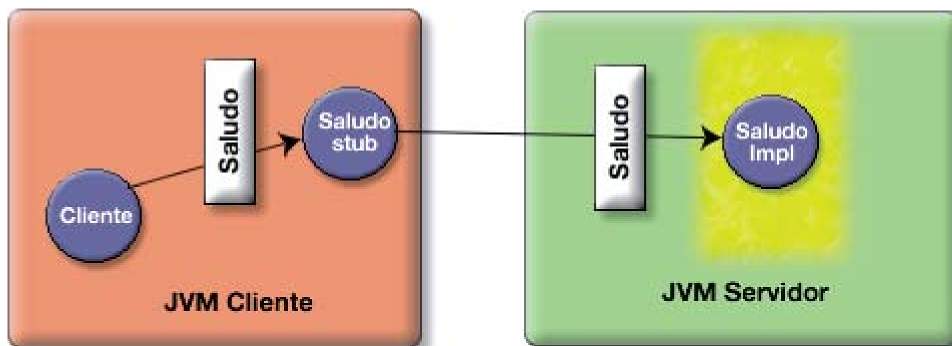

Figura 10: Ejemplo de la omisión del objeto skeleton

Concretando más, para implementar y usar una clase remota con RMI se debe cumplir las siguientes condiciones:

- Se deben definir la clase remota como una interfaz que hereda de la interfaz java.rmi.Remote.
- Todos los métodos de esa interfaz deben declarar la excepción RemoteException.
- Se debe definir una clase que implemente la interfaz.
- Se debe llamar al compilador rmic para que cree las clases stub y skeleton

- Un servidor debe crear uno o más objetos remotos y asignarles un nombre a cada uno.
- Algún cliente debe localizar un objeto remoto, referenciando su nombre, obtener el stub (que implementa la clase remota) y realizar las llamadas al stub.

La siguiente imagen muestra la estructura de clases e interfaces que se usan para definir una sencilla clase remota llamada Saludo. Se define la interfaz Saludo que extiende la interfaz Remote. La clase SaludoImpl la escribe el programador e implementa la interfaz Saludo. La clase SaludoStub también implementa la interfaz Saludo y la construye el compilador de RMI. Por último, el cliente se comunica con un objeto instancia de la clase SaludoStub que implementa la interfaz Saludo. Esta interfaz es la única que ve el cliente.

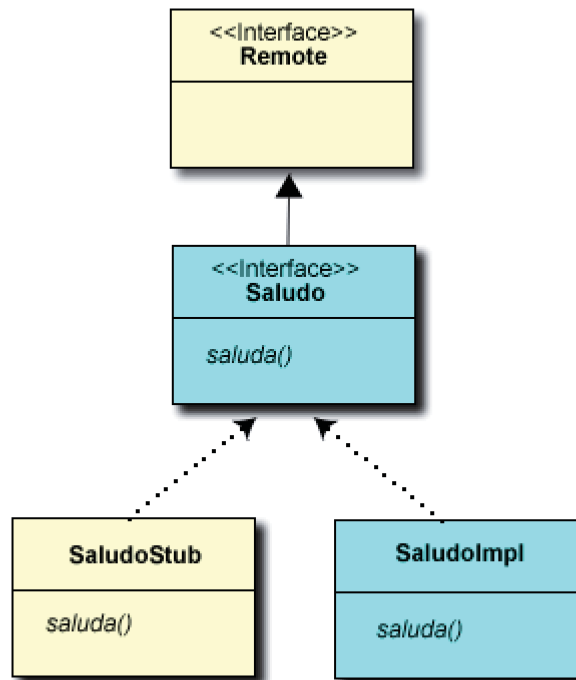

Figura 11: Estructura de clases e interfaces para definir una clase remota llamada Saludo

Al implementar la interfaz Saludo, los objetos de la clase SaludoImp son también objetos Remote, ya que la interfaz Saludo hereda de la interfaz Remote.

Muy importante, a clase stub se debe instalar en la JVM del objeto cliente para que éste pueda usarla. Pregunta:

Suponer que se tiene tres objetos instanciados de una misma clase remota, y se quiere acceder a ellos desde un objeto cliente. ¿Cuántos stubs se necesitarían? ¿Cuántos skeletons?

3.5.1.2 Paso de argumentos

En cuanto a los argumentos y los valores devueltos por las llamadas remotas, deben ser de uno de los siguientes tipos:

- Objetos primitivos
- Objetos serializables
- Un array o un colección de objetos primitivos o serializables
- Un objeto java.rmi.Remote

Un caso muy interesante, por la frecuencia con la que sucede en la arquitectura EJB, es el de un objeto remoto que es devuelto en una llamada a otro objeto también remoto. El siguiente ejemplo proporciona una representación de lo que ocurre en este caso.

Suponer que se tiene un conjunto de objetos remotos de la clase Estudiante, cada uno con un identificador determinado. Suponer también un objeto remoto llamado estudianteFactory que puede localizar al objeto Estudiante con un identificador determinado.

- El cliente invoca el método getEstudiante("Id#2334") del stub de estudianteFactory. El stub trasmite la llamada al skeleton.
- El skeleton desempaqueta la llamada e invoca el método getEstudiante("Id#2334") en el objeto remoto estudianteFactory. El objeto remoto localiza el objeto estudiante que se solicita y devuelve su referencia al Skeleton. Al ser estudiante un objeto remoto, estará disponible su stub en la JVM.
- El objeto skeleton estudianteFactory serializa el stub de estudiante y se lo pasa al skeleton, el cual lo trasmite al stub. El stub lo desempaqueta y crea una copia local en la JVM del cliente. Por último, el stub devuelve al objeto cliente una referencia local al stub de estudiante.

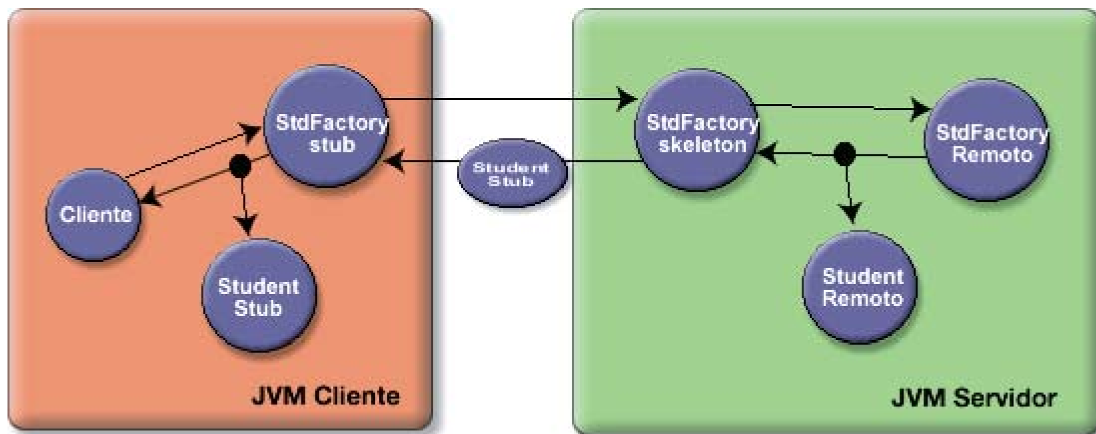

Figura 12: Objeto remoto devuelto en una llamada a otro objeto

Pregunta:

¿Cuáles son las ventajas de pasar un objeto remoto en lugar de un objeto real Estudiante? ¿Cuáles son los inconvenientes?

Es muy importante recordar que cuando se pasa un objeto serializado de una JVM a otra, la JVM que lo recibe debe tener disponible el fichero class correspondiente al tipo del

objeto que se pasa. Esto sucede así incluso cuando se pasa un stub. Si la JVM a la que se pasa el stub no tiene su definición, se obtendrá un error.

3.5.2 EJB y RMI

¿Cómo usa la arquitectura EJB la tecnología RMI? Se comenzará por presentar la siguiente figura, que muestra la diferencia fundamental entre EJB y RMI.

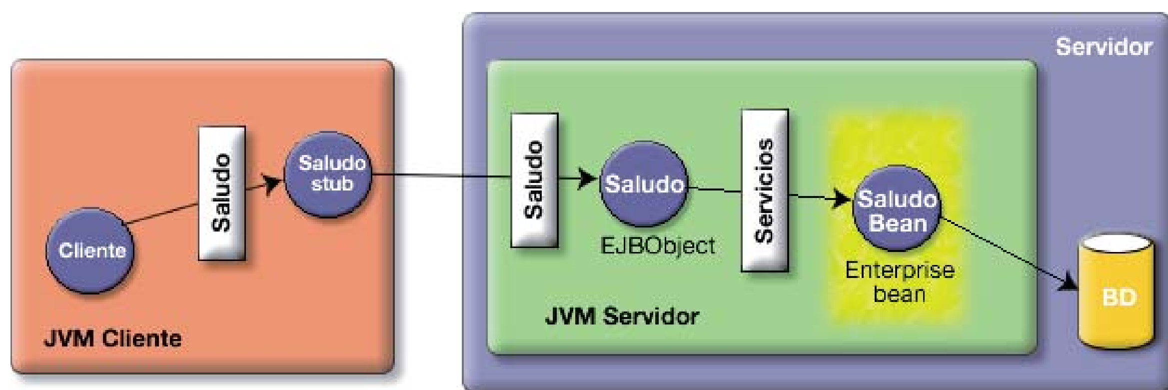

Figura 13: Diferencia entre EJB y RMI

En el lado del cliente todo es igual. El cliente sigue comunicándose con un stub que implementa la interfaz Saludo. Sin embargo, los cambios se encuentran en el lado del servidor. El objeto remoto (Saludo EJBObject) no tiene la implementación realizada por el programador, sino que ésta se encuentra en el llamado objeto Bean (SaludoBean). El objeto remoto hace de "cortafuegos" que separa el bean de los clientes y permite intercalar las llamadas al contenedor en las peticiones de los clientes. Así es posible incorporar los servicios de transacciones, seguridad, etc., proporcionados por el servidor.

Además del objeto remoto que implementa la interfaz del bean (llamada interfaz componente), la arquitectura EJB proporciona otro objeto remoto llamado objeto home que hace el papel de factoría del bean. Este objeto home (SaludoHome, en el ejemplo) proporciona al cliente métodos remotos para crear y localizar instancias de beans. La siguiente figura proporciona una imagen completa de ambas características:

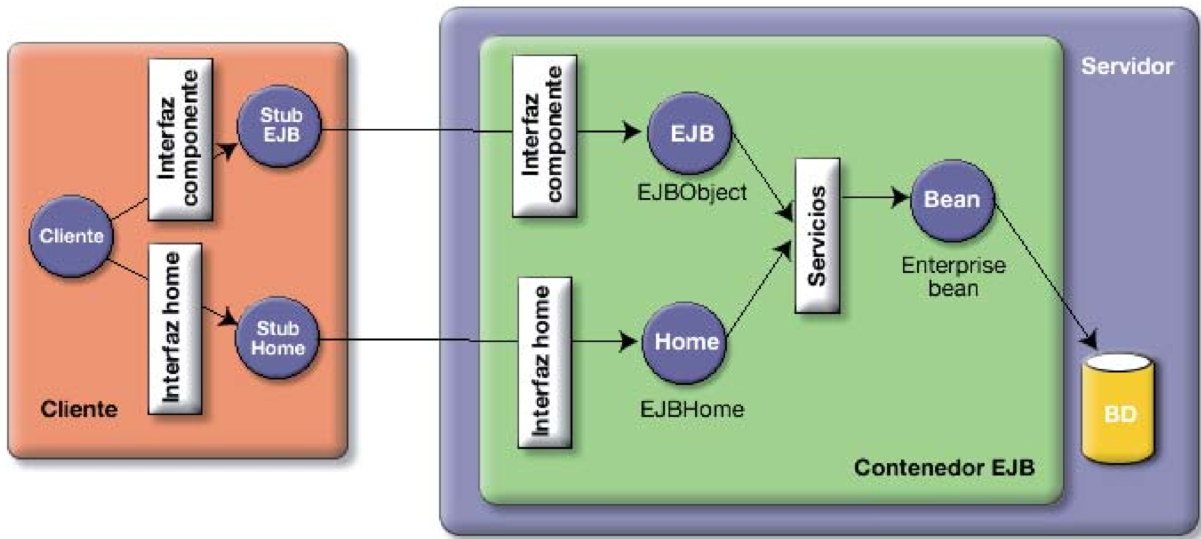

Figura 14: Métodos remotos del objeto home

Se verá a continuación con detalle las clases que se deben implementar para definir un enterprise bean, así como las interfaces que extienden:

- La clase bean SaludoBean que implementa los métodos de negocio y que debe heredar de la clase javax.bean.SessionBean.
- La interfaz componente Saludo que define los métodos accesibles por el cliente y que debe heredar de la interfaz javax.ejb.EJBObject.
- La interfaz home SaludoHome que define los métodos de creación de beans y que debe heredar de la interfaz javax.ejb.EJBHome.

A su vez, las interfaces EJBObject y EJBHome heredan de la interfaz RMI Remote. Esto hace que las interfaces Saludo y SaludoHome sean también remotas. Sin embargo, ni la clase SaludoBean ni la clase SessionBean implementan la interfaz Remote. Esto es muy importante para entender la arquitectura de EJB, la clase bean no es remota. Recordar que el objetivo principal de la estructura de clases e interfaces de EJB es evitar que el cliente llame directamente al bean. Existe siempre un objeto intermedio, el EJBObject, que intercepta las llamadas de los clientes y realiza todos los servicios que proporciona el servidor de aplicaciones.

La siguiente figura representa la estructura de clases del ejemplo que se ha visto.

