Arquitectura de Software y Entornos de Trabajo (Frameworks) de Contenedores Ligeros

Autor: Damian Ciocca

Director: Javier Blanqué

Universidad Nacional de Luján
Int. Ruta 5 y 7
6700 Luján, Buenos Aires
República Argentina
Año 2008

Arquitectura de Software y Entornos de Trabajo (Frameworks) de Contenedores Ligeros

Damian Ciocca
Universidad Nacional de Luján
Int. Ruta 5 y 7
6700 Luján, Buenos Aires
República Argentina
damianciocca@gmail.com

RESUMEN

El presente trabajo explorará las arquitecturas de software J2EE y los entornos de contenedores ligeros para el desarrollo de software y cómo algunos autores hacen hincapié en los POJOs, no como una nueva tecnología ni una técnica, pero si, como una filosofía que permite al desarrollador concentrarse en la lógica de negocio. Al mismo tiempo, estos autores destacan permanentemente que el framework que se utilice no debe ser intrusivo y el dominio debe estar aislado, desacoplando los objetos de dominio del resto del sistema.

Se estudiarán los conceptos y definiciones necesarias para la comprensión de la arquitectura del software y los entornos de trabajo o frameworks. También se focalizará en la utlización de capas lógicas para el desarrollo de software, conceptos relevantes del diseño arquitectónico del software y patrones de diseño, teniendo en cuenta las buenas prácticas de la POO (Programación Orientada a Objetos).

Por otra parte se desarrollará una comparación exhaustiva entre las arquitecturas EJB y sin EJBs, siendo esto el punto de partida para el estudio de los contenedores ligeros.

Luego se analizará a uno de los contenedores ligeros más populares de la actualidad: Spring y el patrón de diseño: ID (Inyección de Dependencia) y cómo los mismos promueven el uso de las interfases, siendo un recurso muy poderoso en la POO.

Como corolario de este trabajo, se explicará una serie de técnicas para la realización de pruebas unitarias durante el ciclo de vida del desarrollo del software y se volcará todo lo explicado anteriormente en un ejemplo práctico para que el lector logre una mejor comprensión.

Palabras clave:

ID. Inyección de Dependencia. POJO. Arquitectura del Software. Entornos de Trabajo. Frameworks. Contenedores Ligeros. Patrón MVC. EJB. J2EE. Java. Inversión del Control. Spring.

Agradecimientos:

Quiero agradecer en primer lugar a mi director de Tesis, Profesor Javier Blanqué por brindarme su tiempo y su dedicación en este proyecto. También quiero agradecer a toda mi familia, porque me han regalado su apoyo durante el desarrollo de este trabajo y toda la carrera.

A mis amigos y compañeros de estudio Santiago Fiore, Gustavo Vega, Maria Luz Blanco, Juan Pablo Sarubbi, Marcelo Fernandez por sus experiencias, consejos, críticas y sobre todo, por su total acompañamiento.

A todas las personas que contribuyeron directa o indirectamente aportando en este trabajo y por último a Macarena Amallo, mi novia, por estar conmigo, por entender mi falta de tiempo y brindarme siempre su apoyo incondicional.

Objetivos del trabajo....9
1.1 Introducción a la arquitectura del software....12
1.1.1 Introducción a los entornos de trabajo (frameworks)....14
1.2 Un poco de historia....15
1.2.1 Smalltalk....18
1.2.2 NeXT y NEXTSTEP....19
1.3 Requerimientos de una arquitectura de software....19
1.3.1 Performance y escalabilidad de las aplicaciones....20
1.3.2 Como decidir si la aplicación necesita un servidor de aplicaciones....21
1.4 Principios para evaluar arquitecturas....23
1.5 Aplicaciones multicapa....26
1.5.1 Evolución de las capas....27
1.5.2 Separación en capas....30
1.5.3 Capas lógicas....32
1.5.4 Capas lógicas VS físicas....35
1.6 El diseño arquitectónico del software....36
1.6.1 Los patrones y la arquitectura del software....37
1.6.2 Un poco de historia....37
1.6.3 Patrones vs. Anti-patrones....38
1.6.3.1 Patrón: MVC (Modelo Vista Controlador)....39
1.6.3.2 Anti-patrón: Dominio anémico....41
1.6.4 Patrones GOF vs. Patrones GRASP....41
1.8.1 SOA: Service Oriented Architecture (Arquitectura Dirigida por Servicios)....43
1.8.2 MDA: Model Driven Architecture (Arquitectura Dirigida por Modelos)....44
1.8.3 Test-Driven Development (Desarrollo Dirigido por los Test)....45
1.9 Transacciones....47
1.9.2 Propiedades de las transacciones....48
1.9.3 Niveles de aislamiento de las transacciones....48
1.9.4 Transacciones distribuidas vs. locales....49
1.9.5 Transacciones declarativas vs. programáticas....50
1.10 Pool de conexiones, fuentes de datos y servicios de directorios JNDI....51
Capítulo 2 - Arquitecturas con EJB vs. Arquitecturas sin EJB - 54
2.1 Arquitecturas no distribuidas....54
2.1.1 Web con Business Component Interfaces (Interfases de Negocio)....54
2.1.2 Web con EJB locales....55
2.1.3 Web con Arquitectura de Contenedor Ligero....57
2.1.3.1 Separación en capas....59
2.1.3.2 Inyección de Dependencias (ID) y el Control de Inversión (IoC)....62
2.1.3.3 Formas de inyección de dependencias (ID)....64
2.2 Arquitecturas distribuidas....67
2.2.1 Web con EJB remotos....67
2.2.2 Web con Web Services (servicios web)....68
2.3 Comparación de Debilidades y Fortalezas....71
Capítulo 3 – Spring – 73
3.1 Introducción....73
3.1 Un poco de historia....74
3.2 La especificación EJB 3.0....76

3.4 Problemas de las arquitecturas J2EE tradicionales....79
3.5 EJB 3.0 VS Spring....80
3.6 Contenedores Ligeros....84
3.7 La arquitectura de Spring....85
3.7.1 BeanFactory y ApplicationContext....86
3.7.2 El ciclo de vida de un objeto....87
3.7.3 Breve descripción de la arquitectura de spring....89
3.7.4 Como trabaja la Inversión del Control y la Inyección de Dependencia....91
3.8 Ejemplo práctico - Spring, arquitectura de capas y la ID....92
Capítulo 4 – Testing en aplicaciones de contenedores ligeros – 93
4.1 La importancia de los test unitarios y de la filosofía TDD....93
4.2 Buenas prácticas....95
4.3 Categorías de las Herramientas de testeo....96
4.4 Mock Testing (pruebas con objetos Mocks) VS. Pruebas de integración....96
4.4.1 Mock Test....97
4.4.2 Test de integración....97
4.5 Testeando con Spring....98
4.6 Testeando las diferentes capas....98
4.6.1 Testeando la capa de acceso a los datos (DAO) con DbUnit....99
....102
4.6.2 Testeando la capa de servicios (Managers)....102
4.6.2.1 Testeando la capa de servicio (manager) con EasyMock....103
4.6.3 Testeando la capa Web (testeando la vista y los controladores)....105
....108
Capítulo 5 – Ejemplo práctico....109
5.1 Módulos y dependencias....111
5.2 Grafico de dependencias entre los módulos....114
5.3 Organización lógica de los fuentes Java....115
5.4 Configuración de Spring con Hibernate y Struts....116
5.5 Desarrollo de los test unitarios....126
5.5.1 Pruebas unitarias para la capa de persistencia (módulo BesyAppHibernate)....126
5.5.2 Pruebas unitarias para la capa de servicio (módulo BesyAppService)....128
5.5.3 Pruebas unitarias para la capa de presentación (módulo BesyAppWeb)....129
5.6 Desarrollo de las clases funcionales....129
5.6.1 Clases funcionales para la capa de persistencia (módulo BesyAppHibernate)....129
5.6.2 Clases funcionales para la capa de servicio (módulo BesyAppService)....135
5.6.3 Clases funcionales para la capa de presentación (módulo BesyAppWeb)....137
....141
5.6.4 Clases funcionales para la capa de dominio (módulo BesyAppDomain)....144
Conclusiones....150
Anexo....152
A.1 Estudio comparativo entre Wikipedia y la Enciclopedia Británica....152
Bibliografía....162

Introducción

De acuerdo a las tendencias del mercado actual, existen dos grandes tecnologías para el desarrollo de aplicaciones empresariales:

- J2EE (Java 2 Enterprise Edition, Java 2 Edición Empresarial)
- .NET

Si bien estas dos arquitecturas se dan como las más importantes del momento, la más utilizada en la construcción de aplicaciones web en empresas es LAMP (Linux, Apache, MySQL, PHP | Python | Perl). A veces MySQL se reemplaza por PostgreSQL. Según dos artículos hallados en Internet, esta terminología se emplea para definir el trabajo conjunto con Linux, Apache como web Server, MySql como base de datos y uno de los siguientes lenguajes: PHP, Python ó Perl [INT04] [INT05].

En este trabajo, se hará hincapié en la tecnología J2EE (Java), más específicamente en la tecnología de contenedores ligeros y se dejará de lado todo lo referente a .NET.

Tomando las palabras de Michael Juntao Yuan, quien comenta que *"Las aplicaciones de contenedores ligeros son en la actualidad la acumulación de tanta rabia, en la comunidad Java, de años pasados."* [INT41 - Michael Juntao Yuan], podemos sostener que la creciente expansión de la tecnología de contenedores ligeros es en gran parte debido a la antipatía de los desarrolladores para con la utilización de los contenedores pesados como lo es EJB 2.1.

Con la llegada de Spring e Hibernate (entornos de trabajo pioneros en la difusión de técnicas como: *POA* - Programación Orientada a Aspecto -, *uso de anotaciones* - una de las nuevas características que tiene Java en sus últimas versiones -) y las nuevas especificaciones de los estándares EJB 3.0, los entornos de trabajo ligeros se han vuelto cada vez más importantes e imprescindibles en cualquier diseño y desarrollo de sistema [INT41].

Existen muchos autores como por ejemplo Chris Richardson [CR06] y Brian Sam-Bodden [BSA06] que consideran que los POJOs (Plain Old Java Objects, Objetos Planos de Java) junto a los contenedores ligeros hacen los desarrollos mucho más simples, rápidos, fáciles de testear y mantenibles. También señalan cómo éstos difieren a de los Enterprise Java Beans (EJB). Por otro lado, hacen hincapié en que los POJOs van de la

mano de la POO (Programación Orientada a Objetos) lo que hace al código mucho más robusto.

Los POJOs no son una nueva tecnología ni una técnica pero si una filosofía que permite al desarrollador concentrarse en la lógica de negocio logrando así aplicaciones mucho más robustas. Básicamente es como una vuelta a las raíces. Este movimiento de simplificar las cosas y repensar las prácticas de desarrollo es el que impulsa a la comunidad del open source (codigo abierto) al desarrollo de una gran variedad de contenedores ligeros. [BSA06]

Por otro lado, la nueva especificación de EJB 3.0, introdujo numerosos cambios a favor respecto de la especificación 2.0. Los desarrolladores que venían de la especificación anterior encontraron diferencias notables. Sin embargo, no fue así para los que venían del mundo de los contenedores ligeros, ya que muchas de estas características ya estaban presentes en los contenedores ligeros. [CR06]

Para finalizar esta introducción, es necesario destacar que los entornos de trabajo de contenedores ligeros han promovido mejor y de una forma más limpia la arquitectura de las aplicaciones y hacen más fácil reutilizar componentes del negocio cuando se cambian los proveedores. [INT41].

Objetivos del trabajo

El propósito principal de este trabajo es dar una introducción a las arquitecturas de software y entornos de trabajo de contenedores ligeros para el diseño y desarrollo de software, así como también todo aquel concepto que resulte de vital importancia para lograr una mayor comprensión del tema.

Para ello se considera necesario introducir conceptualmente los siguientes puntos:

- Alternativas en el diseño y construcción de medianos y grandes sistemas.
- Enfoque tradicional de la arquitectura de software y entornos de trabajo.
- Principales arquitecturas, diseños, procedimientos, técnicas y herramientas utilizadas en todo proceso de diseño y desarrollo del software.

Otro fin de este trabajo es que sea de utilidad como material de estudio para alumnos de materias de programación y sistemas de información. Los mismos podrán, de una forma más elegante y sin desviarse de lo que dice la teoría de objetos, implementar los conocimientos adquiridos en un modelo concreto, separado por capas bien definidas.

El trabajo se divide en seis capítulos: A continuación se realizará una breve descripción de los objetivos a alcanzar en cada uno de ellos. Así mismo, se considera oportuno diferenciar los primeros capítulos del resto. Es decir, el primero tiene como objetivo proveer una introducción al diseño y arquitectura del software, entornos de trabajo y conceptos generales. El segundo nos brindará una descripción y comparación entre las tecnologías EJB y la alternativa propuesta, sin EJB. Mientras que en el resto de los capítulos se hará más hincapié en las arquitecturas sin EJB (arquitecturas de contenedores ligeros) y sus implementaciones en el mercado.

Capítulo 1: Introducción a la Arquitectura del Software, Entornos de Trabajo frameworks). Conceptos generales

En principio, se realizará una Introducción y reseña histórica de la arquitectura del software y entornos de trabajo (frameworks). Luego, se detallarán requerimientos y principios para evaluar arquitecturas de software como así también se describirán los fundamentos de las aplicaciones multicapas y como influyen los patrones de diseño en la

arquitectura del software. Y como corolario del capitulo, se describirán algunas arquitecturas populares del momento (SOA y MDA) y se mencionará una técnica de programación que es cada vez más popular en el desarrollo de aplicaciones de contenedores ligeros (Programación Dirigida por Test). Esta técnica será detallada en el último capítulo.

Capítulo 2: Arquitecturas EJB vs. sin EJB

Se realizará una comparación entre las arquitecturas EJB y las arquitecturas sin EJB, destacando sus diferencias, ventajas y desventajas. Al mismo tiempo, se considera necesario realizar una breve introducción a la Inyección de dependencias que luego será ampliada en el capitulo 3.

Capítulo 3: Spring

Este capitulo cubrirá los conceptos básicos del framework Spring. Se comenzará con una breve reseña histórica. Luego se realiza una descripción de la tecnología EJB 3.0 y se la compara con Spring, haciendo hincapié en el problema de las arquitecturas J2EE tradicionales.

Por otro lado, se detallarán cuales son las características de un contenedor ligero lo que nos permitirá entrar al mundo del contenedor ligero Spring. Se explicará cual es su arquitectura y las clases principales. Se detallará cual es el ciclo de vida de un objeto dentro de este contenedor. Es decir, como un contenedor ligero (en este caso Spring) administra los objetos Java.

Capítulo 4: Testing en aplicaciones de contenedores ligeros.

En este último capitulo se cubrirá una de las técnicas más difundidas para el desarrollo que permite crear aplicaciones de alta calidad y bien testeadas. Se realizará una explicación sobre como testear los componentes o piezas de software asociadas a

las diferentes capas, dentro de una arquitectura basada en el framework Spring y la diferencia entre los test de integración y los mock test (explicados más adelante).

Las pruebas o testing del código dependerá de la capa a la cual éste pertenece. Para los objetos que se encuentran en la capa de acceso a los datos se analizará DbUnits. Para los objetos que pertenecen a la capa de servicios se analizará EasyMocks. Para los componentes de software que actúan como controladores, se explicará como usar la herramienta: StrutTestCase.

Capitulo 5: Ejemplo práctico.

Como corolario de este trabajo, se llevará a cabo una descripción más detallada de lo que es la inyección de dependencias y la inversión del control a traves de un ejemplo concreto utilizando Spring como framework de aplicación junto con Struts y Hibernate para el manejo de la presentacion y persistencia de datos respectivamente.

Capítulo 1 - Introducción a la Arquitectura del Software, Entornos de Trabajo (frameworks). Conceptos generales -

1.1 Introducción a la arquitectura del software

En primer lugar, se tomarán tres definiciones de la arquitectura del software que consideraremos como puntos de anclaje para lograr una mayor comprensión de la misma.

- Como se menciona en Wikipedia (Software Architecture), "e*s el proceso y la disciplina para hacer más eficiente la implementación del diseño de un sistema*" [INT01].

- La arquitectura del software se concentra en cómo será el diseño de los componentes de software y cómo hacer para que trabajen todos juntos [INT02].

- Es un conjunto de decisiones de diseño las cuales, si se toman incorrectamente, causan que el proyecto de software sea cancelado [INT03].

En función de las definiciones anteriores, podemos comprender que la arquitectura del software ayuda a lograr un diseño eficiente del sistema, centrándose en cómo se interrelacionan todos los componentes del software. Así mismo, la relevancia e importancia de dicha arquitectura, hace que cualquier diseño incorrecto, produzca el fracaso del proyecto de software.

Por otro lado, Les Bass et al. [BCK03] también destaca la importancia de la arquitectura del software basándose en los siguientes puntos:

- Representa una abstracción del sistema.

- Manifiesta las primeras decisiones de diseño.

- Constituye un modelo relativamente pequeño, permitiendo ver cómo se estructura un sistema y cómo sus elementos trabajan juntos. Este modelo es transferible a través de sistemas.

Podemos entrever que de distintas ópticas y autores, encontramos un común denominador en la importancia de la arquitectura del software en función del éxito o fracaso del proyecto de software.

1.1.1 Introducción a los entornos de trabajo (frameworks)

A continuación se definirá el concepto de framework en función de la importancia que él mismo tiene en el desarrollo de sistemas.

Un framework de arquitectura es una estructura de soporte en el cual el desarrollo [de un sistema de información] puede ser organizado y encarado con mayor simplicidad. Un buen framework de arquitectura es aquel que resuelve los problemas técnicos más complejos, como por ejemplo, el manejo de transacciones, permitiendo a los desarrolladores concentrarse en el diseño y desarrollo del dominio de la aplicación [EE03].

Por otra parte, Eric Evans [EE03], hace constantemente hincapié en que un buen framework no debe ser intrusivo y que el dominio debe estar aislado, desacoplando los objetos de dominio del resto de las funcionalidades del sistema.

Podemos apreciar que Eric Evans, pone énfasis en que un buen framework es aquel que logre aislar el dominio del resto del sistema, logrando facilitar el desarrollo del mismo.

Por otra parte, como se menciona en el libro de Rod Johnson [RJ05], "u*n framework no impone demasiados requisitos para el desarrollo de código y obliga a los desarrolladores a no escribir código que no sea el apropiado. Un framework debe proporcionar una guía en cuanto a las buenas prácticas".*

Tomando a este autor, vemos una clara diferencia con Eric Evans, en cuanto a que el primero puntualiza en que un buen framework logra guiar el desarrollo respetando pautas de calidad y buenas prácticas sin hacer hincapié en el aislamiento del dominio.

Otra definición la brinda Stephen T. Albin en su libro [SA03], quien define a un framework como una estructura compuesta por partes integradas. En el mundo de la POO (Programación Orientada a Objetos) los frameworks de aplicaciones son librerías de clases.

Básicamente los frameworks incluyen los siguientes puntos de vista:

- Procesamiento: Requerimientos funcionales y casos de uso.

- Información: Modelo de objetos, diagramas de entidad relación y diagramas de flujos de datos.

- Estructura: Diagramas de componentes que representan a los clientes, servidores, aplicaciones, bases de datos y sus interconexiones.

Una arquitectura determina si estos puntos de vista son necesarios en el desarrollo de un sistema de software.

Como conclusión podemos decir que a un framework se lo puede ver como una arquitectura de software que modela las relaciones generales de las entidades del dominio, proveyendo una infraestructura y "entorno de trabajo" que extiende el dominio.

El uso de un entorno de trabajo que enmascare todas estas cuestiones facilita el desarrollo del sistema, poniendo énfasis en lo que realmente es el corazón de toda aplicación: el dominio [EE03].

1.2 Un poco de historia

Según Stephen T. Albin [SA03] el desarrollo del software nació en el año 1949 cuando la computadora (EDSAC) fue creada (es la primera computadora diseñada con la

arquitectura Von Neumann [INT52]). Los programas fueron inicialmente creados por instrucciones de código binario de máquina. Esto dificultaba la lectura de los desarrollos por parte de los programadores.

Alrededor de la década del 50, se empezaron a escribir las primeras subrutinas que permitían a los desarrolladores poder reutilizar el código. Luego a final de los 50 aparecieron lo primeros programas escritos en lenguaje de alto nivel, siendo más legible por el ser humano, que mediante otros programas, se traducían en código de bajo nivel. De esta forma, el primer cambio del paradigma ocurrió.

A mediados de los años sesenta, FORTRAN se estableció como el lenguaje dominante para la programación científica.

En 1968, apareció el término “crisis del software”, en una conferencia organizada por la OTAN sobre desarrollo de software, etiquetando de esa manera a aquellos problemas que surgían en el desarrollo de sistemas de software, tales como la no satisfacción de los requerimientos, ni las necesidades de los clientes y el exceso en los presupuestos y en los tiempos [INT40].

En ese mismo año, Edsger Dijkstra publicó un documento sobre el diseño de un sistema llamado “THE”. Éste es uno de los primeros escritos para documentar el diseño de un sistema de software usando las capas jerárquicas (división de capas), de las cuales la frase “capas de abstracción” fue derivada. Aunque el término arquitectura todavía no había sido utilizado para describir el diseño del software, ésta era ciertamente la primera aproximación a la arquitectura del software.

Al principio de los años 70, ocurrió el segundo cambio del paradigma, con la llegada de los modelos estructurados de desarrollo (1970- Algol, Simula67, 1972- Diagramas de Jackson [INT53]) y diseño del software.

En 1972, David Parnas introdujo un concepto importantísimo en el mundo del software, que es el ocultamiento de la información, siendo uno de los principios fundamentales en el diseño de sistemas de la actualidad.

Los métodos de diseño estructurado no podían escalar como la complejidad de los sistemas de software, y en la mitad de los años 80, un nuevo paradigma de diseño comenzó a tomar fuerza, el paradigma de diseño/desarrollo orientado a objetos. Si bien hay antecedentes de dicho paradigma, en la década del 60, con el lenguaje Simula 67 (una extensión del Algol) y variantes de Lisp orientado a objetos (McCarthy), en 1980 explotó el paradigma con la aparición del SmallTalk.

Con la programación orientada a objetos, los ingenieros de software podrían (en teoría) modelar el dominio del problema dentro de una lengua cotidiana. Y así aparece Smalltalk y C++ (haciendo popular a la POO). Cabe aclarar que C++ hacía demasiados compromisos de compatibilidad con C y eso le sacaba pureza "expresiva" en la modelización de objetos. En los apartados consecutivos (1.3.1 y 1.3.2) se nombrarán dos aplicaciones de la programación orientada a objetos (POO): Smalltalk como lenguaje padre de la POO y un sistema operativo llamado NEXTSTEP escrito puramente en un lenguaje POO derivado de la fusión de SmallTalk y C, llamado Objective-C [INT54].

Cerca de los años 90, el diseño del software experimenta otro cambio más, pero esta vez, hacia técnicas de diseño tales como: Tarjetas de Clases / Responsabilidades / Colaboradores (CRC) y análisis de casos de uso.