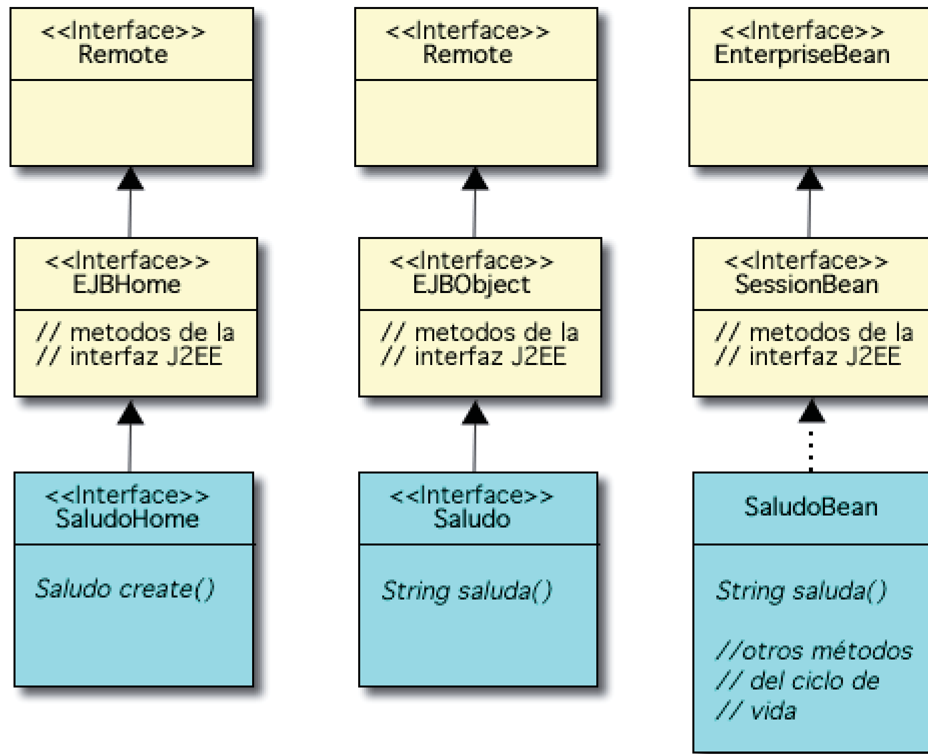

Figura 15: Estructura de Clases del ejemplo

¿Quién crea las clases que implementan todas estas interfaces de las que se está hablando? Evidentemente, como ya se ha dicho, esa es tarea del contenedor EJB. La siguiente tabla resume quién escribe cada una de las clases de un bean.

Tabla 3: Descripción de Clases de un Bean

Programador	**Contenedor de aplicaciones**
La interfaz componente que hereda de javax.ejb.EJBObject. (Saludo.java)	La clase EJBObject que implementa la interfaz componente. (Objeto remoto SaludoEJB.java)
	La clase stub EJBObject que implementa la interfaz componente y sirve para comunicarse por la red con la clase EJBObject anterior. (SaludoStub.java)
La interfaz home que hereda de javax.ejb.EJBHome. (SaludoHome.java)	La clase EJBHome que implementa la interfaz home. (SaludoHome.java)
	La clase stub EJBHome que implementa la interfaz home y sirve para comunicarse por la red con la clase EJBHome anterior. (SaludoHomeStub.java)
La clase bean que implementa javax.ejb.SessionBean o javax.ejb.EntityBean. (SaludoBean.java)	

3.5.3 Interfaces locales y remotas

Hasta ahora se ha considerado siempre que el cliente y el bean se encuentran en distintas máquinas virtuales. Por eso es necesario RMI para conectar el cliente y el EJBObject. Pero ¿por qué introducir RMI cuando el cliente y el EJBObject se encuentran en la misma JVM? La introducción de RMI en la arquitectura la dota de flexibilidad, pero también le añade penalización en el rendimiento, debido sobre todo a la necesidad de serializar todos los argumentos y llamadas. Esta penalización no está justificada cuando el objeto cliente del bean es, por ejemplo, un servlet o una página JSP que reside en la misma JVM que el bean. Tampoco está justificado el uso de RMI cuando se están comunicando dos beans que residen en la misma JVM.

La especificación 2.0 de EJB propone el uso de las interfaces locales EJBLocalHome y EJBLocalObject como una solución a estas situaciones. A la hora de programar el bean lo único que cambia es que las interfaces home y componente deben heredar de EJBLocalHome y EJBLocalObject. Estas interfaces ya no son remotas, por lo que los métodos no van a tener que declarar la excepción RemoteException. En el nombre de las interfaces suelen añadirse la palabra local, para indicar que se tratan de objetos que van a llamarse sin usar RMI. En el caso del bean Saludo, llamaríamos SaludoLocal a la interfaz componente y SaludoLocalHome a la interfaz home.

```
package moduloEjb;

import javax.ejb.*;

public interface SaludoLocal extends EJBLocalObject {
   public String saluda();
}
package moduloEjb;
import javax.ejb.*;

public interface SaludoHomeLocal extends EJBLocalHome {
   public SaludoLocal create() throws CreateException;
}
```

En la arquitectura EJB, la única diferencia es que las llamadas entre el cliente y el EJBObject serían llamadas normales entre objetos Java y no tendrían que ser serializadas. Pero el objeto EJBObject seguiría haciendo las mismas funciones de proteger a los beans del exterior y de incorporar los servicios del contenedor EJB.

Por último, en el lado del cliente existen dos detalles que hay que modificar cuando se están utilizando interfaces locales. En primer lugar, las interfaces locales no declaran la excepción RemoteException, por lo que no es necesario que el cliente capture estas excepciones con un try/catch. El segundo aspecto tiene que ver con el casting del objeto home que obtenemos de JNDI. En el caso de que el objeto home sea remoto, como hemos visto en la sesión anterior que antes de hacer el casting había que convertirlo en un objeto Java con la llamada al método PotableRemoteObject.narrow(). En el caso de estar trabajando con interfaces locales no es necesario hacer esta conversión con lo que para hacer el casting bastaría con hacer:

```
Object ref = jndiContext.lookup("SaludoEJB");
SaludoHome home = (SaludoHome) ref;
```

Terminando con unas breves consideraciones que pueden ayudar a determinar si conviene usar interfaces locales o remotas:

- El funcionamiento de los beans usando interfaces locales es más eficiente ya que todas las llamadas al interfaz componente se realiza en la misma JVM y los parámetros no tienen que ser copiados ni serializados.
- El uso de interfaces locales modifica la semántica de la aplicación, ya que hace que los objetos que se pasan en las llamadas a los beans se pasen por referencia en lugar de por valor. Una modificación de ese objeto en la implementación del método del bean tendría un efecto lateral en el cliente. Estos efectos laterales son los responsables de multitud de bugs muy difíciles de localizar.
- El funcionamiento de las interfaces locales sólo es posible cuando los clientes de los beans van a estar en la misma JVM que los beans. Por ello, sólo podríamos aplicarlas a llamadas a los beans desde servlets o páginas JSP, pero no desde clientes independientes implementados con Swing o desde aplicaciones independientes para realizar tareas de integración.
- El uso de interfaces locales va a hacer mucho más rígida la arquitectura de la aplicación, ya que obliga a que los beans residan en la misma máquina. El uso de interfaces remotas permite una aplicación distribuida en distintas máquinas, lo que facilita la posible aplicación de técnicas de escalabilidad de la aplicación basadas en el clustering de contenedores EJB.
- El uso de interfaces locales permite en los beans de entidad usar relaciones entre beans gestionadas por el contenedor, cosa que no es posible cuando se usan interfaces remotas.

3.5.4 Funcionamiento de la clase home

La clase home de un bean define los métodos de creación de instancias del bean. Cuando se despliega un bean en el contenedor EJB, se crea automáticamente una única instancia de la clase home del bean desplegado. Esta instancia reside en el contenedor EJB y queda a la espera de recibir peticiones de los clientes para crear instancias del bean.

El funcionamiento completo de la clase home se podría resumir con la siguiente figura y los siguientes pasos (para simplificar se usará como ejemplo el bean de sesión sin estado SaludoBean que vimos en la sesión anterior).

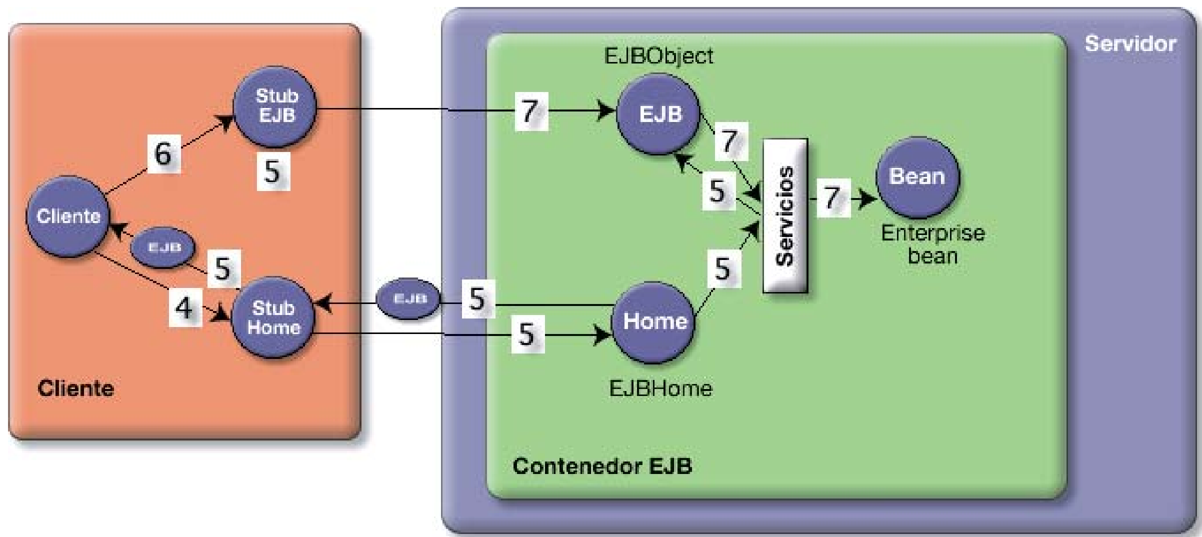

Figura 16: Funcionamiento de la clase home

1. Desplegando en el contenedor EJB el bean SaludoBean. Inmendiatamente, el contenedor crea una instancia de SaludoHome y la registra en JNDI con el nombre "SaludoEJB". Ésta va a ser la única instancia de objeto Home del bean SaludoBean.
2. El cliente realiza un lookup en JNDI buscando el objeto home con el nombre "SaludoEJB".
3. JNDI devuelve el stub del objeto Home y el cliente lo instala en su JVM. Este stub define la interfaz home del bean que el cliente puede usar para crear nuevas instancias del bean.
4. El cliente llama al método create() del stub para que el objeto Home realice una instanciación del bean y devuelva un stub de la instancia del bean SaludoEJB recién creada. Este stub será un stub del EJBObject que intercepta las llamadas al bean.
5. El objeto Home recibe la petición y le pide al contenedor EJB que inicialice un nuevo bean. Se crea un objeto de la clase SaludoBean y se crea su correspondiente EJBObject que intercepta las llamadas de los clientes. Por último, se devuelve el stub del EJBObject al cliente.
6. Ahora ya por fin el cliente puede realizar una petición al bean Saludo. Ya puede usar la interfaz componente del bean, porque el stub devuelto por el objeto home implementa esta interfaz. Por ejemplo, puede ejecutar el método Saluda() del stub.
7. Se realiza una llamada a un método de la interfaz componente. Esto es, el stub transfiere por la red todos los datos al EJBObject, éste realiza todas las comprobaciones necesarias y, por último, transfiere la petición al objeto bean que está representando.

Esta descripción del funcionamiento de la clase home, como se ha dicho, contempla únicamente el caso en de los beans de sesión sin estado. El funcionamiento es distinto según el tipo de bean. Cuando se vea el ciclo de vida de cada uno de los tipos de bean se explicará cómo funciona la clase home en el resto de casos.

3.6 Tipos de Enterprise JavaBeans

La tecnología EJB define tres tipos de beans: beans de sesión, beans de entidad y beans dirigidos por mensajes.

Los beans de entidad representan un objeto concreto que tiene existencia en alguna base de datos de la empresa. Una instancia de un bean de entidad representa una fila en una tabla de la base de datos. Por ejemplo, se podría considerar el bean Cliente, con una instancia del bean siendo Eva Martínez (ID# 342) y otra instancia Francisco Gómez (ID# 120).

Los beans dirigidos por mensajes pueden escuchar mensajes de un servicio de mensajes JMS. Los clientes de estos beans nunca los llaman directamente, sino que es necesario enviar un mensaje JMS para comunicarse con ellos. Los beans dirigidos por mensajes no necesitan objetos EJBObject porque los clientes no se comunican nunca con ellos directamente. Un ejemplo de bean dirigido por mensajes podría ser un bean ListenerNuevoCliente que se activara cada vez que se envía un mensaje comunicando que se ha dado de alta a un nuevo cliente.

Por último, un bean de sesión representa un proceso o una acción de negocio. Normalmente, cualquier llamada a un servicio del servidor debería comenzar con una llamada a un bean de sesión. Mientras que un bean de entidad representa una cosa que se puede representar con un nombre, al pensar en un bean de sesión deberías pensar en un verbo. Ejemplos de beans de sesión podrían ser un carrito de la compra de una aplicación de negocio electrónico o un sistema verificador de tarjetas de crédito.

3.6.1 Beans de sesión

Los beans de sesión representan sesiones interactivas con uno o más clientes. Los bean de sesión pueden mantener un estado, pero sólo durante el tiempo que el cliente interactúa con el bean. Esto significa que los beans de sesión no almacenan sus datos en una base de datos después de que el cliente termine el proceso. Por ello se suele decir que los beans de sesión no son persistentes.