A mediados de los 90, fue requerido un nuevo enfoque que integrará diferentes ópticas de diseño de un mismo sistema, para el manejo de la complejidad de diseño y desarrollo de sistemas de software de alta complejidad. Este nuevo enfoque culminó en el desarrollo de lo que se conoce como el Lenguaje de Modelado Unificado o más conocido como UML (Unified Modeling Language, Lenguaje de Modelado Unificado), que integra conceptos de modelado y notaciones de varias metodologías. El desarrollo del UML comenzó a finales de 1994 con Grady Booch y Jim Rumbaugh que luego crearon Rational Software Corporation. Ambos comenzaron su trabajo sobre la unificación de sus métodos (el método de Booch y la OMT – Object Modeling Technique, Tecnica de Modelado de Objeto - de Rumbaugh) [INT42]. El método OMT era mejor para el análisis orientado a objetos (OOA), mientras que el método de Booch era mejor para el diseño orientado a objetos (OOD) [INT43] [INT44].

Luego, según los sitios de Internet [INT43] [INT45], Ivar Jacobson, el creador del método OOSE (Object Oriented Software Engineering, Ingeniería del Software Orientada a Objetos), se une a Rational Software Corporation. Siendo esta la primer metodología de diseño orientada a objetos dirigida por casos de uso para el diseño del software. Y así, basados en la unión primero de Booch y Rumbaugh y luego de Jacobson, al que se lo conoce "Padre" del concepto de Caso de Uso o Use Case, se crea lo que se conoce como UML.

Para concluir, hacia finales de los 90, los patrones de diseño comenzaron a ganar en popularidad como un camino para compartir conocimientos de diseño (en el apartado 1.7.3 se ampliará el tema de patrones de diseño).

Como conclusión, el autor del libro [SA03] piensa que estamos experimentando el quinto cambio, que es el reconocimiento de la arquitectura de software como un aspecto importante en el ciclo de vida del desarrollo del software.

1.2.1 Smalltalk

Según [INT30], Smalltalk-80 es un lenguaje de programación orientado a objetos, creado en parte para un uso educativo (en Xerox PARC) por Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg y otros durante los años 70, influenciados por Sketchpad y Simula.

Smalltalk-80 versión 1, fue entregado a una pequeña cantidad de compañías (Hewlett-Packard, Apple, Tektronix, y DEC) y universidades (Berkeley) para una peer review (revisión por pares) y puesta en práctica en sus plataformas. La primera máquina estándar donde corrió SmallTalk fue la Xerox Dorado, con monitor bit-mapped (mapa de bit). Se considera oportuno aclarar que La Xerox Dorado no era una máquina estándar, sino una máquina orientada a la inteligencia artificial y a la modelización (simulación de modelos gráficos), con un costo de USD 50.000 mínimo, 4 MB de RAM y 40 MB en disco. Más adelante (en 1983) sale la versión 2 de Smalltalk-80.

Squeak es una implementación open source (código abierto) derivada de Smalltalk-80, versión 1, mientras que VisualWorks deriva de la versión 2 de Smalltalk-80.

1.2.2 NeXT y NEXTSTEP

La compañía de software NeXT desarrollo un sistema operativo multitárea orientado a objetos, diseñado para correr en sus computadoras NeXT ("black boxes, cajas negras") [INT33]. NEXTSTEP 1.0 fue lanzado el 18 de septiembre 1989.

Según [INT33], NEXTSTEP es una combinación de:

- Sistema operativo Uníx.
- Display PostScript (DPS).
- Lenguaje de programación Objective-C.
- Capa de aplicación orientada a objetos.

Está íntegramente desarrollado orientado a objetos, lo que significa que los desarrolladores pueden reutilizar parte del código de una manera fácil.

Luego Sun y NeXT acordaron migrar NeXTstep a la plataforma Solaris de Sun, con lo cual el nuevo producto pasó a llamarse OpenStep.

Por otra parte, Apple consideraba las posibilidades de incluir Solaris o Windows NT a sus equipos, pero finalmente se decidió por comprar en diciembre de 1997 a NeXT y basó su generación de MAC OS X en OpenStep [INT36].

1.3 Requerimientos de una arquitectura de software

Según Rod Johnson [RJ02], toda arquitectura de software deberá cumplir con una serie de requerimientos que hacen al buen diseño de una aplicación empresarial:

- Ser robusta.
- Ser performante y escalable. Se ampliará en el apartado 1.4.1.
- Sacar ventaja de los principios del diseño orientado a objetos.
- Eliminar la complejidad innecesaria (por ejemplo, evitar el uso de servidores de aplicaciones cuando no es necesario). Se ampliará en el apartado 1.4.2.
- Ser mantenible y extendible.
- Ser fácil su testeo.

Y dependiendo de los requerimientos del negocio, deberá:

- Soportar múltiples clientes.
- Ser portable.

1.3.1 Performance y escalabilidad de las aplicaciones

Resulta de importancia distinguir entre la performance, throughput (rendimiento de procesamiento) y escalabilidad. Esto nos ayudará a comprender que la alta performance de una aplicación como así también la alta escalabilidad no siempre resultan ser correlativas. A continuación desarrollaremos una serie de definiciones extraídas del libro de Rod Jonson [RJ04] para lograr una mayor comprensión del tema.

- Performance de una Aplicación: Tiempo que ésta demanda para realizar el trabajo.

- Tiempo de respuesta: Tiempo que tarda la aplicación en procesar una petición (ej. una HTTP request – petición HTTP - desde el browser del cliente).

- Latencia: Se considera cero el tiempo de proceso, es decir, solo los demás tiempos sin importar lo que tarda la aplicación en procesar. Ejemplo, la invocación a métodos remotos presentan alta latencia.

- Throughput (rendimiento de procesamiento): Es la cantidad de trabajo que se puede realizar en una aplicación o componente en un período de tiempo dado.

- Escalabilidad: Típicamente es lo bien que maneja el aumento en la cantidad de transacciones u operaciones que una aplicación debe manejar.

    - Escalabilidad horizontal: Se refiere a incrementar el throughput por ejemplo, armando un cluster de servidores e implementando balanceo de carga.

    - Escalabilidad vertical: Se refiere a incrementar recursos al hardware ya existente, por ejemplo, añadiendo más CPU o RAM a un servidor.

Como conclusión, podemos decir que en la práctica a veces existe un conflicto o "trade-off" entre la performance y escalabilidad. Es decir, una aplicación puede ser altamente preformante en un simple servidor, pero bajar su performance más de lo lógico, cuando se intenta escalar a un cluster (o grupo) de servidores. Es decir, que tiene una escalabilidad menor que lineal.

1.3.2 Como decidir si la aplicación necesita un servidor de aplicaciones

Como se menciona en [RJ04] y en el sitio de SUN [INT62], existen una serie de ventajas al evitar el uso de un servidor de aplicaciones y se listan a continuación:

- En caso de productos propietarios, obtenemos menores costos de licencias.

- Aumenta el tiempo de inicio o startup.

- Simple administración y menor curva de aprendizaje para los desarrolladores.

- Mayor cantidad de componentes, mayor posibilidad de rotura de uno de ellos (menor MTBF o Tiempo Medio Entre Fallas o Mean Time Between Failures)

Así como también, un servidor de aplicaciones es el indicado cuando:

- Se necesiten transacciones distribuidas, por ejemplo, si se tiene acceso a múltiples bases de datos. Además es posible integrar alguna implementación JTA (Java Transaction API, API de Transacciones de Java) de terceros como JOTM (Java Open Transaction Manager, Administrador Abierto de Transacciones Java) más allá de la solución que traen integrada estos servidores.

- Se necesita la distribución de los componentes de la aplicación en distintos servidores físicos.

- Se necesita un contenedor de EJB: Cuando se necesite un contenedor donde alojar la logica de negocio.

- Se necesita un contenedor Web: Donde la capa de presentacion pueda correr.

- Se necesita JMS (Java Messaging Service o Servicio de Mensajeria de Java): Es un estandar para el manejo de colas de mensajes permitiendo una comunicación asíncrona entre los componentes.

- Se necesita autenticación y autorización: Brinda un subsistema de autenticación y autorización para el acceso a la aplicación.

- Se necesita balanceador de carga: Brinda un subsistema para distribuir las peticiones http en varios servidores de aplicaciones o servidores web aumentando la escalabilidad horizontal.

- Se necesita un sistema de manejo de transacciones distinto al del administrador de la base de datos (por ejemplo cuando se necesitan DBMS de múltiples fabricantes).

- Se necesita una administración y manejo automatico de la seguridad, multiprocesamiento, multithreaded, manejo de memoria.

- Se necesita un manejo de web services.

1.4 Principios para evaluar arquitecturas

En [RJ04] se proponen un conjunto de valores ó principios, de un carácter técnico, que nos servirán para evaluar arquitecturas de software, sus posibles implementaciones y ayudarnos a lograr proyectos exitosos. Algunos de ellos son:

a. Simplicidad

Una definición interesante de simplicidad es la que se da en los principios del manifiesto [INT17] como "*el arte de maximizar el trabajo no hecho".*

Uso de reglas de sentido común y economía, siempre usar la solución que tenga menos componentes, y componentes más básicos y que provean un comportamiento entendible y único, para encarar un problema.

Algunos criterios a seguir [RJ04]:

- Soluciones simples a problemas simples.

- Se debe poder hacer refactoring (descartar partes del código y "reescribirlo" para lograr alguna mejora - por ejemplo, mejor performance o más funcionalidad -). Las claves para el refactoring son:

    - Buen diseño OO (Orientado a Objetos).
    - Uso de Interfases.
    - Ocultar tecnologías como EJB detrás de interfases Java.

b. Productividad

Según Rod Johnson [RJ04], el desarrollo J2EE tiene y tuvo problemas de baja productividad porque los enfoques ortodoxos en la arquitectura introducen una complejidad innecesaria, que impide a los desarrolladores concentrarse en el desarrollo de la aplicación.

Para mejorar la productividad se propone, desde el punto de vista de la arquitectura:

- Evitar la complejidad arquitectural innecesaria.

- Evitar el uso innecesario de EJB's.

- Usar abstracciones para ocultar la complejidad de las APIs J2EE.

- Usar un O/R mapping (mapeador Objeto/Relacional) para simplificar la capa de persistencia (como por ejemplo Hibernate).

- Usar un buen framework de aplicaciones (como por ejemplo Spring o Pico Container).

- Usar arquitecturas de referencia y empezar las aplicaciones usando *Templates de Arquitecturas*. Por ejemplo, si vamos a desarrollar una aplicación con Spring o Struts usamos un/os template/s genérico/s.

c. Orientación a Objetos

Como se menciona en el libro de Rod Johnson [RJ04], el diseño OO es más importante que las tecnologías específicas tales como J2EE. Debemos evitar que nuestras elecciones tecnológicas restringan las posibilidades de tener un buen diseño OO. Es muy común, en las aplicaciones J2EE, la existencia de *falsos objetos:* objetos en apariencia pero que no tienen comportamiento. Por ejemplo Transfer Objects (usualmente llamados DTOs. Data Transfer Objects. Objetos para Transferencia de Datos), Value Objects (Objetos con Valor) u objetos persistentes que sólo tienen getters y setters.

d. Importancia de los requerimientos de negocio

Según Rod Johnson [RJ04], la arquitectura debería estar al servicio de los requerimientos del negocio y no determinada por la tecnología que decidamos usar. Muchas veces se asumen como ciertos los siguientes requerimientos "*fantasmas"*:

- Proveer soporte para las bases de datos distribuidas. Cuando la mayoría de las aplicaciones trabajan con sólo una BD.

- La habilidad para portar a otro servidor de aplicaciones a costo cero.

- La habilidad para portar a otra base de datos fácilmente.

Todos los requerimientos antes mencionados, son requerimientos de negocio *potenciales*, pero para cada aplicación particular debe evaluarse si son requerimientos reales. Desde luego, cuando diseñamos una aplicación debemos tener en cuenta los posibles requerimientos futuros. Pero no debemos armar

nuestras especulaciones basándonos en las capacidades de la plataforma de desarrollo, sino en las necesidades del negocio.

1.5 Aplicaciones multicapa

Toda arquitectura de una aplicación empresarial está dividida en tres grandes capas lógicas bien definidas [MF02]. De la misma manera coincide Rod Johnson [RJ02], sin embargo utilizan diferentes terminologías. Estas son:

- **Capa de acceso a los datos**
  - Según Rod Jonhson [RJ02], se denomina Integration layer (capa de integración) ó Enterprise Information System tier (capa de sistema de información empresarial).
  - Según Martin Fowler [MF02], se denomina Datasource layer (capa de origen de datos).

- **Capa de negocio**
  - Según Rod Jonhson [RJ02], se denomina middle tier (capa media).
  - Según Martin Fowler [MF02], se denomina domain tier (capa de dominio).

- **Capa de presentación**
  - Según Rod Jonhson [RJ02], se denomina User Interfase tier (capa de interfase de usuario).
  - Según Martin Fowler [MF02], se denomina Presentation layer (capa de presentación).

Antes de entrar en detalle en cada una de las capas, veremos porque es importante esta separación, como fueron evolucionando a lo largo de la historia y como afecta directamente a la arquitectura del software.

1.5.1 Evolución de las capas

Como se menciona en el libro de Martin Fowler [MF02], la noción de capas aparece en la década del noventa con las aplicaciones cliente - servidor, donde el cliente generalmente, era la interfase gráfica más la lógica de negocio (o lógica de dominio) y el servidor, era una base de datos relacional.

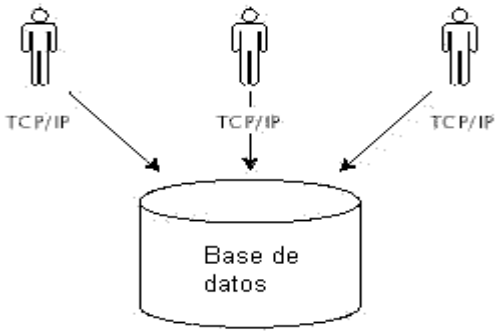

Figura 1. Clientes operando sobre una base de datos vía TCP-IP

Apareció un problema con respecto a la ubicación de la lógica de negocio, ya que la mayoría de los desarrolladores la escribían en el cliente. A medida que la lógica crecía, el código se hacía inmantenible. A esto se lo conoce como “Fat Clients” (Clientes Gordos).

Otras desventajas que podemos encontrar en este tipo de arquitecturas son [RRR03]:

- Implementar la seguridad es más difícil ya que los algoritmos y la lógica se encuentran del lado del cliente.

- Aumenta los problemas de mantenimiento y actualización del sistema.

- Problemas con las versiones de los clientes (distintas versiones de la aplicación cliente).
- Lógicas complicadas y complejas demandan clientes robustos y pesados.

Como una primera solución a algunos de estos problemas, se optó por colocar la lógica de negocio en el servidor (reduciendo la complejidad en la distribución de las aplicaciones), más específicamente como, stored procedures (procedimientos almacenados) en la base de datos. Sin embargo estos tienen ciertas limitaciones que mencionaremos a continuación.

- Tienen baja **performance** en problemas distintos que el tratamiento de los datos.
  Si bien, en un procedimiento almacenado individual, la ejecución puede ser más rápida a su equivalente en código, éstos pueden tener un efecto adverso en cuestiones de escalabilidad. Al estar embebidos en la base de datos, dependen exclusivamente de la potencia del servidor de base de datos. Es decir, si bien en algunos casos son más rápidos, tienen riesgos severos de escalabilidad de CPU. Roland Bouman [INT34] [INT35]. Por otra parte, los SPs pueden ser muy performantes, en algunos casos, más que las aplicaciones Java. Los problemas de performance y conflictos con otras aplicaciones modernas pueden ser menores en el caso de arquitecturas como J2EE.

- Afectan negativamente al **rendimiento** de otras aplicaciones que utilizan el mismo servidor de base de datos [INT35].

- Dificulta la migración a sistemas de administración de bases de datos de otros proveedores ya que cada motor de bases de datos tiene su propio lenguaje interno de procedimientos almacenados, la mayoría de ellos propietarios [INT34] [INT35].

La mejor excusa para evitar los SPs es el hecho de que no están integrados en un único ambiente OOP, y muchos de ellos (TRANSACT SQL por ejemplo) no son orientados a objetos.

Al mismo tiempo que ganaban en popularidad las arquitecturas cliente-servidor, también comenzaba a aparecer la programación orientada a objetos. Y ésta última fue la que tuvo la respuesta para evitar el problema de la complejización excesiva de la lógica de negocio, creando 3 capas: La capa de presentación para lo que respecta a interfase de usuario, la capa de negocio para la lógica de dominio y la capa de acceso a los datos.

Luego con la llegada de Internet, toda la gente repentinamente quiso que sus aplicaciones cliente-servidor funcionaran con un navegador Web. Sin embargo, si toda la lógica de dominio estaba del lado del cliente (rich clients, clientes ricos), ésta debía ser desarrollada de nuevo para poder soportar clientes Web. Así las aplicaciones de 3 capas, comenzaron a ganar popularidad ya que la lógica se trasladaba al servidor.

En 1995, los sitios Web estaban programados puramente en HTML donde el contenido de las páginas no sufría demasiadas actualizaciones, pero con el tiempo fue ganando popularidad el lenguaje PHP, haciendo los sitios muchos más dinámicos, cambiando la manera de interactuar de los usuarios con las paginas Web. Es decir, dejaron de ser simples sitios informativos (con contenido estático) para ser participativas. Aquí es donde nace el concepto de la WEB 2.0. Se puede decir que es una vuelta a los "Fat Clients" (Clientes Gordos), si bien no son tan "gordos" como antes, no son "Thin Clients" (Clientes Delgados) clásicos [BG].

Según el sitio de Internet [INT55], la Web 2.0 *es la transición que se ha dado, de aplicaciones tradicionales de escritorio hacia aplicaciones que funcionen a través de la Web*. Mediante toda esta evolución, surge lo que se conoce como AJAX (Javascript Asincrónico y XML – Asynchronous Javascript and Xml), el cual engloba varias tecnologías como ser XHTML-CSS, DOM (Document Object Model, Modelo de Objetos de Documentos) y XML [BG].

En la actualidad la Web 2.0 implica ciertas tecnologías como ser [BG] [INT55]:

- Transformar software de escritorio hacia aplicaciones web.
- Respetar a los estándares de XHTML.
- Uso de hojas de estilo (CSS)
- Sindicación de contenidos.
- Ajax
- Flash, Flex o Lazlo.
- Uso de Ruby on Rails u otra herramienta equivalente para programar páginas dinámicas

Como corolario de esta sección, se nombrarán a modo de ejemplo algunos servicios que marcan una clara evolución hacia la Web 2.0 [BG] [INT55]:

- Wikipedia (Enciclopedias)
- Blogs (Páginas personales)
- Google AdSense (Servicios de Publicidad)
- BitTorrent (Distribución de contenidos)
- Flickr (Comunidades fotográficas)

1.5.2 Separación en capas

Cuando uno piensa en términos de capas, lo que se tiene que tener en cuenta es, que éstas esconden su lógica al resto de las mismas y solo brindan puntos de acceso a dicha lógica. Es decir, una capa solo brindará servicios a la subsiguiente, mientras que la de más bajo nivel no conocerá nada de dicha capa. Por ejemplo, suponiendo que la capa 4 es la de más alto nivel (más cerca de la presentación que del acceso a los datos), ésta usará los servicios proporcionados por la capa 3, mientras que a su vez ésta última, usará los servicios de la capa 2, pero la capa 4 no conocerá nada de la capa 2 [MF02].

Ventajas:

- Permite sustituir capas por otras implementaciones alternativas, por ejemplo, se podrá sustituir la capa de presentación implementada con tecnologías "web" por otras implementadas con tecnologías "swing" [MF02]. Esto permite que problemas complejos sean distribuidos en pequeñas piezas más manejables [INT07].

- Las capas superiores brindan servicios a las capas inferiores permitiendo la rehusabilidad del código [NT07].

- Minimiza las dependencias entre las capas. Si se quiere cambiar por un framework de O/R mapping (Object / Relational mapping, mapeos del modelo relacional a objetos), ésto no impactará en las capas superiores [MF02].

- No es necesario conocer más capas que la que se está viendo y las "lindantes". [MF02].

- Los detalles de implementación de cada capa son escondidos para el resto [INT07].

Desventajas:

- Si sobrecargamos a la aplicación empresarial con capas, ésta perderá performance [MF02].

- Tienen el problema de los "cambios en cascada". Por ejemplo, en una típica aplicación empresarial que muestra un dato en su UI (User Interfase, Interfase de Usuario), que viene desde la base de datos, si este cambia o se modifica, este cambio impacta en todas las capas por las que pasa el dato [MF02].

Se considera necesario aclarar que la separación en capas puede conducir al uso indebido de las mismas. Si bien no se puede considerar como una desventaja en si misma, se considera oportuno entrever que su mal uso produce una violación del encapsulamiento: Como las capas ocultan los detalles de implementación al resto de las mismas, una violación de esto puede "costar caro", ya que si una capa (ej. capa A) usa directamente los detalles de implementación que se encuentran en otra (ej. capa B), cuando se produzcan cambios en la capa B también se verá afectada la capa A [INT07].

Según Martín Fowler en su sitio de Internet [INT06], hace referencia a algunos principios básicos de la separación en capas a través de una puntuación.

Los principios con mayor puntuación son:

- Bajo acoplamiento entre las capas, alta cohesión dentro de ellas.
- Separación de responsabilidades.
- Adaptabilidad: permite el cambio de una capa.
- Las capas UI o de presentación no contienen lógica de negocio.

- Las capas de negocio no contienen lógica de presentación ni hacen referencia a la capa de presentación.
- No existen las referencias circulares entre las capas.
- Las capas deberán ser testeadas individualmente.
- Capas inferiores no deben depender de capas superiores.
- Las capas pueden compartir aspectos de infraestructura como ser aspectos de seguridad.

A sí mismo, Eric Evans en su libro [EE03], explica que la separación en capas permite: un diseño cohesivo (donde las dependencias están hacia las capas inferiores), reduciendo el acoplamiento entre las capas superiores y por último, aislar el modelo de dominio del resto de las capas, permitiendo, escribir código relacionado únicamente con el dominio de la aplicación.

Como corolario de este apartado, se puede distinguir, que tanto Eric Evans como Martin Fowler coinciden en sus principios [INT06], dándole Eric Evans la importancia de aislar el dominio del resto de la aplicación.

1.5.3 Capas lógicas

Como se menciona en el libro de Martin Fowler [MF02], existen 3 capas principales:

- Capa de presentación.
- Capa de negocio.
- Capa de acceso a datos.

Esta separación, mencionada por Martin Fowler [MF02], está presente en la mayoría de los desarrollos Web por capas. Esto no quiere decir que no tengamos más capas, sino que la cantidad de las mismas va a depender del sistema a desarrollar. Es decir, existen grandes variaciones en cuanto a la cantidad de capas que se utilizan en los distintos desarrollos de aplicaciones Web. La mayoría de las arquitecturas utilizan las siguientes 4 capas conceptuales [EE03].