A diferencia de los bean de entidad, los beans de sesión no se comparten entre más de un cliente, sino que existe una correspondencia uno-uno entre beans de sesión y clientes. Por esto, el contenedor EJB no necesita implementar mecanismos de manejo de concurrencia en el acceso a estos beans.

Existen dos tipos de beans de sesión: con estado y sin él.

3.6.1.1 Beans de sesión sin estado

Los beans de sesión sin estado no se modifican con las llamadas de los clientes. Los métodos que ponen a disposición de las aplicaciones clientes son llamadas que reciben datos y devuelven resultados, pero que no modifican internamente el estado del bean. Esta propiedad permite que el contenedor EJB pueda crear una reserva (pool) de instancias, todas ellas del mismo bean de sesión sin estado y asignar cualquier instancia a cualquier cliente. Incluso un único bean puede estar asignado a múltiples clientes, ya que la asignación sólo dura el tiempo de invocación del método solicitado por el cliente.

Una de las ventajas del uso de beans de sesión, frente al uso de clases Java u objetos RMI es que no es necesario escribir los métodos de los beans de sesión de una forma segura para threads (thread-safe), ya que el contenedor EJB se va a encargar de que nunca haya más de un thread accediendo al objeto. Para ello usa múltiples instancias del bean para responder a peticiones de los clientes.

Cuando un cliente invoca un método de un bean de sesión sin estado, el contenedor EJB obtiene una instancia de la reserva. Cualquier instancia servirá, ya que el bean no puede guardar ninguna información referida al cliente. Tan pronto como el método termina su ejecución, la instancia del bean está disponible para otros clientes. Esta propiedad hace que los beans de sesión sin estado sean muy escalables para un gran número de clientes. El contenedor EJB no tiene que mover sesiones de la memoria a un almacenamiento secundario para liberar recursos, simplemente puede obtener recursos y memoria destruyendo las instancias.

Los beans de sesión sin estado se usan en general para encapsular procesos de negocio, más que datos de negocio (tarea de los entity beans). Estos beans suelen recibir nombres como ServicioBroker o GestorContratos para dejar claro que proporcionan un conjunto de procesos relacionados con un dominio específico del negocio.

Es apropiado usar beans de sesión sin estado cuando una tarea no está ligada a un cliente específico. Por ejemplo, se podría usar un bean sin estado para enviar un e-mail que confirme un pedido on-line o calcular unas cuotas de un préstamo.

También puede usarse un bean de sesión sin estado como un puente de acceso a una base de datos o a un bean de entidad. En una arquitectura cliente-servidor, el bean de sesión podría proporcionar al interfaz de usuario del cliente los datos necesarios, así como modificar objetos de negocio (base de datos o bean de entidad) a petición de la interfaz. Este uso de los beans de sesión sin estado es muy frecuente y constituye el denominado patrón de diseño session facade. Algunos ejemplos de bean de sesión sin estado podrían ser:

- Un componente que comprueba si un símbolo de compañía está disponible en el mercado de valores y devuelve la última cotización registrada.
- Un componente que calcula la cuota del seguro de un cliente, basándose en los datos que se le pasa del cliente.

3.6.1.2 Beans de sesión con estado

En un bean de sesión con estado, las variables de instancia del bean almacenan datos específicos obtenidos durante la conexión con el cliente. Cada bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el bean. Este estado conversacional se modifica conforme el cliente va realizando llamadas a los métodos de negocio del bean. El estado conversacional no se guarda cuando el cliente termina la sesión.

La interacción del cliente con el bean se divide en un conjunto de pasos. En cada paso se añade nueva información al estado del bean. Cada paso de interacción suele denominarse

con nombres como setNombre o setDireccion, siendo nombre y dirección dos variables de instancia del bean. Algunos ejemplos de beans de sesión con estado podrían ser:

- Un ejemplo típico es un carrito de compra, en donde el cliente va guardando uno a uno los ítems que va comprando.
- Un enterprise bean que reserva un vuelo y alquila un coche en un sitio Web de una agencia de viajes.
-

El estado del bean persiste mientras que existe el bean. A diferencia de los beans de entidad, no existe ningún recurso exterior al contenedor EJB en el que se almacene este estado.

Debido a que el bean guarda el estado conversacional con un cliente determinado, no le es posible al contenedor crear un almacén de beans y compartirlos entre muchos clientes. Por ello, el manejo de beans de sesión con estado es más pesado que el de beans de sesión sin estado. En general, se debería usar un bean de sesión con estado si se cumplen las siguientes circunstancias:

- El estado del bean representa la interacción entre el bean y un cliente específico.
- El bean necesita mantener información del cliente a lo largo de un conjunto de invocaciones de métodos.
- El bean hace de intermediario entre el cliente y otros componentes de la aplicación, presentando una vista simplificada al cliente.

3.6.2 Beans de entidad

Los beans de entidad modelan conceptos o datos de negocio que puede expresarse como nombres. Esto es una regla sencilla más que un requisito formal, pero ayuda a determinar cuándo un concepto de negocio puede ser implementado como un bean de entidad. Los beans de entidad representan "cosas", objetos del mundo real como hoteles, habitaciones, expedientes, estudiantes, y demás. Un bean de entidad puede representar incluso cosas abstractas como una reserva. Los beans de entidad describen tanto el estado como la conducta de objetos del mundo real y permiten a los desarrolladores encapsular las reglas de datos y de negocio asociadas con un concepto específico. Por ejemplo un bean de entidad Estudiante encapsula los datos y reglas de negocio asociadas a un estudiante. Esto hace posible manejar de forma consistente y segura los datos asociados a un concepto.

Los beans de entidad se corresponden con datos en un almacenamiento persistente (base de datos, sistema de ficheros, etc.). Las variables de instancia del bean representan los datos en las columnas de la base de datos. El contenedor debe sincronizar las variables de instancia del bean con la base de datos. Los beans de entidad se diferencian de los beans de sesión en que las variables de instancia se almacenan de forma persistente.

Aunque entraremos en detalle más adelante, es interesante adelantar que el uso de los beans de entidad desde un cliente conlleva los siguientes pasos:

- Primero el cliente debe obtener una referencia a la instancia concreta del bean de entidad que se está buscando (el estudiante "Francisco López") mediante un método

finder. Estos métodos finder se encuentran definidos en la interfaz home e implementados en la clase bean. Los métodos finder pueden devolver uno o varios beans de entidad.

- El cliente interactúa con la instancia del bean usando sus métodos get y set. El estado del bean se carga de la base de datos antes de procesar las llamadas a los métodos. Esto se encarga de hacerlo el contenedor de forma automática o el propio bean en la función ejbLoad().
- Por último, cuando el cliente termina la interacción con la instancia del bean sus contenidos se vuelcan en el almacén persistente. O bien lo hace de forma automática el contenedor o bien éste llama al método ejbStore().

Son muchas las ventajas de usar beans de entidad en lugar de acceder a la base de datos directamente. El uso de beans de entidad nos da una perspectiva orientada a objetos de los datos y proporciona a los programadores un mecanismo más simple para acceder y modificar los datos. Es mucho más fácil, por ejemplo, cambiar el nombre de un estudiante llamando a student.setName() que ejecutando un comando SQL contra la base de datos. Además, el uso de objetos favorece la reutilización del software. Una vez que un bean de entidad se ha definido, su definición puede usarse a lo largo de todo el sistema de forma consistente. Un bean Estudiante proporciona un forma completa de acceder a la información del estudiante y eso asegura que el acceso a la información es consistente y simple.

La representación de los datos como beans de entidad puede hacer que el desarrollo sea más sencillo y menos costoso.

3.6.2.1 Diferencias con los beans de sesión

Los beans de entidad se diferencian de los beans de sesión, principalmente, en que son persistentes, permiten el acceso compartido, tienen clave primaria y pueden participar en relaciones con otros beans de entidad:

Persistencia

Debido a que un bean de entidad se guarda en un mecanismo de almacenamiento se dice que es persistente. Persistente significa que el estado del bean de entidad existe más tiempo que la duración de la aplicación o del proceso del servidor J2EE. Un ejemplo de datos persistentes son los datos que se almacenan en una base de datos.

Los beans de entidad tienen dos tipos de persistencia: Persistencia Gestionada por el Bean (BMP, Bean-Managed Persistence) y Persistencia Gestionada por el Contenedor (CMP, Container-Managed Persistence). En el primer caso (BMP) el bean de entidad contiene el código que accede a la base de datos. En el segundo caso (CMP) la relación entre las columnas de la base de datos y el bean se describe en el fichero de propiedades del bean, y el contenedor EJB se ocupa de la implementación.

Acceso compartido

Los clientes pueden compartir beans de entidad, con lo que el contenedor EJB debe gestionar el acceso concurrente a los mismos y por ello debe usar transacciones. La forma de hacerlo dependerá de la política que se especifique en los descriptores del bean.

Clave primaria

Cada bean de entidad tiene un identificador único. Un bean de entidad alumno, por ejemplo, puede identificarse por su número de expediente. Este identificador único, o clave primaria, permite al cliente localizar a un bean de entidad particular.

Relaciones

De la misma forma que una tabla en una base de datos relacional, un bean de entidad puede estar relacionado con otros EJB. Por ejemplo, en una aplicación de gestión administrativa de una universidad, el bean alumnoEjb y el bean actaEjb estarían relacionados porque un alumno aparece en un acta con una calificación determinada.

Las relaciones se implementan de forma distinta según se esté usando la persistencia manejada por el bean o por el contenedor. En el primer caso, al igual que la persistencia, el desarrollador debe programar y gestionar las relaciones. En el segundo caso es el contenedor el que se hace cargo de la gestión de las relaciones. Por ello, estas últimas se denominan a veces relaciones gestionadas por el contenedor.

3.6.3 Beans dirigidos por mensajes

Son el tercer tipo de beans propuestos por la última especificación de EJB. Estos beans permiten que las aplicaciones J2EE reciban mensajes JMS de forma asíncrona. Así, el hilo de ejecución de un cliente no se bloquea cuando está esperando que se complete algún método de negocio de otro enterprise bean. Los mensajes pueden enviarse desde cualquier componente J2EE (una aplicación cliente, otro enterprise bean, o un componente Web) o por una aplicación o sistema JMS que no use la tecnología J2EE.

3.6.3.1 Diferencias con los beans de sesión y de entidad

La diferencia más visible es que los clientes no acceden a los beans dirigidos por mensajes mediante interfaces, sino que un bean dirigido por mensajes sólo tienen una clase bean.

En muchos aspectos, un bean dirigido por mensajes es parecido a un bean de sesión sin estado.

- Las instancias de un bean dirigido por mensajes no almacenan ningún estado conversacional ni datos de clientes.
- Todas las instancias de los beans dirigidos por mensajes son equivalentes, lo que permite al contenedor EJB asignar un mensaje a cualquier instancia. El contenedor puede almacenar estas instancias para permitir que los streams de mensajes sean procesados de forma concurrente.
- Un único bean dirigido por mensajes puede procesar mensajes de múltiples clientes.

Las variables de instancia de estos beans pueden contener algún estado referido al manejo de los mensajes de los clientes. Por ejemplo, pueden contener una conexión JMS, una conexión de base de datos o una referencia a un objeto enterprise bean.

Cuando llega un mensaje, el contenedor llama al método onMessage del bean. El método onMessage suele realizar un casting del mensaje a uno de los cinco tipos de mensajes de JMS y manejarlo de forma acorde con la lógica de negocio de la aplicación. El método onMessage puede llamar a métodos auxiliares, o puede invocar a un bean de sesión o de entidad para procesar la información del mensaje o para almacenarlo en una base de datos.

Un mensaje puede enviarse a un bean dirigido por mensajes dentro de un contexto de transacción, por lo que todas las operaciones dentro del método onMessage son parten de un única transacción.

3.7 La arquitectura de los distintos tipos de beans

Hasta ahora se ha simplificado la descripción de la arquitectura EJB suponiendo que se tiene un único cliente y centrando la discusión en el acceso de los clientes a los beans.

Se va a ampliar en este punto la explicación de la arquitectura EJB, centrándose en la gestión de la concurrencia de acceso de los clientes y en el proceso de creación de los beans. Se verá que la gestión de la concurrencia es distinta según tengamos un bean de sesión con estado, un bean de sesión sin estado, un bean de entidad o un bean dirigido por mensajes. Lo mismo sucede con el proceso de creación de los beans y su ciclo de vida en el contenedor EJB.

Todas estas consideraciones hacen que cada tipo de bean tenga unas características propias, en cuanto a eficiencia, escalabilidad, tiempo de respuesta, etc. Es muy importante conocer estas características, ya que van a incidir directamente en el rendimiento de la aplicación que estés diseñando. Cuando se vea el ciclo de vida de cada uno de los tipos de beans se harán más consideraciones que también afectarán al rendimiento de cada tipo de bean.

3.7.1 Beans de sesión sin estado

Se comenzará por recordar que los beans de sesión sin estado (por ejemplo, el bean SaludoEJB) proporcionan al cliente un conjunto de métodos remotos que representan tareas que el cliente puede solicitar. No existe ningún estado que deba mantener el bean: el cliente realiza la petición al EJBObject, el EJBObject pasa la petición al bean, el bean realiza la operación y devuelve la respuesta al cliente.

La simplicidad del funcionamiento de este tipo de beans hace posible que sean muy escalables y que tengan un rendimiento muy bueno. El contenedor puede utilizar bastantes técnicas para optimizar su rendimiento, como son el mantener una reserva (pool) de beans, el reutilizar una instancia de un bean para distintos clientes o el usar un único EJBObject para varios clientes.

La siguiente figura muestra un ejemplo en el que más de un cliente está solicitando servicios de un bean de sesión sin estado.