La siguiente descripción de arquitectura en capas, se encuentra en el Capítulo 4 de [EE03]. Lo interesante de esta clasificación es que le otorga una entidad específica a la *Capa de Dominio* (pone énfasis en la misma).

La interfase de usuario

Esta capa se encargará de resolver como exponer la lógica de negocio a través de una interfase. La interfase más común es la Web. Por lo que se deberá tener en cuenta los límites de la distribución de los objetos entre el browser (navegador) - cliente - y el servidor web.

La capa de aplicación

Esta capa no contiene reglas de negocio. Será la encargada de coordinar y delegar trabajo a los objetos de dominio de la capa subsiguiente.

Muchas veces, esta capa, no hace el trabajo de coordinar y delegar, sino que tiene lógica de negocio incorporada, y esto conlleva a tener un Modelo de dominio anémico (será ampliado en el apartado 1.6.3.2).

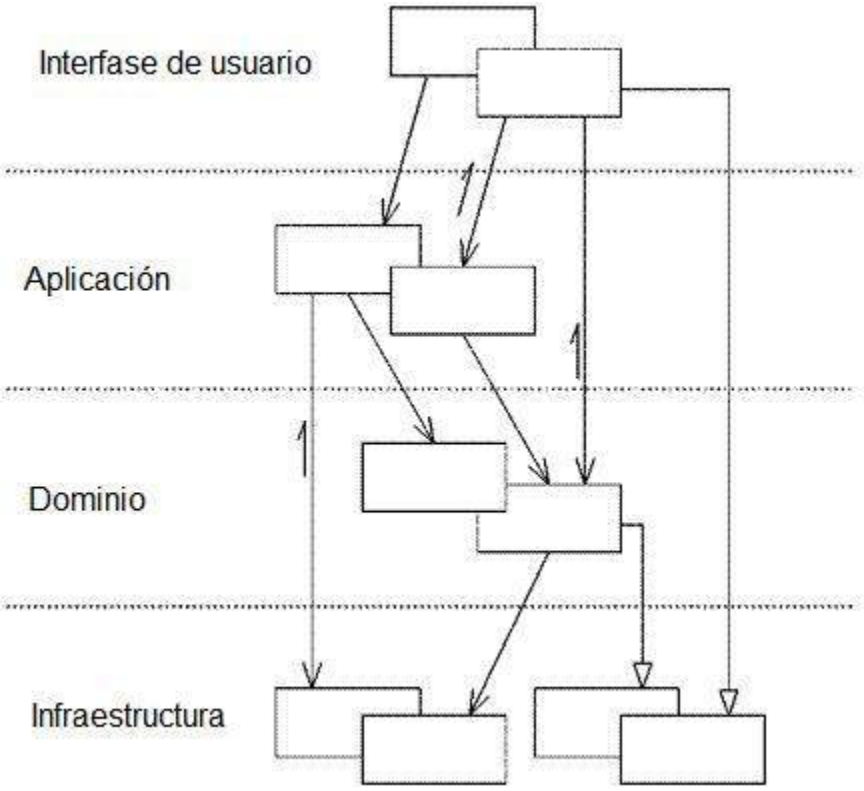

Figura 2. Arquitectura de capas. [EE03]

La capa de dominio (o capa de modelo)

También conocida como “capa de negocio". Se encargará de exponer la lógica de negocio a las capas de más alto nivel. Aquí encontraremos las reglas de negocio. Esta capa es el corazón de la aplicación y en ella se representarán:

- Conceptos de negocio.
- Reglas de negocio.
- Información acerca de la situación del negocio.

La capa de infraestructura

Esta capa será la encargada de proporcionar las capacidades técnicas genéricas que apoyan las capas más altas:

Provee la persistencia para el dominio. Ya sea, cómo establecer la conexión, cómo acceder a la BD y finalizar la conexión. Básicamente en una aplicación empresarial, las tecnologías de acceso son varias y a continuación citaremos algunas de ellas:

- Vía JDBC que viene con la API de Java.

- Utilizando algún framework de O/R Mapping: Como Hibernate, JDO (Java Data Object), TopLink, iBattis, OJB de Apache.

Es decir, soportará la interacción entre las capas por medio de un framework de arquitectura.

Como conclusión, algunos proyectos no tienen la distinción entre las capas de interfase de usuario y aplicación. Otros tienen múltiples capas de infraestructura. Pero lo importante es la separación de la capa de dominio que permitirá lo que se conoce como MDA (Model Driven Design, Diseño Dirigido por el Modelo) [EE03].

1.5.4 Capas lógicas VS físicas

La separación en capas vista hasta el momento es lógica, es decir, estamos particionando al sistema en pequeños bloques para reducir el acoplamiento entre diferentes partes del mismo. Pero no quiere decir que todo el sistema siempre este en un mismo equipo. Puede que diferentes partes de la aplicación se distribuyan en diferentes servidores y requieran de un acceso remoto a dichos componentes [RJ04].

En el siguiente capítulo, veremos como se distribuyen las capas dependiendo de las tecnologías utilizadas y nos centraremos exclusivamente en las arquitecturas J2EE y más específicamente en las arquitecturas de contenedores ligeros (Lightweight Container). Pero antes de interiorizarnos en las arquitecturas J2EE (EJB) y las arquitecturas de contenedores ligeros, analizaremos como influye y se relacionan los patrones de diseño y la arquitectura del software.

1.6 El diseño arquitectónico del software

Según Stephen T. Albin [SA03], la experiencia demuestra que como el tamaño y la complejidad de las aplicaciones y los equipos de desarrollo crecen, surge la necesidad de un mayor control en el diseño de las aplicaciones. Se necesita un mejor control del proceso del diseño del software para mejorar la calidad del producto y realizar predicciones más exactas de la cantidad de esfuerzo requerido para desarrollar un sistema.

Los obstáculos que se nos presentan para alcanzar un diseño arquitectónico de alta calidad en el desarrollo del software son [SA03]:

- Comunicación pobre entre o con los stakeholders. Siendo uno de los más importantes.

- Carencia de conocimiento de la importancia del diseño arquitectónico en el desarrollo del software.

- Carencia de la comprensión del papel del arquitecto del software para el caso de metodologías monorol.

- Una visión extendida propone que el diseñar es una forma de arte, no una actividad técnica. Lo es hasta cierto punto, pero lo que hacemos aquí es tratar de dilucidar qué cosas no son parte del arte y sí pueden abstraerse e integrarse en un patrón técnico.

- Carencia de la comprensión del proceso del diseño.

- Falta de experiencia en diseño.

- Carencia en la comprensión de cómo evaluar diseños.

1.6.1 Los patrones y la arquitectura del software

Según Frank Buschmann et al. [BMRSS96], un criterio importante para el éxito de los patrones de diseño, es lo bien que resuelven los objetivos de la ingeniería del software. Los patrones deben apoyar el desarrollo, el mantenimiento y la evolución de los sistemas, además de apoyar eficazmente la producción industrial del software.

1.6.2 Un poco de historia

Durante 1970 un arquitecto llamado Christopher Alexander realizó el primer trabajo en el área de patrones. En un intento por identificar y describir la integridad de los diseños de calidad, Alexander y sus colegas estudiaron diversas estructuras, que fueron diseñadas para solucionar el mismo problema. Identificó similitudes entre los diseños de alta calidad. Entonces usó el término patrón en varios de sus libros para referirse a estas similitudes [PK04].

Por otra parte, como se menciona en [MF97], mucha gente niega a Alexander su papel central como inspirador para los patrones del software. Peter Goad precisa que la noción de patrones es utilizada por muchos escritores en otros campos, muchos de quién él piensa son mejores ejemplos que Alexander.

Además, el libro de Gamma et al. [GHJV95] ha tenido mucha más influencia en el campo de los patrones del software que el trabajo de Alexander.

- A Pattern Language: Towns, Buildings, Construction (Oxford University Press, 1977).
- The Timeless Way of Building (Oxford University Press, 1979).

En 1987, influenciado por las escrituras de Alexander, Kent Beck y Cunningham aplicaron las ideas arquitectónicas de patrones para el diseño y el desarrollo del software. Emplearon algunas de las ideas de Alexander para desarrollar un conjunto de patrones que sirvieron para el desarrollo de elegantes interfases de usuario en Smalltalk [PK04].

En 1994, con la publicación del libro *Design Patterns: Elements of Reusable Object-Oriented Software on design patterns* por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides se explicó la utilidad de los patrones. Desde entonces, muchos otros libros han publicado sobre patrones de diseño y otras buenas prácticas para la ingeniería del software [PK04].

1.6.3 Patrones vs. Anti-patrones

Según Frank Buschmann et al. [BMRSS96], un patrón describe un problema particular de diseño que se repite y se presenta en contextos específicos de diseño, y propone un esquema genérico bien probado como solución.

Como se menciona en [SJM02], "*un patrón es un camino para hacer algo"*. Es decir, es una descripción bien conocida a un problema que suele incluir los siguientes aspectos:

- Descripción del problema.
- Contexto de la aplicación de ese patrón o escenario de uso.
- Solución concreta.
- Las consecuencias de utilizar ese patrón.

Mientras que los antipatrones son soluciones negativas que no presentan una solución ideal para un problema particular. Conocer los antipatrones permite evitarlos o recuperarse de ellos, ya que presentan otro problema más que una solución al mismo [INT16].

Como conclusión podemos decir que:

- Un buen patrón explica una solución a un tipo de problema que ocurre una y otra vez.

- Un buen antipatrón explica porque la solución original que parece ser la adecuada, se vuelve negativa causando efectos no deseados a la aplicación y como recuperarse de los problemas que éste genera.

A continuación, se darán dos ejemplos: el primero es un popular patrón de diseño y el otro es un antipatron bastante común en toda aplicación empresarial.

1.6.3.1 Patrón: MVC (Modelo Vista Controlador)

Según lo explica Martin Fowler, tanto en su sitio de Internet [INT28], como en su libro [MF02], el MVC (Model View Controller, Modelo Vista Controlador) es un patrón que tuvo sus inicios por los años '70 para la plataforma Smalltalk y desde entonces ha ejercido mucha influencia en los frameworks de presentación (UI frameworks).

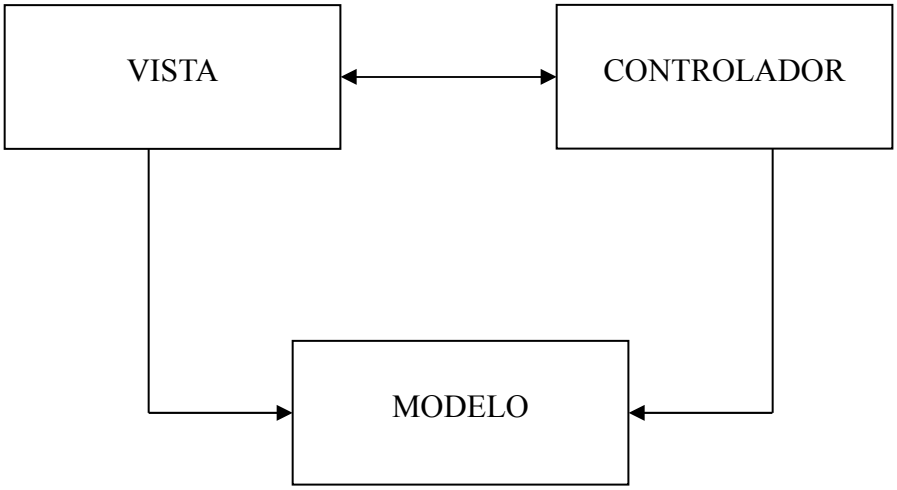

Figura 9. Patrón MVC. [MF02]

En el grafico se observan dos separaciones principales, una es, la separación entre la vista y el modelo y la otra es, entre la vista y el controlador [MF02].