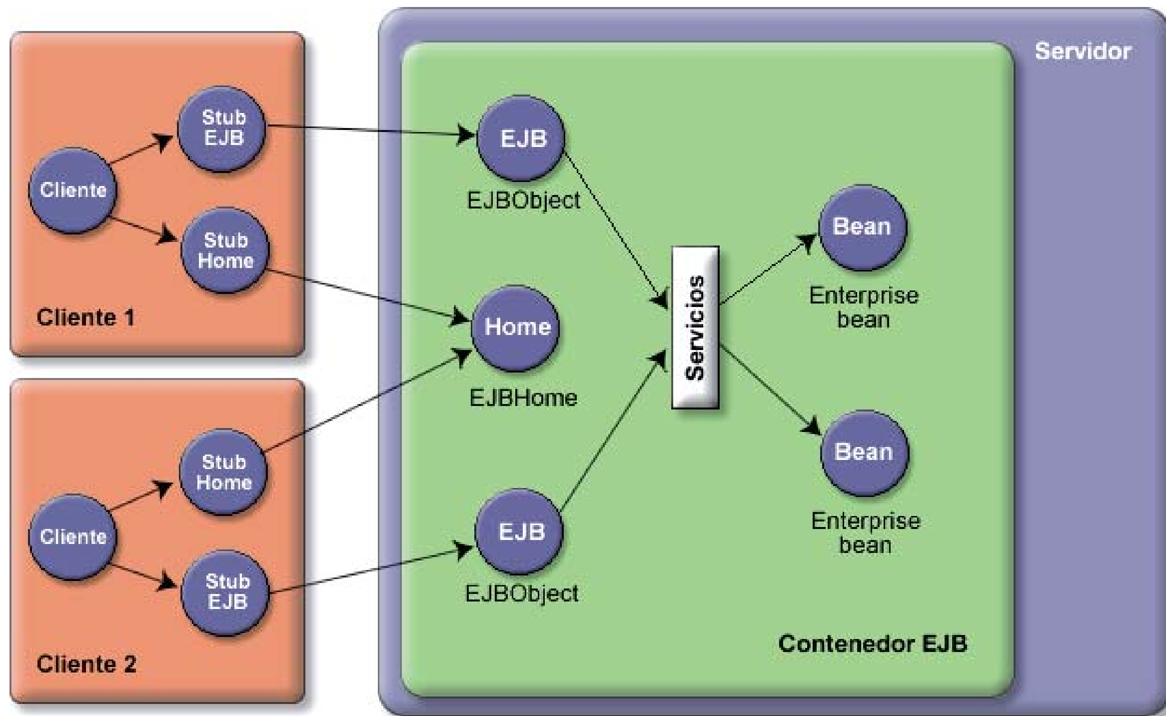

Figura 17: Solicitud de clientes de servicios a un bean de sesión sin estado

¿Cuándo crea el contenedor nuevas instancias de beans? Cuando lo considera necesario para mantener el rendimiento del servicio a los clientes. Puede ser que en un momento dado haya una avalancha de peticiones concurrentes y sea necesario aumentar el número de beans del pool de beans. Eso sí, como ya ha quedado claro, el método create() no hace que se cree un objeto nuevo. Se sabe que en Java las operaciones más costosas tienen que ver con la creación y desaparición de objetos y con la posible puesta en marcha del recolector de basura que ello conlleva.

Se verá el detalle del ciclo de vida del contenedor EJB durante el proceso de creación y uso de los beans de sesión sin estado.

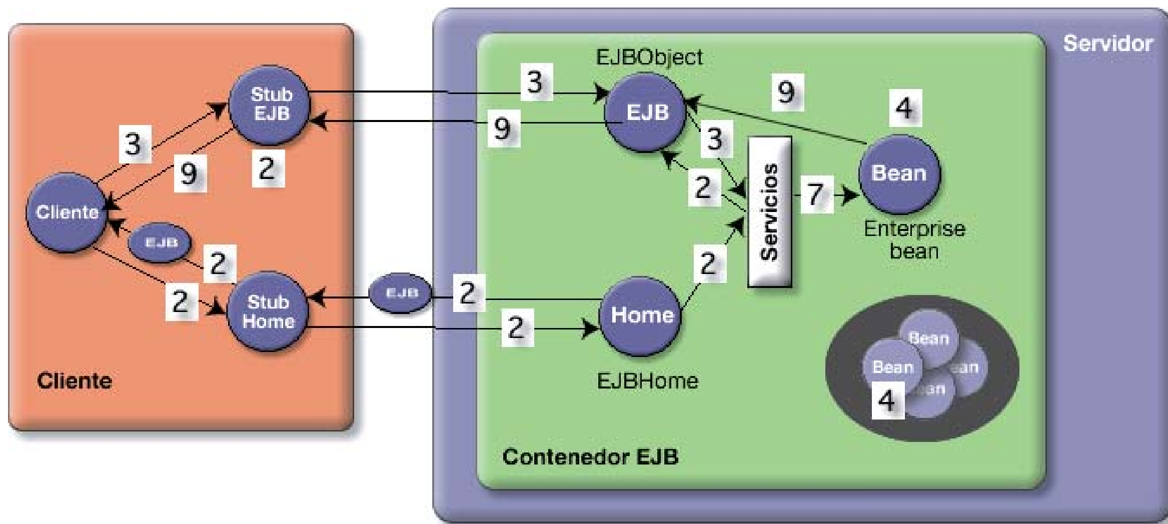

Figura 18: Ciclo de vida del contenedor EJB

1. El cliente obtiene una referencia al objeto home.
2. El cliente usa el método create() de la interfaz home para crear una instancia de un objeto EJBObject cuya referencia (su stub) se pasa al cliente.
3. El cliente invoca un método de negocio en el objeto EJBObject.
4. El contenedor reusa una instancia del bean existente en la reserva, si existe, o crea una nueva instancia del bean e inicializa su contexto llamando a setSessionContext().
5. Si la instancia del bean ha sido creada en el paso anterior, el contenedor invoca ejbCreate() en la instancia del bean.
6. El contenedor comienza una transacción, si hay que hacerlo.
7. El contenedor invoca el método de negocio solicitado en la instancia del bean, y el bean realiza la operación solicitada.
8. El contenedor confirma la transacción, si hay que hacerlo.
9. Los resultados de la llamada al método de negocio se devuelven al cliente.
10. El cliente puede invocar métodos adicionales en el objeto EJBObject. El contenedor puede realizar invocar los métodos adicionales en instancias distintas del bean.
11. El cliente llama a remove() en el objeto EJBObject cuando ha terminado de usarlo, como una forma de comunicarle al contenedor EJB que ya no va a necesitar más el bean.
12. En algún momento, el contenedor decide reducir el tamaño de la reserva de beans de sesión e invoca el método ejbRemove() en la instancia del bean que va a eliminar.

3.7.2 Beans de sesión con estado

Los beans de sesión con estado mantienen un estado distinto para cada cliente. Por ello, ya no es posible usar alguna de las optimizaciones que hemos comentado para los beans de sesión sin estado. Siempre hay una relación uno a uno entre clientes y objetos EJBObjects e instancias del bean.

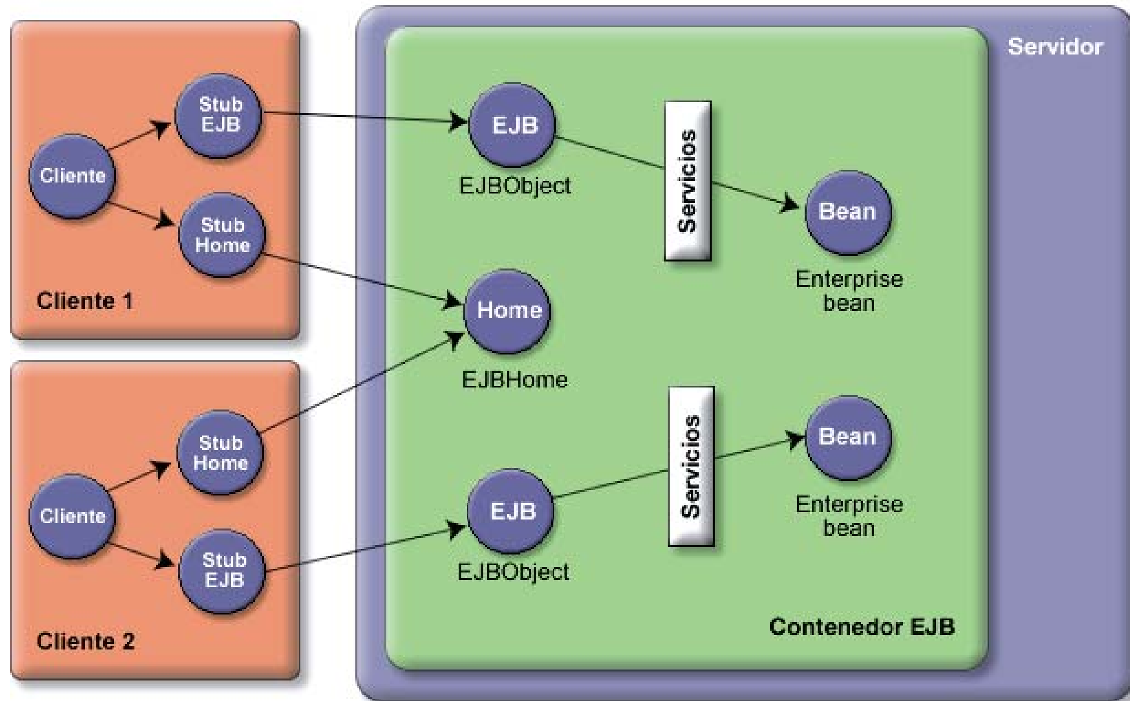

Figura 19: Solicitud de clientes de servicios a un bean de sesión con estado

El proceso de creación de un bean de sesión con estado es idéntico al de un bean de sesión sin estado, con la salvedad de que, al mantenerse en el bean un estado propio de la conexión con el cliente, la instancia del bean asignada al cliente es la que debe ejecutar todas las llamadas. No es posible, como se hacía en el punto 4 del ciclo de vida anterior, invocar a distintas instancias del bean.

3.7.3 Beans de entidad

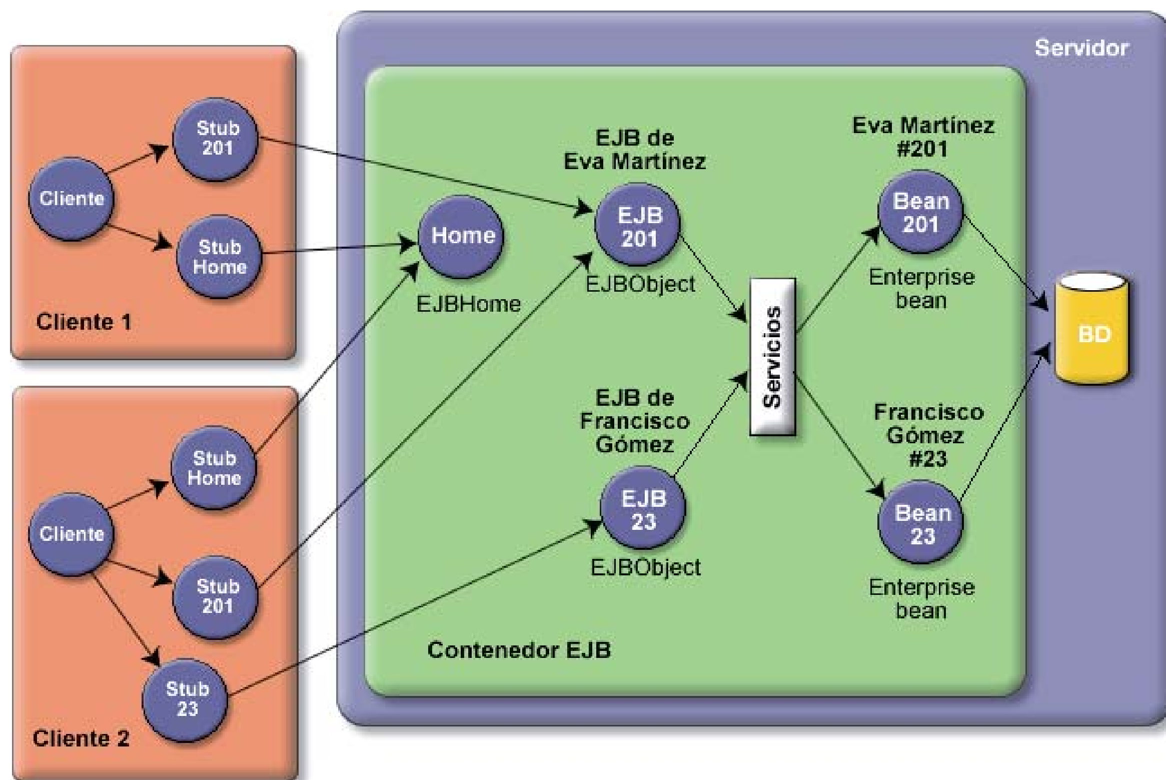

Figura 20: Solicitud de clientes de servicios a un bean de entidad

En los beans de entidad, cada instancia de bean representa un objeto de negocio concreto. Por ejemplo, el estudiante ID213 o el aula ID101. Cada instancia de bean de entidad representa una fila de una o varias tablas (en el caso en que las tablas de la base de datos estén normalizadas y que la información de un objeto esté repartida en más de tabla).

En este caso, es posible que más de un cliente esté accediendo al mismo bean, por lo que el contenedor debe tener esto en cuenta y gestionar la concurrencia. Más adelante se vera qué tipo de estrategias existen para gestionar estos accesos concurrentes de los clientes a los beans de entidad.

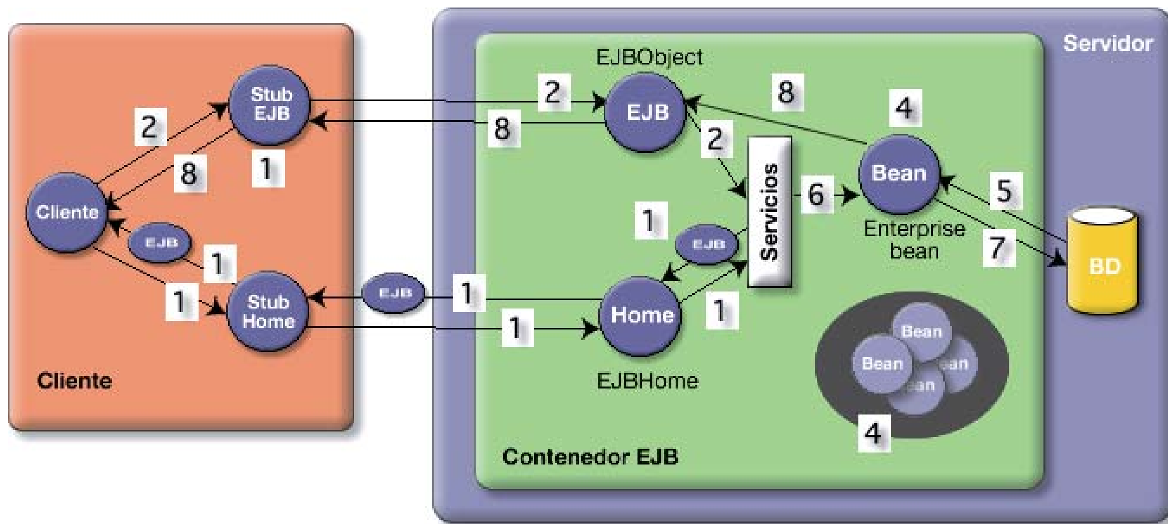

Figura 21: Pasos del ciclo de vida del bean de entidad

En cuanto al ciclo de vida del bean se puede resumir en los siguientes pasos:

1. El cliente obtiene una referencia al bean de entidad de alguna forma, como usando un método finder o recorriendo una relación de un bean.
2. El cliente invoca un método de negocio en el objeto EJBObject. Asumimos que no hay ninguna transacción activa en ese momento.
3. El contenedor comienza una transacción, si hay que hacerlo.
4. El contenedor reutiliza una instancia existente en la reserva, si hay alguna disponible, o instancia una nueva instancia del bean e inicializa su contexto llamando a setEntityContext().
5. El contenedor carga el estado persistente del bean del almacén persistente, o bién automáticamente o bien llamando a ejbLoad().
6. El contenedor invoca el método de negocio deseado por el cliente y el bean realiza la operación.
7. El contenedor guarda el estado del bean en el almacén persistente, o bien automáticamente o bien llamando a ejbStore().
8. El contenedor confirma la transacción, si hay que hacerlo.
9. Los resultados de la llamada al método de negocio se devuelven al cliente.

Esta es una descripción muy simplificada en la que no se contemplan muchas de las técnicas de optimización y caché que se utilizan para mejorar el rendimiento reduciendo el trabajo en cada paso o eliminándolo completamente.

3.7.4 Beans dirigidos por mensajes

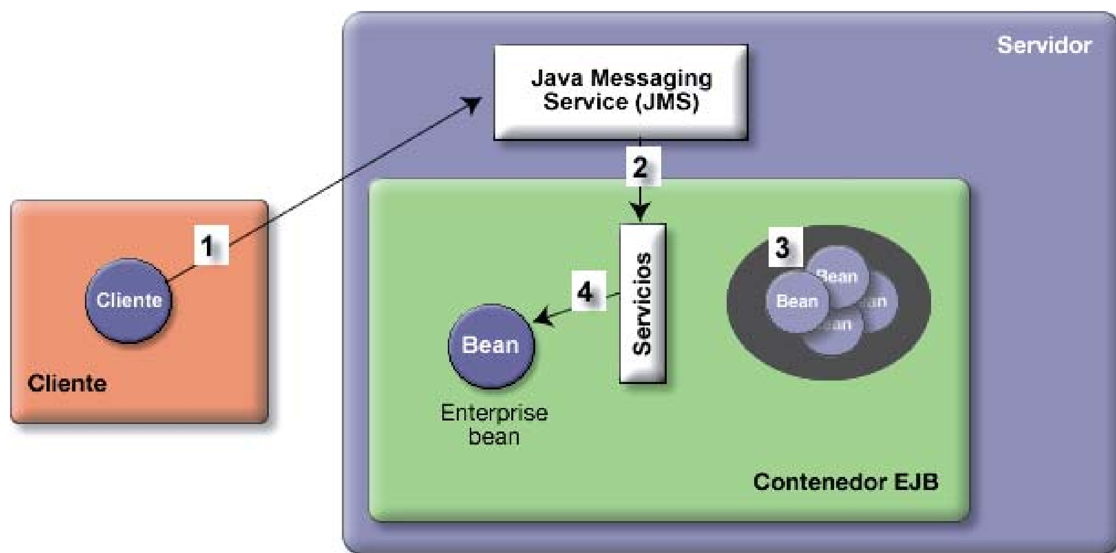

Figura 22: Pasos del ciclo de vida del bean dirigido por mensajes

1. El cliente envía un mensaje al servicio de mensajería JMS
2. El servicio de mensajería envía el mensaje al contenedor
3. El contenedor obtiene un bean dirigido por mensajes de la reserva de beans
4. El contenedor envía el mensaje al bean mediante una llamada al método onMessage().

3.8 Despliegue de aplicaciones y beans

El proceso de despliegue de un bean es algo complejo y es muy común que se produzcan errores en su desarrollo. Para complicar más las cosas, es un proceso que no está estandarizado por J2EE y que depende de cada servidor de aplicaciones.

Vamos a intentar explicar los elementos fundamentales que sí contempla J2EE. Se trata de la especificación del contenido que de los distintos tipos de ficheros que intervienen en el despliegue. Son los siguientes:

- Fichero EJB JAR: Fichero de componentes EJB. Fichero JAR que contiene uno o más componentes EJB y un único fichero descriptor del despliegue (ejb-jar.xml)
- Fichero EAR: Fichero de aplicación enterprise. Fichero JAR que contiene uno o más ficheros EJB JAR y uno o más ficheros WAR (que contienen aplicaciones web) y un descriptor de despliegue (application.xml)

A continuación se mostrará el contenido de estos ficheros para el ejemplo de aplicación enterprise en la que se usa el bean SaludoEjb.

Fichero EJB JAR (podría llamarse saludoEjb.jar).

```
/saludoEjb
/saludoEjb/moduloEjb
/saludoEjb/moduloEjb/SaludoBean.class
/saludoEjb/moduloEjb/Saludo.class
/saludoEjb/moduloEjb/SaludoHome.class
/saludoEjb/META-INF
/saludoEjb/META-INF/ejb-jar.xml
/saludoEjb/META-INF/weblogic-ejb-jar.xml
```

El directorio moduloEjb es necesario porque todas las clases SaludoBean, Saludo y SaludoHome se incluyen el paquete Java moduloEjb (para evitar solapamiento de nombres con otros posibles beans desarrollados por terceros).

El fichero weblogic-ejb-jar.xml es el descriptor de despliegue específico del servidor de aplicaciones WebLogic. Es muy importante saber que en él se define el nombre JNDI del bean o los beans del fichero EJB JAR.

Fichero EAR (podría llamarse saludoEar.jar):

```
/saludoEar
/saludoEar/saludoEjb.jar
/saludoEar/saludoWar.jar
/saludoEar/META-INF
/saludoEar/META-INF/application.xml
/saludoEar/META-INF/weblogic-application.xml
```

El fichero weblogic-application.xml es el descriptor de despliegue de la aplicación enterprise específico del servidor de aplicaciones WebLogic.

El contenido de los descriptores de despliegue se muestra a continuación:
ejb-jar.xml:

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<!-- Generated XML! -->

<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SaludoBean</ejb-name>
      <home>moduloEjb.SaludoHome</home>
      <remote>moduloEjb.Saludo</remote>
      <ejb-class>moduloEjb.SaludoBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SaludoBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
```

```
  </assembly-descriptor>

</ejb-jar>
```

weblogic-ejb-jar.xml:

```
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 8.1.0
EJB//EN' 'http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd'>

<!-- Generated XML! -->

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>SaludoBean</ejb-name>
    <stateless-session-descriptor>
      <pool>
      </pool>

      <stateless-clustering>
      </stateless-clustering>

    </stateless-session-descriptor>

    <transaction-descriptor>
    </transaction-descriptor>

    <jndi-name>SaludoBean</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

4 Arquitecturas Propuestas

4.1 Arquitectura de Spring propuesta

La arquitectura para el caso de Spring Framework, utilizará Hibernate el cual, se utiliza como motor de persistencia, formando parte de toda la capa de acceso a datos. La capa de negocio y web se implementan mediante Spring, utilizando su modelo MVC para la parte web y su modelo de AOP (Aspect-oriented programming) para la capa de negocio. Adicionalmente se utilizan otros componentes de Spring que se detallarán más adelante.

A continuación se muestra un diagrama de componentes general:

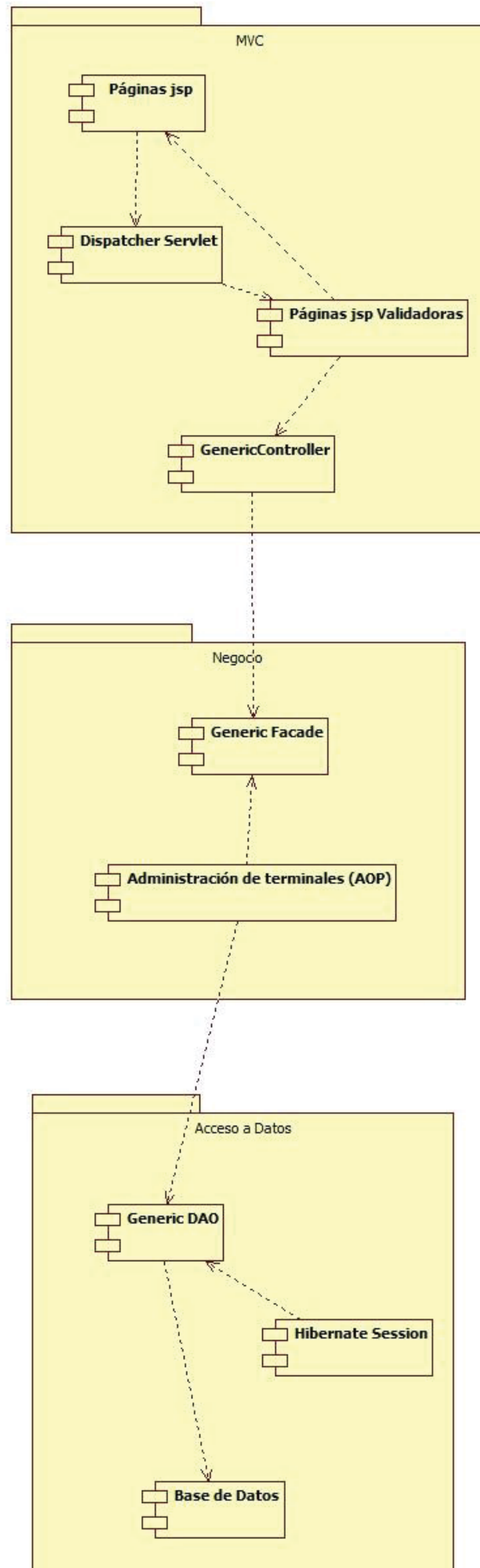

Figura 23: Diagrama de componentes de UML, para la propuesta de Spring

4.1.1 MVC

En la capa web se utilizará el modelo MVC de Spring. El esquema de herencia de los controllers utilizados es el siguiente:

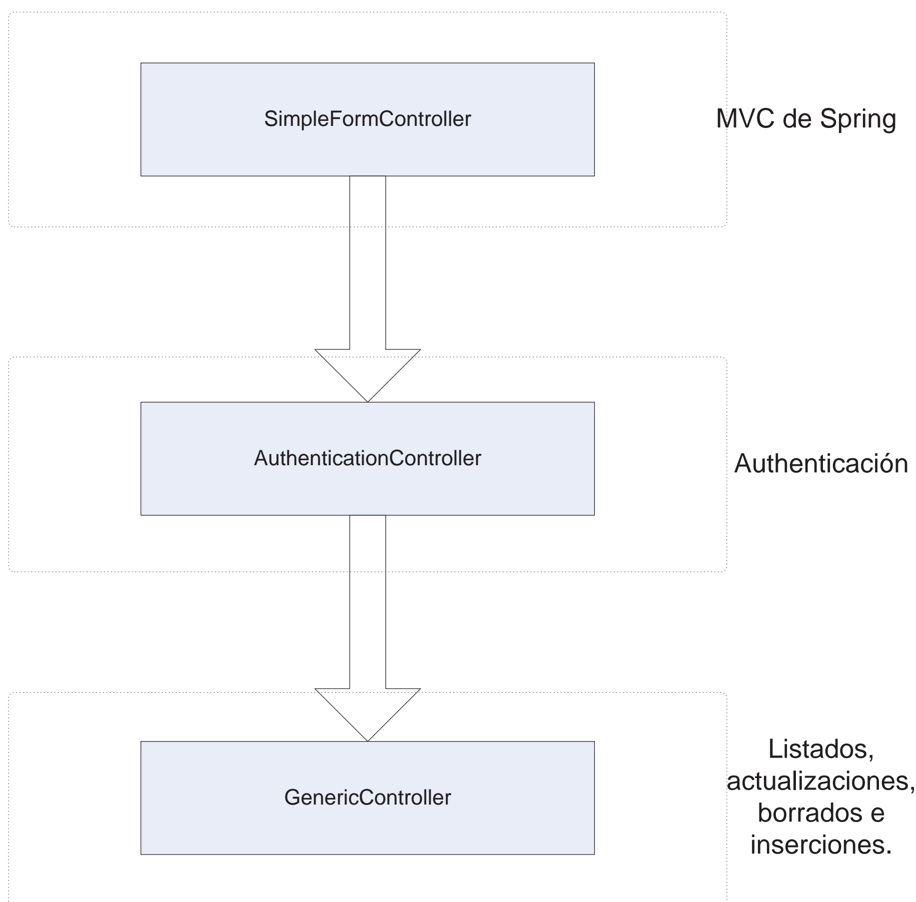

Figura 24: Esquema de herencia de los Controllers

4.1.2 Negocio

En la capa de negocio se utiliza el modelo de AOP de Spring. Se basará en el patrón de diseño[1] SessionFacade[2]. Éstas clases centralizan todo el proceso de negocio de la aplicación, y son envueltos con varios aspectos para proporcionar transaccionalidad automática,

[1] Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño es una solución a un problema de diseño.