Con respecto a la primera separación:

- Se puede testear el modelo de dominio independientemente de la UI.

- Se pueden tener múltiples vistas para el mismo modelo.

- La dirección de la dependencia: La vista o presentación depende del modelo y no al revés.

Con respecto a la segunda separación, si bien no es tan importante como la primera, nos permite realizar lo siguiente:

- Si se desea desarrollar un alta y una modificación sobre un mismo formulario, esto se puede llevar a cabo con una vista y dos controladores, donde estos serían la estrategia de la vista.

A su vez, Rod Johnson explica en su libro [RJ02], las diferencias entre los tres componentes, poniéndole igual énfasis en cada una de las separaciones, a diferencia de Martin Fowler que observa una acentuada diferencia entre los componentes de la vista y del modelo.

- El **modelo**: Corresponde al modelo de dominio. No contiene código de presentación.

- El **controlador**: Reacciona ante las peticiones del cliente desde la UI y actualiza el modelo a un estado consistente.

- La **vista**: Realiza la presentación de los datos del modelo.

Este patrón arquitectónico se implementó en muchos frameworks como por ejemplo en Struts. Struts, como lo dice Rod Johnson [RJ02], tiene una implementación estándar de este patrón, minimizando la cantidad de código a escribir.

1.6.3.2 Anti-patrón: Dominio anémico

Es uno de los anti-patrones que existe desde hace tiempo. Este modelo no permite colocar lógica de negocio en los objetos de dominio. En lugar de esto, existen una serie de servicios que capturan toda esa lógica y los objetos de dominio solo se encargan del manejo de datos (objetos sin comportamiento).

Lo más gracioso de este anti-patrón es que se contradice con la idea básica del diseño orientado a objetos, la cual es combinar datos y procesos en objetos. Básicamente, un sistema que tenga un dominio anémico es un sistema con diseño procedural más que orientado a objetos [INT19].

Según Eric Evans en su libro [EE03], la capa de servicios o de aplicación, se debe mantener *fina*. Es decir, esta capa no contiene reglas o conocimiento de negocio, solamente deberá coordinar las tareas y delegar trabajo a los objetos de dominio que se encuentran en una capa inferior.

A la capa de dominio o capa de negocio Eric Evans [EE03] la define como que es la responsable de representar los conceptos y reglas de negocio e información de la situación.

Como conclusión, podemos decir que para evitar caer en este modelo (anti-patrón), la capa de servicios deberá permanecer *fina* y toda la lógica de negocio deberá estar en la capa de dominio, donde residen los objetos de negocio, como bien lo plantea Eric Evans.

1.6.4 Patrones GOF vs. Patrones GRASP

Es necesario introducir conceptualmente, como los patrones de diseño impactan sobre las arquitecturas y ayudan al proceso de desarrollo de software.

Como se menciona en [GHJV95], los patrones de diseño hacen más fácil reutilizar diseños y arquitecturas. Ayudan a elegir las alternativas de diseño que hacen a un

sistema reutilizable y evitan las alternativas que comprometen la rehusabilidad. Los patrones de diseño pueden incluso mejorar la documentación y el mantenimiento de sistemas existentes.

A continuación se presentan los 23 patrones extraídos del libro conocido como Design Patterns, Elements Of Reusable Object Oriented Software (Patrones de diseño, Elementos del software orientado a objetos reutilizables) [GHJV95]. Puesto que el libro fue escrito por 4 autores - Gamma, Helm, Johnson y Vlissides en 1995, estos patrones se los conoce como los patrones GoF (gang-of-four, pandilla de los cuatro):

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy
- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Flyweight
- Observer
- State
- Strategy
- Visitor

Mientras que por otro lado tenemos a los patrones GRASP (General Responsibility Assignment Software Patterns, Patrones de Software para la Asignación de Responsabilidad).

Como se menciona en el libro de Craig Larman [CL03], estos patrones constituyen un apoyo para el diseño de un objeto. Se basan en la asignación de responsabilidades. Básicamente las responsabilidades están relacionadas con las obligaciones que tiene un objeto en cuanto al hacer y conocer.

Los patrones GRASP, no compiten con los patrones de diseño (patrones GoF) sino que nos ayudan a encontrar clases de objetos y sus responsabilidades (que hacen y a quien conocen).

A continuación se nombrarán los patrones GRASP extraídos del libro de Craig Larman [CL03] y del sitio de Internet [INT18]:

- Bajo Acoplamiento
- Alta Cohesión
- Experto
- Creador
- Controlador
- Polimorfismo
- Fabricación Pura
- Indirección
- No hables con extraños

1.7 Arquitecturas y técnicas de desarrollo: MDA, SOA y TDD

1.8.1 SOA: Service Oriented Architecture (Arquitectura Dirigida por Servicios)

Según la enciclopedia digital online wikipedia [INT22], la arquitectura dirigida por servicios expresa una perspectiva de la arquitectura de software, que define el uso de servicios de software con bajo acoplamiento para soportar los requerimientos de negocio.

Como se menciona en [RRR03], SOA es la última evolución de sistemas distribuidos, que permite que componentes de software, incluyendo funciones, objetos, y procesos de diversos sistemas, se expongan como servicios. Los servicios Web (web services) se basan en esta arquitectura. Según la investigación de Gartner (el 15 de junio de 2001), los "servicios Web son componentes de software débilmente acoplados a las excesivas tecnologías existentes en Internet".

1.8.2 MDA: Model Driven Architecture (Arquitectura Dirigida por Modelos)

Como se menciona en el sitio de IBM [INT20] y [INT21], la OMG (Object Managment Group, Grupo Encargado de Objetos) creó un conjunto de conceptos y estructuras que ayudan al desarrollo de una aplicación empresarial. Básicamente separa las decisiones del negocio, de las decisiones arquitectónicas. Es decir, marca una clara diferencia entre el dominio y la arquitectura a emplear. A esto la OMG lo llama Model Driven Architecture o MDA.

Las arquitecturas dirigidas por modelos son un estándar de la OMG que nos permite trasladar modelos de negocios concretos al desarrollo de la aplicación.

En la ingeniería de software, el modelado de aplicaciones se puede llevar a cabo mediante una metodología poderosa llamada UML la cual nos permite a través de diferentes técnicas, capturar las características más importantes del sistema de software.

Algunas herramientas del mercado:

- Herramientas open source
  - JAMDA (http://jamda.sourceforge.net/),
  - ModelJ (http://modelj.sourceforge.net/download.html).
  - Atlas (http://atlas-mda.org/confluence/display/WEBSITE/Home)

- Herramientas pagas:
  - RapidJ (http://www.codecanvas.com.au/rapidj/)

Podemos ver una demostración gratis del poder de estas arquitecturas, en especial el RapidJ, en http://www.codecanvas.com.au/rapidj/demonstration.php

1.8.3 Test-Driven Development (Desarrollo Dirigido por los Test)

Según el sitio de Internet [INT24], es una técnica de programación que combina lo que se conoce como *Test-first development* (primero se desarrolla el test) con la técnica de refactoring (modificar el código). Es promocionada por la metodología ágil de desarrollo de software llamada eXtreme Programming (XP) ó programación extrema en su regla número 10 [INT23].

En lugar de codificar el código funcional en primer lugar y luego generar el test unitario, esta técnica sugiere emprezar al revés, es decir, con el test unitario y luego con el código funcional. Una de las mayores ventajas, es que nos permite pensar en términos de diseño antes de escribir el código. Gracias a esto obtenemos código testeado (siendo de gran utilidad cuando participan varios del desarrollo de una aplicación) [INT24] y nos permite generar código de alta calidad [MR05]. A continuación se mostrará un ejemplo del funcionamiento de esta técnica:

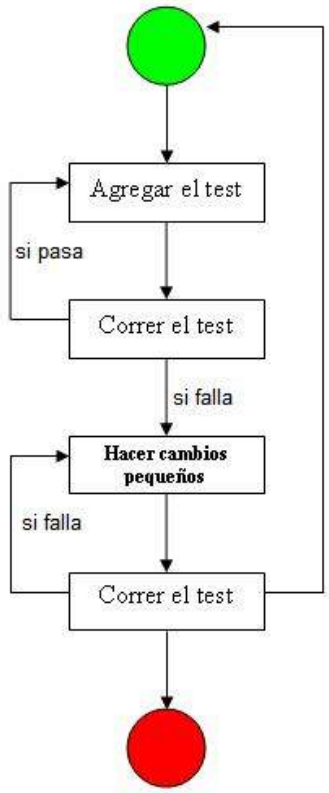

Figura 10. Diagrama de actividades – UML. [MR05]

Como bien se observa en el grafico, esta técnica exige, primero, el desarrollo del test que se corresponderá con una pequeña porción de código funcional y así se irá avanzando mediante la técnica de refactoring. No admite agregar código a menos que exista previamente el test unitario.

Como se menciona en el libro de [MR05], el desarrollo dirigido por test, se podrá llevar a cabo en las arquitecturas de contenedores ligeros (específicamente usando el framework Spring) de la siguiente manera: (Esto será ampliado en el capítulo sobre Testing en arquitecturas de contenedores ligeros).

- Para testear la capa de acceso a los datos, es decir, para testear los DAOs:
  - DBUnit.

- Para testear la capa de servicios:
  - EasyMock.
  - jMock.

- Para testear los controladores:
  - StrutsTestCase.
  - Spring Mocos.
  - Cactus.

- Para testear la capa de presentación:
  - Canoo.
  - jWebUnit.

1.9 Transacciones

Como se menciona en [INT25], el manejo de transacciones es uno de los requerimientos más importantes en el desarrollo de toda aplicación empresarial.

Según Kevin Mukhard et al. [MLC01], el propósito principal de las transacciones es, llevar la base de datos de un estado consistente al siguiente, manteniendo la integridad de los datos (todas sus reglas y comprobaciones).

Según el sitio de Oracle [INT37], una transacción es un conjunto de operaciones atómicas (alta, baja, modificación o consulta) de una base de datos que se trata como si fuera una operación atómica, manteniendo por completo la integridad y consistencia existente.

Las principales sentencias de control de transacciones son [MLC01]:

- Commit: Convierte en permanentes todos los cambios realizados en la base de datos.

- Rollback: Devuelve la base de datos al estado que tenia después de la última operación realizada con éxito.

- Begin Transaction: Representa un punto en el que los datos a los que hace referencia una conexión son lógica y físicamente coherentes. Cada transacción dura hasta que se completa sin errores y se ejecuta la sentencia *commit* [INT111].

1.9.2 Propiedades de las transacciones

Generalmente se conoce a estas cuatro propiedades como ACID (Atomicity, Consistency, Isolation and Durability, Durabilidad, Aislamiento, Consistencia e Indivisibilidad) [MLC01].

- **Atomicidad**: La transacción es tratada como una sola unidad en ejecución (ya sea que se trate de una sola sentencia SQL (Structured Query Language, Lenguaje Estructurado de Consultas) o varias.

- **Consistencia**: Cuando se completa la transacción, ésta deja a la base de datos en un estado consistente.

- **Aislamiento**: Esta cuestión es pertinente solo para el acceso concurrente a una base de datos. El aislamiento se refiere a la separación bien definida entre cambios a la base de datos que ocurren en transacciones diferentes. En la siguiente sección, se ampliará este concepto.

- **Durabilidad**: Cuando se completa una transacción, los cambios realizados por ésta permanecen.

1.9.3 Niveles de aislamiento de las transacciones

Como se menciona en [MLC01], el aislamiento, se refiere al grado en que las acciones realizadas por una transacción puedan ser vistas por otras transacciones.

- El nivel más elevado se refiere a que ninguna acción realizada por una transacción pueda ser vista por otras transacciones, hasta que ésta termina (con rollback o commit). Se llama lectura repetible.