[2] El patrón de Fachada permite aislar el acceso a otros EJB's mediante un "Session EJB", por eso su nombre Session Façade. Otra ventaja que otorga el uso "Fechadas de Sesión" es el número de requisiciones necesarias para ser transferida la información entre el Cliente (JSP/Servlet/Applet) y los distintos EJB's.

autorización mediante permisos, y auditoria de operaciones. Para localizar los facades se utiliza el patrón ServiceLocator[3], implementado mediante el container IoC de Spring. La localización de un facade se realiza mediante Injection por Spring. El diagrama de ésta capa es el siguiente:

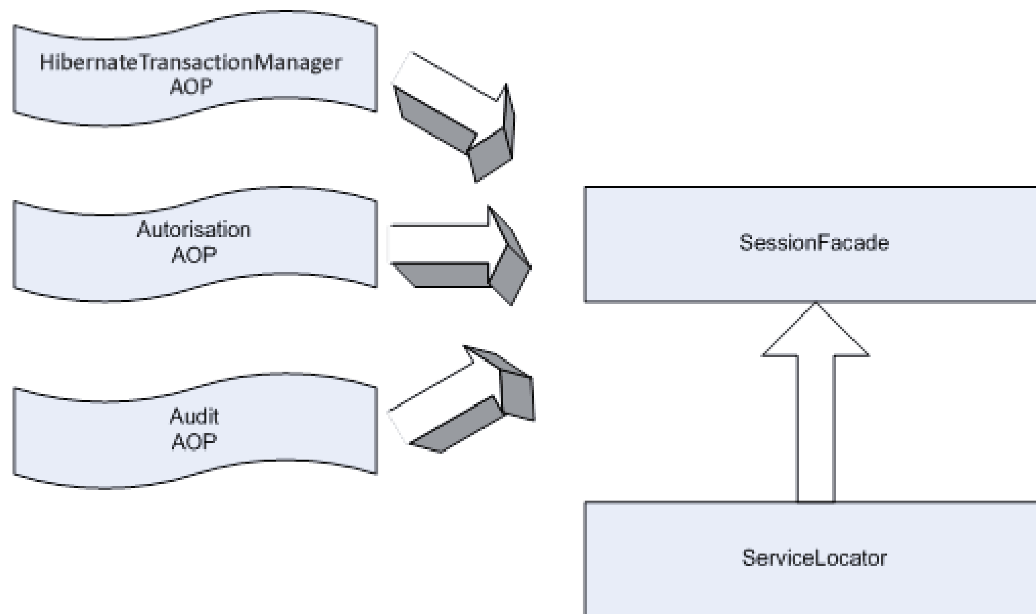

Figura 25: Diagrama de la capa de Negocio

4.1.3 Acceso a datos

El acceso a datos se realiza mediante Hibernate. Spring proporciona un nodo de conexión sencillo para integrarlo sin una configuración excesiva.

Principalmente todo el acceso a datos debería realizarse mediante Hibernate, aunque se puede dejar la puerta abierta a la utilización de JDBC estándar para algún caso en el que sea más sencillo, como por ejemplo, la llamada a procedimientos almacenados.

[3] En más de una ocasión un cliente deberá hacer uso de JNDI ya sea para obtener una conexión a la base de datos, una referencia a la clase home de un enterprise bean, o una referencia a los canales de mensajería. Al ser algo común para muchas componentes, tiende a aparecer código similar múltiples veces y la creación repetida del objeto InitialContext que puede tomar cierto tiempo.

Utilizar un Service Locator permite abstraer todo los usos de JNDI simplificando el código del cliente, creando un único punto de control y mejorando el performance de la aplicación. Para el caso particular, y más utilizado, de obtener objetos home existe el patrón de diseño EJBHomeFactory que consiste en buscar el nombre de la referencia de cierto objeto en los archivos XML de configuración de la aplicación para luego hacer utilizar JNDI para obtenerlo.

4.2 Arquitectura de EJB3 Propuesta

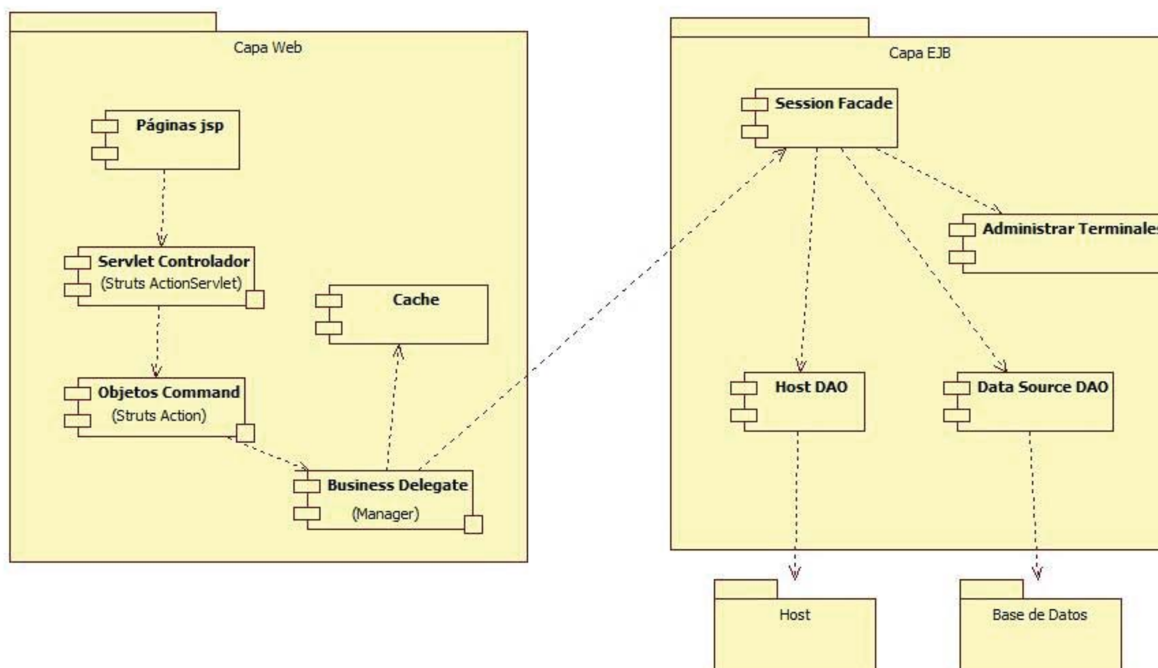

Figura 26: Diagrama de componentes para la arquitectura de EJB

En el diagrama de componentes de UML presentado, podemos ver la nueva arquitectura de EJB, el cual cuenta con 2 partes importantes: Capa Web y Capa EJB.

En la capa Web, tenemos las páginas jsp por la cual el usuario va a ingresar el sistema. Cuando se hace una solicitud por parte del usuario, el Servlet Controlador recibe la petición y después la distribuye a los objetos Command, los cuales a su vez, envían la petición al Business Delegate el cual facilita el acceso de la capa del cliente a la capa EJB, ocupando la API de EJB. Este Manager, al mismo tiempo que envía la petición a la capa EJB, copia en cache la petición, para tenerla de respaldo en el caso de algún problema con la consulta.

En el Session Facade, se controlan los permisos de los usuarios. Si está correcto el acceso, entonces se comienza con la lógica del negocio y se accede a la Base de Datos, a través del Data Source DAO, independientemente del host que haya por detrás, con el DAO, nos despreocupamos de la conexión , cierre de recursos utilizados, etc.

5 Análisis de Resultados

Para realizar la implementación de las arquitecturas propuestas, fue necesario un equipo computacional, el cual tiene las siguientes características:

- Notebook Dell Vostro 1500.
- Procesador Intel Core 2 Duo 2.2 GHZ.
- HDD de 160 Gb y memoria RAM de 2Gb.
- El software utilizado es el siguiente:
    - Windows 7.
    - Spring Framework 2.6

5.1 Resultados EJB v/s Spring Framework

Luego de implementar las arquitecturas, se procedió a realizar la comparación, la cual arrojó los siguientes resultados:

Tabla 4: Resultados de Comparativa

Nota	EJB 3	Spring Framework	Nota
2	**Manejo de Transacciones (Transaction Manager)**		**3**
Sólo se soporta JTA Transaction Manager. Este es el manejador de transacciones que se usa en forma primaria en las aplicaciones de negocio.		JTA es una de las alternativas disponibles. Pueden usarse diferentes ORM como Hibernate, JDO, JDBC, ODBC.	
1	**Oportunidad de Transacciones (Transaction Opportunities)**		**2**
Sólo los atributos de una transacción son soportados. No se soportan transacciones anidadas.		Soporta atributos de transacción y diferentes niveles de aislamiento. Las transacciones anidadas son soportadas sólo si el manejado de transacciones las implementa.	
3	**Persistencia de Entidades (Entity Persistente)**		**2**
Define su propio manejo de persistencia, posibilidad de emplear anotaciones en ORM, EJB QL y sentencias SQL nativas, además de integración con Hibernate.		Usa implementaciones ORM de terceros como Hibernate, IBATIS, JDO, OJB. Spring provee una mayor diversidad de alternativas para el manejo de persistencia en comparación con EJB, pero el manejo de estas, es mucho más fácil y amigable en EJB.	
1	**Programación Orientada a Aspectos [AOP (Interceptors)]**		**3**
Interceptores por defecto pueden ser especificados y aplicados a todos los componentes, interceptores de callback, Los interceptores pueden ser implementados en la misma clase o en una clase separada. Pueden ser seteados (establecidos) usando anotaciones en el descriptor de despliegue.		Provee servicios de aplicación en forma declarativa, además se puede definir aspectos personalizados.	

<table>
<tr><td>1</td><td colspan="2">Configuración de la aplicación (Application configuration)</td><td>2</td></tr>
<tr><td colspan="2">Usa anotaciones de metadatos en forma primaria, pero es posible sobrescribirlas en el descriptor de despliegue.</td><td colspan="2">En forma primaria se usan archivos XML de configuración, posibilidad de usar Jakarta Commons Attributes ó las anotaciones J2SE estándares.</td></tr>
<tr><td>3</td><td colspan="2">Seguridad (Security)</td><td>2</td></tr>
<tr><td colspan="2">Soporta seguridad declarativa por medio de anotaciones de metadatos y declaraciones en el descriptor de despliegue.</td><td colspan="2">Provee integración con la solución Open Source Acegi Security Framework, el mismo que soporta seguridad declarativa basada en el uso de IoC y AOP.</td></tr>
<tr><td>2</td><td colspan="2">Flexibilidad de Servicios (Services flexibility)</td><td>1</td></tr>
<tr><td colspan="2">Depende de la implementación de EJB. Si el servidor provee una estructura modular entonces sólo se requieren los servicios que este puede usar.</td><td colspan="2">Cualquier servicio puede ser ensamblado usando un archivo XML de configuración.</td></tr>
<tr><td>3</td><td colspan="2">Integración de Servicios (Services Integration)</td><td>2</td></tr>
<tr><td colspan="2">El servidor de aplicaciones contiene una implementación de EJB y tiene la oportunidad de optimizar rendimiento. Existe soporte para clustering.</td><td colspan="2">Spring Framework es desarrollado en forma separada de un servidor de aplicaciones y resulta más dificultoso optimizar su integración. No se aplica si no se usa un servidor de aplicaciones.

Esta característica es una gran desventaja frente a EJB, pero Spring puede ser fácilmente integrado con otros frameworks, y adicionalmente una aplicación Spring puede ser publicada en cualquier servidor de aplicaciones del mercado.</td></tr>
<tr><td>2</td><td colspan="2">Funcionalidad Adicional (Additional Functionality)</td><td>3</td></tr>
<tr><td colspan="2">Depende de la implementación EJB proporcionada por el proveedor.</td><td colspan="2">Provee oportunidades de integración con varios productos Open-Source, Spring MVC.</td></tr>
</table>

5.2 Resultados de la Arquitectura Hibrida

Los resultados de la comparación muestran que Spring Framework es especialmente preferible para usarse en compañías pequeñas, en conjunto con otros productos Open-Source. Es un framework muy simple, conveniente y flexible, pero al mismo tiempo muy poderoso.

En el otro extremo EJB 3 puede ser utilizado por compañías cuyo plan de mantenimiento de aplicaciones basadas en EJB es un compromiso a largo tiempo y buscan que las nuevas versiones de una aplicación sean por completo compatibles con las versiones anteriores. La integración de EJB con el servidor de aplicaciones provee grandes oportunidades en escalamiento, y optimización del programa de desarrollo.

Spring surge como reacción a un modelo EJB bastante distanciado de la simplificación.
Fue planteado originalmente como un mecanismo alterno a la complejidad existente en aproximaciones como EJB 2.x. Spring es una alternativa útil en escenarios que no justifican el manejo de un contenedor EJB, aunque también puede ser utilizada como tecnología complementaria a EJB sobre aquellos aspectos que no cubre la especificación estándar.

Spring puede proporcionar mucha ayuda al código cliente que necesita acceder y utilizar EJBs, tanto por la reducción de la duplicación de código utilizado para acceder a EJBs, y también permitiendo que EJB sea utilizado como si se tratara de cualquier otro servicio accedido a través de interfaz de negocios.

Spring ofrece dos mecanismos que pueden hacer más fácil el acceso y utilización de EJB. Uno de ellos es encapsulando el objeto de búsqueda de inicio de manera que puede ser manejado en una forma declarativa, y el resultado se inyecta en el código de cliente que lo necesita. La otra, utilizable para Stateless Session Beans, es para encapsular, a través de un objeto proxy dinámicamente generado, la búsqueda tanto del objeto y el EJB. Este proxy puede ser inyectado en código de cliente para ser utilizado.

Usar EJB's vía Spring tiene la ventaja de poder trabajar de forma bastante robusta frente a los cambios que se produzcan en el estándar de EJB.

En resumen, los resultados de la integración entre Spring Framework y EJB son los siguientes: (de acuerdo a las métricas de comparación, podemos ver ejemplos de códigos para cada una de las métricas en el Anexo 8):

Tabla 5: Resultados de Integración de Tecnologías

Nota	Arquitectura Hibrida
3	**Manejo de Transacciones (Transaction Manager)**
Es estándar, puesto que pueden ocuparse diferentes ORM como Hibernate, JDO, JDBC, ODBC	
2	**Oportunidad de Transacciones (Transaction Opportunities)**
La implementación soporta atributos de transacción y diferentes niveles de aislamiento. El manejo de	

transacciones anidadas puede implementarse, siempre y cuando se implementen mediante Spring.	
3	**Persistencia de Entidades (Entity Persistente)**
El manejo de la persistencia es mucho más fácil implementarla a través de EJB (por ejemplo, integración con Hibernate).	
3	**Programación Orientada a Aspectos [AOP (Interceptors)]**
Los interceptores pueden ser implementados en la misma clase o en una clase separada utilizando EJB, pero utilizando Spring se puede definir aspectos personalizados, por lo que se hace más completa la implementación.	
3	**Configuración de la aplicación (Application configuration)**
La configuración del sistema es bastantes standard puesto que se ocupa XML para definir parámetros que se deseen ocupar.	
3	**Seguridad (Security)**
Gracias a las propiedades de EJB, la seguridad es standard puesto que se pueden ocupar anotaciones de metadatos y declaraciones en el descriptor de despliegue.	
2	**Flexibilidad de Servicios (Services flexibility)**
Depende de la implementación de EJB. Si el servidor provee una estructura modular entonces sólo se requieren los servicios que este puede usar. Cualquier servicio puede ser ensamblado usando un archivo XML de configuración. Utilizando las propiedades de EJB, se puede concluir que los servicios pueden ser tratados de maneja fácil a través de su estructura modular, puesto que al ocupar XML se no es fácil poder identificar y reutilizar las funciones que traen estos XMLs.	
3	**Integración de Servicios (Services Integration)**
Utilizando notaciones de EJB es fácil optimizar rendimiento, por el estándar que conlleva. En este caso no se ocupó Spring Framework puesto que se tendría que agregar un servicio adicional para poder integrar diferentes servicios que se requieran, por lo que dificultarían su rendimiento.	
3	**Funcionalidad Adicional (Additional Functionality)**
Al incorporar la arquitectura Spring, podemos encontrar una gran gama de funcionalidades adicionales que se pueden instalar a este sistema, por ejemplo seguridad en grupos.	

En resumen, se puede decir que Spring Framework es especialmente preferible para usarse en compañías pequeñas, en conjunto con otros productos Open-Source. Es un framework muy simple, conveniente y flexible, pero la mismo tiempo muy poderoso. Se recomienda el uso de Spring en los casos donde un contenedor de aplicaciones pesado no es necesario.

En el otro extremo EJB 3 puede ser utilizado por compañías cuyo plan de mantenimiento de aplicaciones basadas en EJB es un compromiso a largo tiempo y buscan que las nuevas versiones de una aplicación sean por completo compatibles con las versiones

anteriores. La integración de EJB con el servidor de aplicaciones provee grandes oportunidades en escalamiento, y optimización del programa de desarrollo.

6 Conclusiones y Trabajo Futuro

La especificación 3 de EJB presenta grandes ventajas, como el uso de anotaciones, en lugar de descriptores XML verbosos (que pueden seguir utilizándose y, de hecho, cuando no es posible volver a tocar un código, siguen siendo indispensables), un modelo de programación más liviano y menos intrusivo, una especificación de persistencia mucho más flexible (basada en los ORM que ya existían en el mercado) y el uso de inyección de dependencia en lugar del uso directo de JNDI lookup. Pero también tiene sus inconvenientes, los cuales pueden ser corregidos a través de otras tecnologías como es Spring Framework, el cual trata de mejorar todas las falencias que EJB 3 tiene, siendo así una gran alternativa frente a un problema determinado.

El análisis comparativo de EJB y Spring Framework ha llevado a las siguientes conclusiones generales:

- Los criterios y resultados de la evaluación pueden ser útil para las empresas de TI, para ayudar en la selección y uso de los componentes de gestión del Framework en Negocio de la empresa.
- Los criterios seleccionados son flexibles, por lo que se pueden ampliar de acuerdo a la evolución de las tecnologías.

Este proyecto no termina con este informe, está abierto a cualquier persona que desee indagar en el tema, tomando otras métricas o diseñando una nueva arquitectura para cada tecnología, puesto que todas las métricas son tomas de forma personalizada, de acuerdo a ciertas condiciones específicas que la problemática planteó. Fácilmente se pueden tomar diferentes funcionalidades de Spring y EJB, haciendo que el sistema reaccione de maneja mucho más eficiente la que propuesta indicada, por lo que sería una buena propuesta de tesis, optimizar las arquitecturas propuestas.

7 Referencias

[1] Aynur Abdurazik. "Suitability of the UML as an Architecture Description Language with applications to testing". Reporte ISE-TR-00-01, George Mason University. Febrero de 2000.

[2] Rod Johnson; J2EE development without EJB. Ed Wrox, ISBN: 9780764558313, 2004.

[3] J. Machacek, A. Vukotic, A. Chakraborty, J.Ditt; Pro Spring 2.5. Ed. Apress, 2008.

[4] C. Walls, R. Breidenbac; Spring in action (2nd Ed.). Ed. Manning, 2007.

[5] Stephen Albin. The Art of Software Architecture: Design methods and techniques. Nueva York, Wiley, 2003.

[6] Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, Fabio Peruzzi. "The Architecture Based Design Method". Technical Report, CMU/SEI-2000-TR-001, ESC-TR-2000-001, Enero de 2000.

[7] L. DeMichiel, "Enterprise JavaBeans Specification, Version 2.1", [Documento en línea], Sun Microsystems, November 12, 2003, 646 pages, Available at HTTP: http://java.sun.com/products/ejb/docs.html

[8] . DeMichiel, M. Keith "JSR 220: Enterprise JavaBeans, Version 3.0. EJB 3.0 Simplified API", [Documento en línea], Sun Microsystems, December 18, 2005, 59 pages, Available at HTTP: http://jcp.org/aboutJava/communityprocess/pfd/jsr220/index.html

[9] J. Graudins "Comparing analysis of Java application servers", Scientific proceedings of Riga Technical University, 2004, p. 118 " 125.

[10] R. Lambert, "An Introduction to the Spring Framework", [Documento en línea], Chicago Java Users Group, June 21, 2005, Available at HTTP: http://cjug.org/presentations/2005/June21/Spring-Framework-Intro-Rob-Lambert.ppt

[11] R. Johnson, J. Hoeller, A. Arendsen "Spring. Java/J2EE Application Framework", [Online document], 2004-2005, 290 pages, Available at HTTP: http://www.springframework.org/documentation.

[12] R. Mordani "Common Annotations for the Java Platform", [Online document], Sun Microsystems, October 12, 2005, 32 pages, Available at HTTP: http://jcp.org/aboutJava/communityprocess/pfd/jsr250/index.html

8 Anexos

Manejo de Transacciones (Transaction Manager)

Para el siguiente ejemplo se usará Spring Framework, pero los conceptos de transacciones son los mismos para la especificación EJB 3.0. En la mayoría de los casos, es sólo cuestión de reemplazar la anotación de Spring Framework @Transactional con la anotación de EJB 3.0 @TransactionAttribute.

```
@Stateless
public class TradingServiceImpl implements TradingService {
   @Resource SessionContext ctx;
   @Resource(mappedName="java:jdbc/tradingDS") DataSource ds;

   public long insertTrade(TradeData trade) throws Exception {
      Connection dbConnection = ds.getConnection();
      try {
         Statement sql = dbConnection.createStatement();
         String stmt =
            "INSERT INTO TRADE (ACCT_ID, SIDE, SYMBOL, SHARES, PRICE, STATE)"
          + "VALUES ('"
          + trade.getAcct() + "','"
          + trade.getAction() + "','"
          + trade.getSymbol() + "',"
          + trade.getShares() + ","
          + trade.getPrice() + ",'"
          + trade.getState() + "')";
         sql.executeUpdate(stmt, Statement.RETURN_GENERATED_KEYS);
         ResultSet rs = sql.getGeneratedKeys();
         if (rs.next()) {
            return rs.getBigDecimal(1).longValue();
         } else {
            throw new Exception("Trade Order Insert Failed");
         }
      } finally {
         if (dbConnection != null) dbConnection.close();
      }
   }
}
```

Este código JDBC no incluye ninguna lógica de transacción, y persiste una orden de compra de acciones en la tabla TRADE de la base de datos. En este caso, la base de datos se encarga de manejar la lógica transaccional.

Oportunidad de Transacciones (Transaction Opportunities)

En el siguiente ejemplo, supongamos que necesitamos actualizar el balance de la cuenta en el mismo momento que insertamos la orden de compra en la base de datos. Por ejemplo:

```
public TradeData placeTrade(TradeData trade) throws Exception {
   try {
      insertTrade(trade);
      updateAcct(trade);
      return trade;
   } catch (Exception up) {
      //log del error
      throw up;
   }
}
```

En este caso, los métodos insertTrade() y updateAcct() usan código estándar JDBC sin transacciones. Una vez que el método insertTrade() finaliza, la base de datos persiste (y hace commit) a la orden de compra. Si el método updateAcct() falla por cualquier razón, la orden de compra seguirá en la tabla TRADE al finalizar el método placeTrade(), resultando en datos inconsistentes en la base de datos. Si el método placeTrade() hubiera usado transacciones, ambas actividades se hubieran incluido en la misma ULT, y se hubiera realizado un rollback de la orden de compra si la actualización de la cuenta fallaba.

Con la popularidad de los frameworks de persistencia Java como Hibernate, TopLink y Java Persistence API (JPA), es bastante raro escribir código JDBC directamente. Generalmente se usa los frameworks de mapeo objeto-relacional (ORM) para hacernos la vida más fácil y reemplazar todo el código JDBC molesto por unas simples invocaciones. Por ejemplo, para insertar una orden de compra del ejemplo anterior en JDBC, usando Spring Framework con JPA, se mapea el objeto TradeData a la tabla TRADE y se reemplaza todo el código JDBC con el código JPA:

```
public class TradingServiceImpl {
    @PersistenceContext(unitName="trading") EntityManager em;

    public long insertTrade(TradeData trade) throws Exception {
       em.persist(trade);
       return trade.getTradeId();
    }
}
```

Podemos ver que en este ejemplo invocamos al método persist() del EntityManager para insertar una orden de compra. Este código no va a insertar la orden de compra en la tabla TRADE como se esperaría ni tampoco lanza una excepción. Simplemente va a retornar el valor 0 como la clave de la orden de compra sin cambiar la base de datos. Este es uno de los primeros errores al procesar transacciones: los frameworks ORM necesitan de transacciones para disparar la sincronización entre el caché de objetos y la base de datos. Es a través del commit de la transacción que el código SQL se genera y se afecta a la base de datos con la acción (es decir, con el insert, update, delete). Sin la transacción no hay un disparador al ORM para que genere el código SQL y persista los cambios, y así el método termina sin excepciones, sin actualizaciones. Si se usa un framework ORM, se debe usar transacciones. Ya no se puede confiar en que la base de datos va a manejar las conexiones y hacer el commit del trabajo.

Estos ejemplos simples deberían bastar para dejar en claro que las transacciones son necesarias para mantener integridad y consistencia de datos. Pero apenas son el inicio de la complejidad y los problemas asociados con la implementación de transacciones en Java.

Probando el código anterior y se descubre que el método persist() no funciona sin transacciones. Investigando se ha encontrado que Spring Framework tiene una anotación @Transactional. Así que se agrega al código, como vemos a continuación:

```
public class TradingServiceImpl {
   @PersistenceContext(unitName="trading") EntityManager em;

   @Transactional
   public long insertTrade(TradeData trade) throws Exception {
      em.persist(trade);
```

```
      return trade.getTradeId();
   }
}
```

Se vuelve a probar el código, y sigue sin funcionar. El problema es que hay que decirle a Spring que estamos usando anotaciones para gestionar las transacciones, a menos que estemos haciendo una prueba unitaria completa, este error suele ser difícil de descubrir, y hace que los desarrolladores terminen añadiendo la lógica de transacciones en los archivos de configuración de Spring en vez de usar anotaciones.