- El nivel más bajo de aislamiento se refiere a que toda acción realizada por una transacción (haya sido efectiva o no) pueda ser vista por otras transacciones. Se llama lectura sucia.

El estándar ANSI/ISO SQL-92 identifica 3 tipos diferentes de interacciones entre transacciones desde el nivel más bajo de aislamiento hasta el más alto. Estas son:

- Lectura sucia.
- Lecturas fantasmas.
- Lecturas no repetibles.

Así mismo, el estándar, especifica cuatro niveles de aislamiento que indica que interacciones (las tres mencionadas anteriormente) son permitidas. Estos son:

- Lectura no confirmada.
- Lectura confirmada.
- Lectura repetible.
- Serializable.

Nivel de aislamiento	Lectura sucia	Lectura no repetible	Lectura fantasma
Lectura no confirmada	*permitida*	*permitida*	*permitida*
Lectura confirmada		*permitida*	*permitida*
Lectura repetible			*permitida*
Serializable			

1.9.4 Transacciones distribuidas vs. locales

Según Kevin Mukhard et al. [MLC01], las aplicaciones empresariales J2EE con EJB y sin EJB tienen dos opciones de gestión de transacciones:

- **Transacciones locales**: Transacciones que conllevan a una única conexión con una única base de datos.

- **Transacciones distribuidas**: Transacciones que abarcan múltiples fuentes de datos.

Ambos tipos de transacciones conservan las cuatro propiedades ACID descriptas en la sección 1.9.2.

Las transacciones distribuidas son gestionadas por el servidor de aplicaciones (en el caso de las aplicaciones J2EE con EJB) usando JTA [INT26] (Java Transaction API , Interfaz API para las transacciones en Java).

Mientras que las transacciones locales son especificas de un recurso, como por ejemplo, una transacción asociada a JDBC.

1.9.5 Transacciones declarativas vs. programáticas

Transacción mediante programación o programaticas. El propio bean es el que demarca la transacción. En las aplicaciones J2EE con EJB esta forma normalmente se llama BMT (Bean-Managed Transaction, Transacción Manejada por los Beans) [INT27]. Para las arquitecturas de contenedores ligeros (sin EJB) se explicará en el capitulo 3.

Transacción en forma declarativa. El desarrollador solamente debe establecer en forma declarativa como quiere que el contenedor maneje las transacciones. En las aplicaciones J2EE con EJB esta forma lleva el nombre de CMT (Container-Managed Transactions, Transacción Manejada por el Contenedor) [INT27]. Para las arquitecturas de contenedores ligeros (sin EJB) se explicará en el capitulo 3.

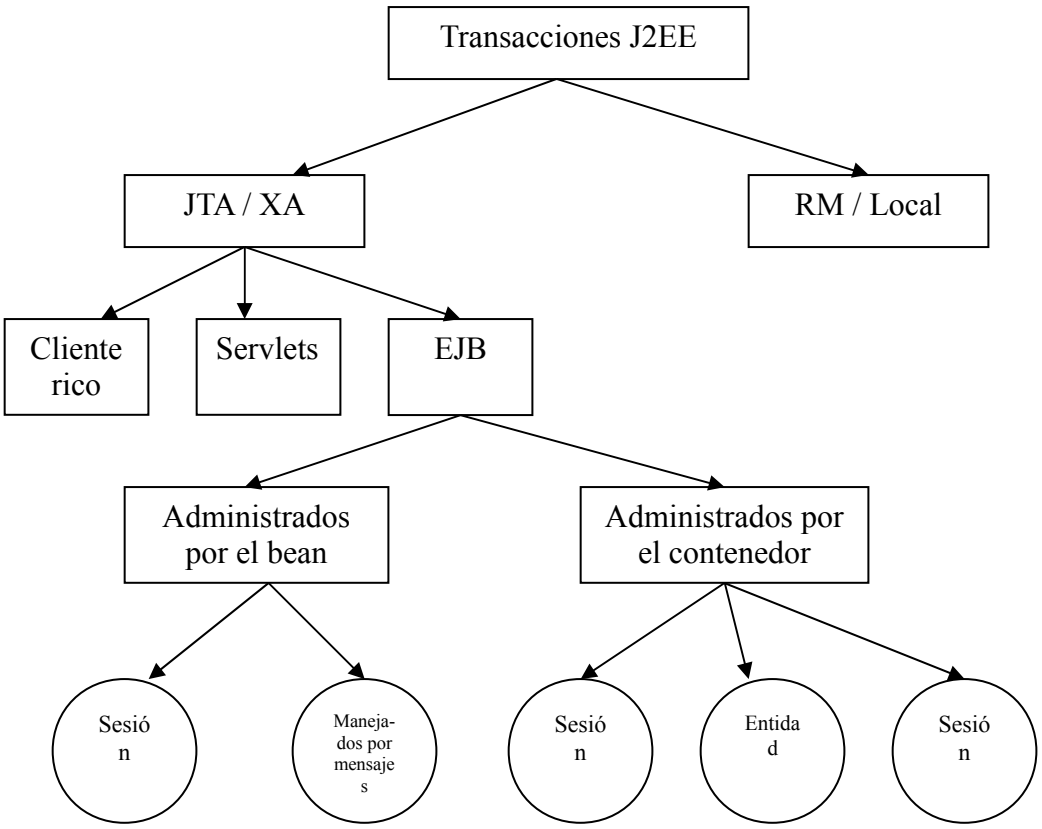

Figura 11. Transacciones J2EE

1.10 Pool de conexiones, fuentes de datos y servicios de directorios JNDI

Como se menciona en el libro de fundamentos de bases de datos con Java [MLC01], toda aplicación empresarial o aplicación multicapa hace uso de DataSources (fuentes de datos) con pool de conexiones para el acceso a la base de datos.

Existe un método que nos permite configurar los parámetros de conexión a la base de datos en un solo lugar de la aplicación y sin necesidad de recompilar la misma para que tome los cambios. A ésto se lo conoce como fuentes de datos o DataSources.

Un objeto DataSource se crea y se gestiona al margen de la aplicación, por ende, podemos modificar dicho objeto para accesar a distintas bases de datos sin realizar un solo cambio en el código [MLC01]. En otras palabras, la fuente de datos o DataSource

oculta los detalles de la conexión a la base de datos, de modo que los programadores no tengan que preocuparse por cuestiones de puertos, URL de conexión, etc.

Además de proporcionar una fuente única para la conexión a la base de datos, las fuentes de datos posibilitan la utilización de pool de conexiones y transacciones distribuidas. Trabajar con un pool de conexiones es mantener un conjunto de conexiones (configurables). Cuando un pedido requiere una conexión, la solicitará a este conjunto y cuando no la va a usar más, la devuelve al pool.

De esta forma nos estamos ahorrando el consumo de tiempo de conexión y desconexión aumentando el rendimiento de la aplicación [INT29].

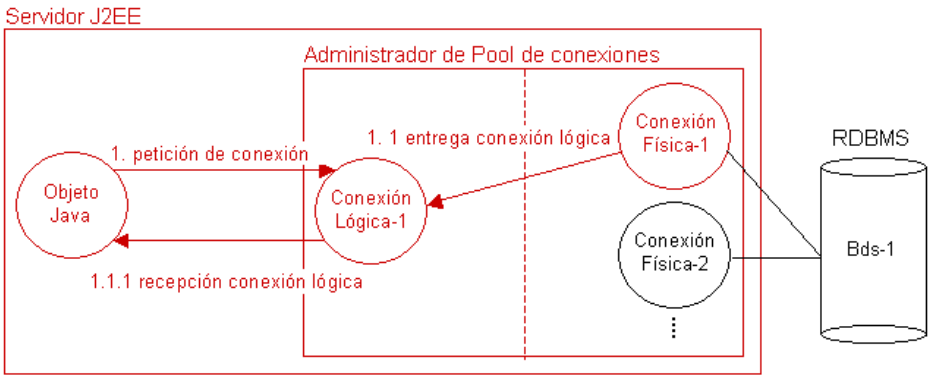

Figura 12 Utilización del pool de conexiones.

Existen 3 tipos diferentes de DataSources [MLC01]:

- **DataSource Básico**: Implementa la interfase javax.sql.DataSource. Una implementación básica no utiliza pool de conexiones ni transacciones distribuidas. Quizás es la más costosa en términos de procesamiento, ya que los objetos “Connection” deben establecerse desde cero cada vez que se desea acceder a la base de datos.

- **DataSource con soporte para pool de conexiones**: Permite la reutilización de los objetos “Connection” en lugar de crearlos desde cero, ofreciendo un mayor rendimiento. Este tipo de fuente de datos implementará la

interfase javax.sql.ConnectionPoolDataSource, donde cada fabricante de base de datos se encargará de implementar la interfase antes mencionada.

- **DataSource con soporte para transacciones distribuidas**: Este tipo de fuente de datos produce objetos “Connection”, que pueden operar con transacciones distribuidas, es decir, una transacción en la que interviene más de una base de datos. Implementará la interfase javax.sql.XADataSource.

Como se menciona en [MLC01], existen dos maneras de crear el DataSource o bien desde el código mediante el API de Java, o por medio de algunas herramientas que proporcionan los servidores de aplicaciones (en caso de las arquitecturas J2EE con EJB). Esta segunda opción, creará y registrará de manera automática la fuente de datos.

A su vez, en las aplicaciones empresariales, los DataSource suelen obtenerse realizando una búsqueda en un contexto. Básicamente un contexto es un directorio. Un contexto es un medio de asociar un nombre a un recurso. En este caso, se asociará una instancia del DataSource con el directorio.

Para poder realizar estas uniones entre nombres y recursos, se utiliza JNDI - Java Naming and Directory interfase - [MLC01].

Capítulo 2 - Arquitecturas con EJB vs. Arquitecturas sin EJB -

Toda arquitectura J2EE – EJB (Enterprise Java Beans, Java Bean Empresarial) necesitará de un servidor de aplicaciones (EJB Container, Contenedor EJB), mientras que una arquitectura sin EJB no lo necesitará (bastará solo con un contenedor WEB).

2.1 Arquitecturas no distribuidas

2.1.1 Web con Business Component Interfaces (Interfases de Negocio)

Como se menciona en [RJ02], es una arquitectura simple que se puede usar en un gran número de aplicaciones. Utiliza un Servlet Container, como por ejemplo Tomcat, que provee la infraestructura necesaria.

La Web tier (capa Web) y la Middle tier (capa Media o de Negocio) corren en la misma JVM (Java Virtual Machine, Máquina Virtual de Java). La capa de negocio consistirá en una serie de interfases implementadas por clases simples de Java.