Cuando se usa la anotación @Transactional de Spring, debemos agregar la línea siguiente a nuestro archivo de configuración de Spring:

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

La propiedad transaction-manager tiene una referencia al bean que gestiona las transacciones, definido en el archivo de configuración de Spring. Este código le dice a Spring que use la anotación @Transactional cuando aplique el interceptor de transacciones. Sin esta línea, se ignora la anotación @Transactional, y en consecuencia no se usan transacciones en el código.

Este es el uso básico de la anotación @Transactional, y es sólo el comienzo. En el ejemplo se usó la anotación @Transaction sin especificar ningún parámetro adicional a al anotación.

Persistencia de Entidades (Entity Persistente)

A la hora de poder integrar Hibernate en EJB, se pudo constatar que en EJB es mucho más sencillo que en Spring Framework.

Para poder integrar Hibernate es necesario crear jboss-Hib-service.xml y ponerlo en el directorio del deploy. El objetivo es crear un servicio y registrarse en JNDI, para que se pueda utilizar para interactuar con DB usando Hibernate.

A continuación se presenta un ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE server>
<server>
         <mbean code="net.sf.hibernate.jmx.HibernateService"
name="jboss.jca:service=HibernateLaborMgmt">
         <depends>jboss.jca:service=RARDeployer</depends>
                <attribute
name="MapResources">org/learn/labormgmt/employee/value/EmployeeValues.hbm.xml,org/le
arn/labormgmt/customer/value/CustomerValues.hbm.xml,org/learn/labormgmt/location/val
ue/StateValues.hbm.xml,org/learn/labormgmt/association/value/AssociationValues.hbm.x
ml</attribute>
       <attribute name="JndiName">java:/hibernate/labormgmt</attribute>
       <attribute name="Datasource">java:/labormgmt</attribute>
       <attribute name="Dialect">org.hibernate.dialect.SAPDBDialect</attribute>
       <attribute name="ShowSql">true</attribute>
       <attribute name="UserTransactionName">UserTransaction</attribute>
       <attribute
name="TransactionStrategy">org.hibernate.transaction.JTATransactionFactory</attribut
e>
```

```
       <attribute
name="TransactionManagerLookupStrategy">org.hibernate.transaction.JBossTransactionMa
nagerLookup</attribute>
   </mbean>
</server>
```

En el código para recibir el servicio, que es en realidad Session Factory:

```
try {
//JNDI lookup to get Hibernate Session factory.
    SessionFactory factory = (SessionFactory)ServiceLocator.getInstance()
        .getService("java:/hibernate/labormgmt");
    } catch (HibernateException e) {
      logger.debug("Error while initilising Hibernate." + e.getMessage());
      throw new InstantiationError("Error while initilising Hibernate."
          + e.getMessage());
    } catch (LocatorException e) {
      logger.debug("Error while initilising Hibernate." + e.getMessage());
      throw new InstantiationError("Error while initilising Hibernate."
          + e.getMessage());
    }
```

Programación Orientada a Aspectos [AOP (Interceptors)]

Existen distintos tipos de interceptores, que interceptan distintos aspectos (excepciones, antes de un método, después de un método, etc.).

Crear un interceptor de método es tan simple como hacer una clase que herede de org.aopalliance.intercept.MethodInterceptor

MethodInterceptor es el interceptor más amplio que existe, y muy fácil de implementar.

```
package com.dosideas.interceptor;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

 public class LogInterceptor implements MethodInterceptor {
     public Object invoke(MethodInvocation invocation) throws Throwable {
         System.out.println("Iniciando llamada");
         Object rval = invocation.proceed();
         System.out.println("Finalizando llamada");
         return rval;
     }
 }
```

Configuración de la aplicación (Application configuration)

En las aplicaciones web, Spring puede configurarse de manera muy simple a través de un Listener o un Servlet.

En el archivo web.xml se agrega la variable de contexto contextConfigLocation, la cual apunta a los archivos de configuración. Luego, puede utilizarse el listener ContextLoaderListener o el servlet ContextLoaderServlet (ambos provistos por el framework) para que inicialicen el contexto de Spring.

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
```

```
        <param-value>
            classpath:archivoDeSpring1.xml,
            classpath:archivoDeSpring2.xml
        </param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

Seguridad (Security)

Una anotación o metadato proporciona un recurso adicional al elemento de código al que va asociado en el momento de su compilación. Cuando la JVM ejecuta la clase busca estos metadatos y determina el comportamiento a seguir con el código al que va unido la anotación.

En el siguiente ejemplo se ocupa la notación @Stateless, el cual establece que el Bean de Sesión es sin estado.

```
package com.autentia.ejb3.bean;

import java.util.List;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import org.apache.log4j.Logger;
import com.autentia.ejb3.bean.entity.BancoEB;
import com.autentia.ejb3.bean.entity.ContratoEB;
import com.autentia.ejb3.bean.entity.CuentaEB;
import com.autentia.ejb3.bean.entity.EmpleadoEB;
import com.autentia.ejb3.dao.EmpleadoDAOLocal;
import com.autentia.ejb3.util.Empleado;

/* Indica que el Bean de Sesión es sin estado */
@Stateless
public class EmpleadoMgrBean implements EmpleadoMgrLocal {

     public static Logger log = Logger.getLogger(EmpleadoMgrBean.class);

     /* Inyección del EJB EmpleadoDAOLocal */
     @EJB
     private EmpleadoDAOLocal empleadoDAO;

     public List<EmpleadoEB> listarEmpleados() {
          return empleadoDAO.buscarEmpleados();
     }

     public boolean altaEmpleado(Empleado empleado) {
          EmpleadoEB empleadoEB = new EmpleadoEB();
          empleadoEB.setNombre(empleado.getNombre());
          empleadoEB.setApellidos(empleado.getApellidos());
          empleadoEB.setNif(empleado.getNif());
          empleadoEB.setContrato(new
ContratoEB(empleado.getSalario(),empleado.getContrato()));
          empleadoEB.setCuenta(new CuentaEB(empleado.getCuenta(), new
BancoEB(empleado.getBanco())));
          empleadoDAO.nuevoEmpleado(empleado);

          if (empleado.getIdEmpleado() != 0) {
               return true;
```

```
            }
            return false;
      }

      public boolean eliminarEmpleado(Empleado empleado) {
            return empleadoDAO.bajaEmpleado(empleado);
      }
}
```

Como se puede observar, el uso de anotaciones es muy fácil y entendible a la hora de desarrollar un software. Existen numerosas anotaciones que nos ayudan al control de la seguridad dentro del EJB (las más comunes):

Anotaciones de un Bean de Sesión

- @Stateful: Indica que el Bean de Sesión es con estado.
- @Stateless: Indica que el Bean de Sesión es sin estado.
- @Init: Especifica que el método se corresponde con un método create de un EJBHome o EJBLocalHome de EJB 2.1. Sólo se podrá; llamar una única vez a este método.
- @Remove: Indica que el contenedor debe llamar al método cuando quiera destruir la instancia del Bean.
- @Local: Indica que la interfaz es local.
- @Remote: Indica que la interfaz es remota.
- @PostActivate: Invocado después de que el Bean sea activado por el contenedor.
- @PrePassivate: Invocado antes de que el Bean esté en estado passivate.

Anotaciones de un Bean de Entidad

- @Entity: Indica que es un Bean de Entidad.
- @EntityListeners: Se pueden definir clases oyentes (listeners) con métodos de ciclo de vida de una entidad. Para hacer referencia a un listener se debe incluir esta anotación seguido entre paréntesis de la clase: @Entity Listeners(MyListener.class)
- @ExcludeSuperclassListeners: Indica que ningún listener de la superclase será; invocado por la entidad ni por ninguna de sus subclases.
- @ExcludeDefaultListeners: Indica que ningún listener por defecto será; invocado por esta clase ni por ninguna de sus subclases.
- @PrePersist: El método se llamará; justo antes de la persistencia del objeto. Podría ser necesario para asignarle la clave primaria a la entidad a persistir en base de datos.
- @PostPersist: El método se llamará; después de la persistencia del objeto.
- @PreRemove: El método se llamará; antes de que la entidad sea eliminada.

- @PostRemove: El método se llamará; después de eliminar la entidad de la base de datos.
- @PreUpdate: El método se llamará; antes de que una entidad sea actualizada en base de datos.
- @PostUpdate: El método se llamará; después de que la entidad sea actualizada.
- @PostLoad: El método se llamará; después de que los campos de la entidad sean cargados con los valores de su entidad correspondiente de la base de datos. Se suele utilizar para inicializar valores no persistidos.
- @FlushMode: Modo en que se ejecuta la transacción: FlushModeType.AUTO (por defecto) y FlushModeType.COMMIT.

Flexibilidad de Servicios (Services flexibility)

La flexibilidad de los servicios que ofrece Spring Framework es alta y fácilmente implementable. En el siguiente ejemplo, al archivo de configuración de Spring Framework se le agregará soporte para un servicio de transacciones, y también se le agregarán algunos servicios adicionales para el manejo de AOP.

El contenido de este archivo quedaría finalmente como sigue:

```
<?xml version='1.0'?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="byName">
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
     <property name="driverClassName"><value>...</value></property>
     <property name="url"><value>...</value></property>
     <property name= "username"><value>...</value></property>
     <property name="password"><value>...</value></property>
  </bean>

<bean id="sessionFactory" class=
"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
<property name="annotatedClasses">
     <list>
        <value>single_table_simple_key_spring.FacturaDTO</value>
     </list>
</property>
<property name="hibernateProperties">
     <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.InformixDialect</prop>
     </props>
</property>
</bean>
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager"/>
<bean id="transactionInterceptor" class=
     "org.springframework.transaction.interceptor.TransactionInterceptor">
<property name="transactionAttributeSource">
    <bean class=
"org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"
/>
```

```
</property>
</bean>
<bean class=
"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
/>
<bean class=
"org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"
/>
<!-- ADMINISTRADORES -->
<bean id="objectManager"
      class="single_table_simple_key_spring.ObjectManagerImpl"/>
</beans>
```

Nótese la dependencia de los servicios entre sí. El administrador de transacciones depende de la fábrica de sesiones Hibernate; y la definición de los interceptores depende del administrador de transacciones (ya que los servicios AOP controlarán las transacciones para los servicios).

Integración de Servicios (Services Integration)

Utilizando notaciones de EJB es fácil optimizar rendimiento, por el estándar que conlleva, puesto que los servidores de aplicaciones contienen una implementación de EJB y tiene la oportunidad de optimizar rendimiento. En este caso no se ocupó Spring Framework puesto que se tendría que agregar un servicio adicional para poder integrar diferentes servicios que se requieran, por lo que dificultarían su rendimiento.

Funcionalidad Adicional (Additional Functionality)

Dentro de las bondades que nos aporta Spring Framework, está la flexibilidad de las funcionales que se pueden ocupar a la hora de desarrollar un software, por ejemplo, la implementación de Spring – MVC es una funcionalidad que se puede agregar con facilidad al sistema.

La poder ocupar esta funcionalidad, debemos indicar a nuestra aplicación web que utilice la servlet (la llamaremos holamundo) implementada por Spring (org.springframework.web.servlet.DispatcherServlet) que atienda a las URLs con el patrón *.lycka . Sip, eso lo hacemos en el web.xml.

```
<servlet>
      <servlet-name>holamundo</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
      <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
      <servlet-name>holamundo</servlet-name>
      <url-pattern>*.lycka</url-pattern>
</servlet-mapping>
```

Spring MVC también tiene su archivo de configuración, al estilo del struts-config.xml de Struts. En este caso el nombre es el nombre de la servlet (<servlet-name>) seguido de “-servlet.xml”. En este caso, holamundo-servlet.xml.

```
<?xml version=“1.0 encoding=“UTF-8?>
<beans xmlns=“http://www.springframework.org/schema/beans”
```

```
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
	<!- the application context definition for the springapp DispatcherServlet ->
	<bean name="/holamundo.html"
class="es.lycka.holamundo.control.HolaMundoController"/>
	<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"></property>
		<property name="prefix" value="jsp/"></property>
		<property name="suffix" value=".jsp"></property>
	</bean>
</beans>
```

Definimos un controlador (es.lycka.holamundo.control.HolaMundoController), que atenderá la llamada a un path ("/holamundo.lycka").

```
<bean name="/holamundo.lycka"
class="es.lycka.holamundo.control.HolaMundoController"/>
```

Este controlador realizará sus operaciones pertinentes, típicamente invocar a un servicio y según su resultado decidirá el siguiente paso: devolverá como resultado una vista (una jsp) con sus parámetros correspondientes. En este caso, se dirigirá a la vista "holamundo" pasando un parámetro, "ahora".

```
public class HolaMundoController implements Controller {
	protected final Log logger = LogFactory.getLog(getClass());
	public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response)
		throws ServletException, IOException {
		String ahora = (new Date()).toString();
		logger.info("Se dirige a la vista holamundo con ahora = " + ahora);
		return new ModelAndView("holamundo", "ahora", ahora);
	}
}
```

holamundo_proyectospringmvc.JPGAhora entra en juego el otro bean definido en nuestro xml, "viewResolver". Para independizar la ubicación de los ficheros de las vistas de los controladores podemos definir reglas para encontrar la vista a partir de su nombre. En este caso, definimos que van a estar dentro del directorio "jsp/" y su extensión será ".jsp".

Así que nos queda construir la jsp que responderá, holamundo.jsp dentro del directorio jsp.

```
<html>
   <head><title>Hola Mundo</title></head>
   <body>
      <h1>Éxito !</h1>
      <p>Hola, has llegado correctamente ahora mismo, <c:out value="${ahora}"/></p>
   </body>
</html>
```

Llegados a este punto se puede desplegar en nuestro servidor y probar qué nos devuelve si introducimos en nuestro navegador la ruta "<URL_Servidor_Aplicaciones>/holamundo/holamundo.lycka".