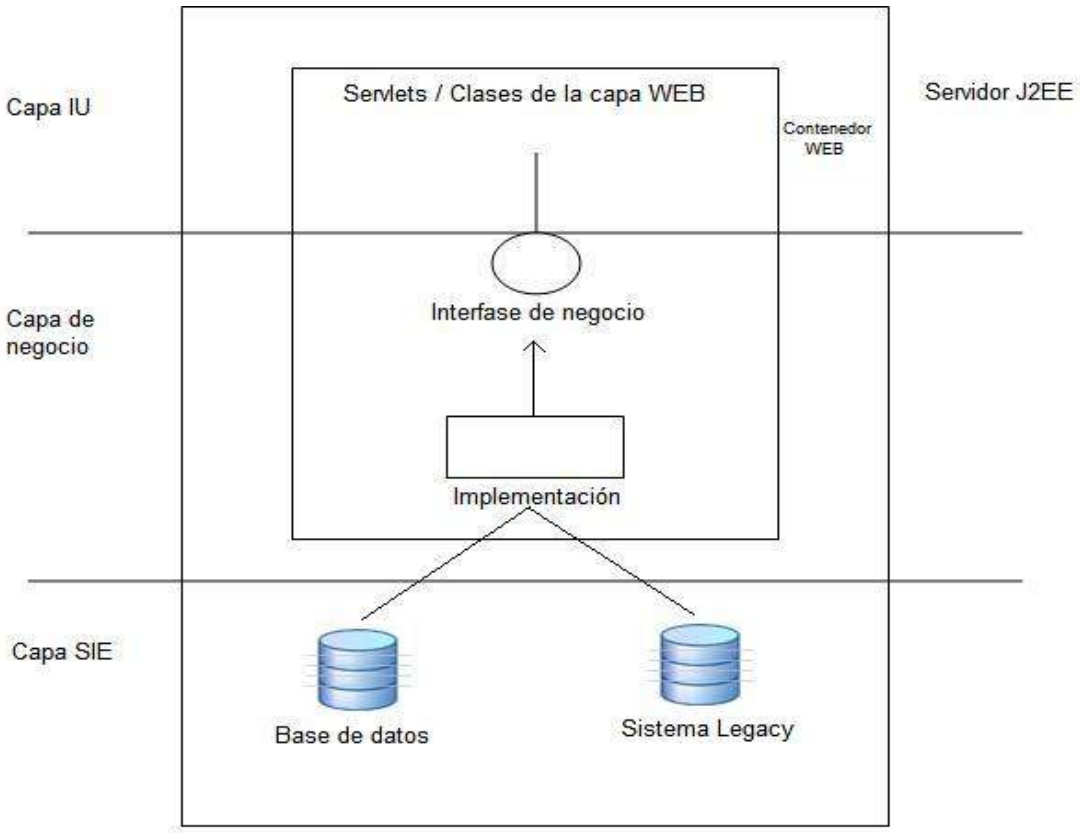

Figura 3. Arquitectura **Web con Business Component Interfases**. [RJ04]

2.1.2 Web con EJB locales

La diferencia de esta arquitectura con respecto a la anterior, está en la implementación de la Middle tier que está dividida en 2 partes [RJ04] y [RJ02]:

- Las Interfases de negocio que corren en el Servlet Container.
- Los EJB.

Ambas partes corren en la misma JVM:

- En la Web tier, se utilizará algún framework MVC (Model View Controller, Modelo Vista Controlador).
- En la Middle tier, se utilizará EJB de Sesión con interfases locales corriendo en un EJB container.

Los objetos de negocios podrán ser accedidos desde la Web tier en forma directa. El EJB Container es el que provee el manejo de transacciones.

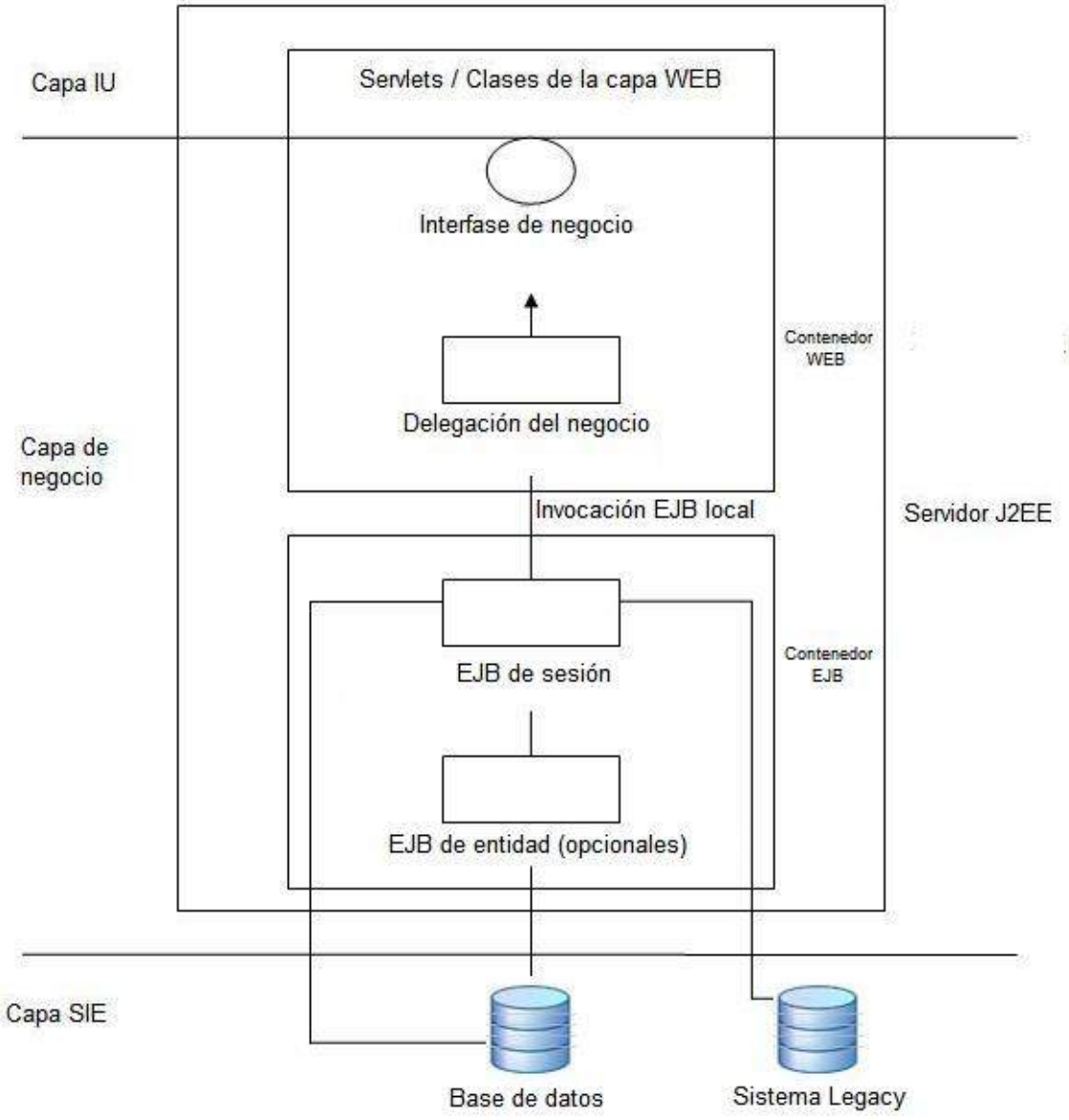

Figura 4. Arquitectura **Web con EJB locales**. [RJ04]

2.1.3 Web con Arquitectura de Contenedor Ligero

Como se menciona en [RJ04], las arquitecturas de contenedores ligeros son una alternativa a las tradicionales arquitecturas EJB. En este tipo de arquitecturas, tanto los servicios como los objetos de dominio son objetos POJOs (Plain Old Java Object, Objetos planos de Java) y éstos, en lugar de correr sobre un contenedor EJB, lo hacen sobre un Lightweight Container o contenedor ligero.

A continuación listaremos una serie de características relevantes que luego serán ampliadas en el siguiente capitulo:

- Un contenedor ligero no se "ata" a J2EE, es decir, puede correr en un contenedor Web, como una aplicación Standalone (independiente) o bien en un contenedor EJB si fuere necesario.

- Este tipo de arquitectura provee un mecanismo para localizar los objetos de negocio. Ya no se necesitarán más los "Services Locators" (Localizadores de Servicios) o "JNDI lookups" (búsquedas por medio de Java Naming and Directory Interfase, Interfase de Nombres y Directorios Java). El contenedor ligero proveerá la registración de los objetos [RJ04].

- Son arquitecturas no invasivas o no intrusivas. Además los objetos corren en la misma JVM y permiten ser combinados con algún framework AOP [MR05].

Algunos de los contenedores ligeros más conocidos son:

- Spring Framework (se explicará con más detalle en el capitulo 3)
  - http://www.springframework.org
- PicoContainer (se explicará con más detalle en el capitulo 4)
  - http://www.picocontainer.org
- HiveMind
  - http://hivemind.apache.org

Como se menciona en [RJ04], una arquitectura sin EJB está organizada de la siguiente manera:

- La capa de presentación será implementada usando algún framework MVC.
  No se necesitarán proxies (intermediarios) para el acceso a los POJOs, sino que los objetos de la capa de presentación trabajarán directamente con las interfases de los objetos de negocios.

- Los objetos de negocio serán POJOs corriendo en un Servlet Container o no.

- Para la capa de persistencia se utilizará algún framework para el O/R Mapping (Mapeo Objeto / Relacional) o bien directamente vía JDBC (Java Database Connectivity, Conectividad Java a Base de datos). Reemplazando a los EJB de entidad de la arquitectura J2EE tradicional (con EJB).

Ventajas [RJ04]:

- Es una arquitectura más simple que las anteriores.

- Esta arquitectura no requiere de un EJB container.

- Altamente portable entre servidores de aplicaciones.

- Los objetos de negocio pueden ser fácilmente testeados fuera del servidor de aplicación.

- El Inversion of control (control de inversión) permite al Lightweight Container “hacer wire-up” (enchufado), asociar los objetos vía xml, es decir, todo el lookup de los objetos es eliminado del código de la aplicación dejando nuestro código más limpio. En la siguiente sección, se explicará este concepto.

Desventajas [RJ04]:

- Como la arquitectura EJB local, esta arquitectura no tiene soporte para clientes remotos.

2.1.3.1 Separación en capas

Como se menciona en el libro de Rod Johnson [RJ05], este tipo de arquitectura se basa en buenas prácticas de POO (Programación Orientada a Objetos).

- **Presentation tier (capa de Presentación)**: Esta capa se apoya sobre una capa de servicios bien definida Esto significa que la capa de presentación deberá ser *fina* y no deberá contener lógica de negocio, sino todo el código que sea específico de la presentación como el código para manejar las interacciones con la web.
  La capa de presentación `puede usar algún framework de presentación como por ejemplo Struts, WebWork, JSF o Spring MVC.

- **Business Services Layer (Capa de servicios de negocio)**: Esta capa provee las siguientes funcionalidades:

  - Lógica de negocio que es específica de los casos de uso: mientras que los objetos de dominio contienen lógica de negocio que es aplicable a muchos casos de uso, esta capa implementa lo que es específico de un caso de uso.

  - Puntos de entrada claramente definidos para las operaciones de negocio. La capa de servicios provee las interfases usadas por la capa de presentación. Esta capa debe exponer Interfases Java, no clases (implementaciones).

  - Administración de transacciones.

- Verificación de restricciones de seguridad.

Esta capa es análoga a la capa de los EJB de sesión de las arquitecturas J2EE tradicionales.

Rod Johnson [RJ05] dice que "*dependiendo de la arquitectura elegida, mucha de la lógica de negocio puede residir en los objetos de dominios persistentes".* Esta aclaración hace ver que la importancia que le da Eric Evans [EE03] a la capa de dominio, no es la misma que la que le da Rod Johnson [RJ05].

- **Persistent Domain Objects (Objetos de dominios persistentes).** Es el core (núcleo) del modelo del dominio. Tiene que estar formada por verdaderos objetos (objetos que tengan comportamiento y encapsulen su estado). Se corresponde con la capa de dominio explicada por Eric Evans en la sección 1.7.3 Capas lógicas**.**

- **Data Access Objects (Objetos de acceso a datos)***:* O también llamados DAO's, encapsulan el acceso a los objetos persistentes del dominio, persisten los objetos transientes y actualizan los objetos existentes.

- **Data bases and legacy systems (Base de datos y sistemas Legacy)**. El caso más común y más simple es una única base de datos relacional. Sin embargo, puede ser más complejo con múltiples BD. Normalmente, esto se conoce como EIS (Enterprise Information System, Sistemas de Información Empresarial).

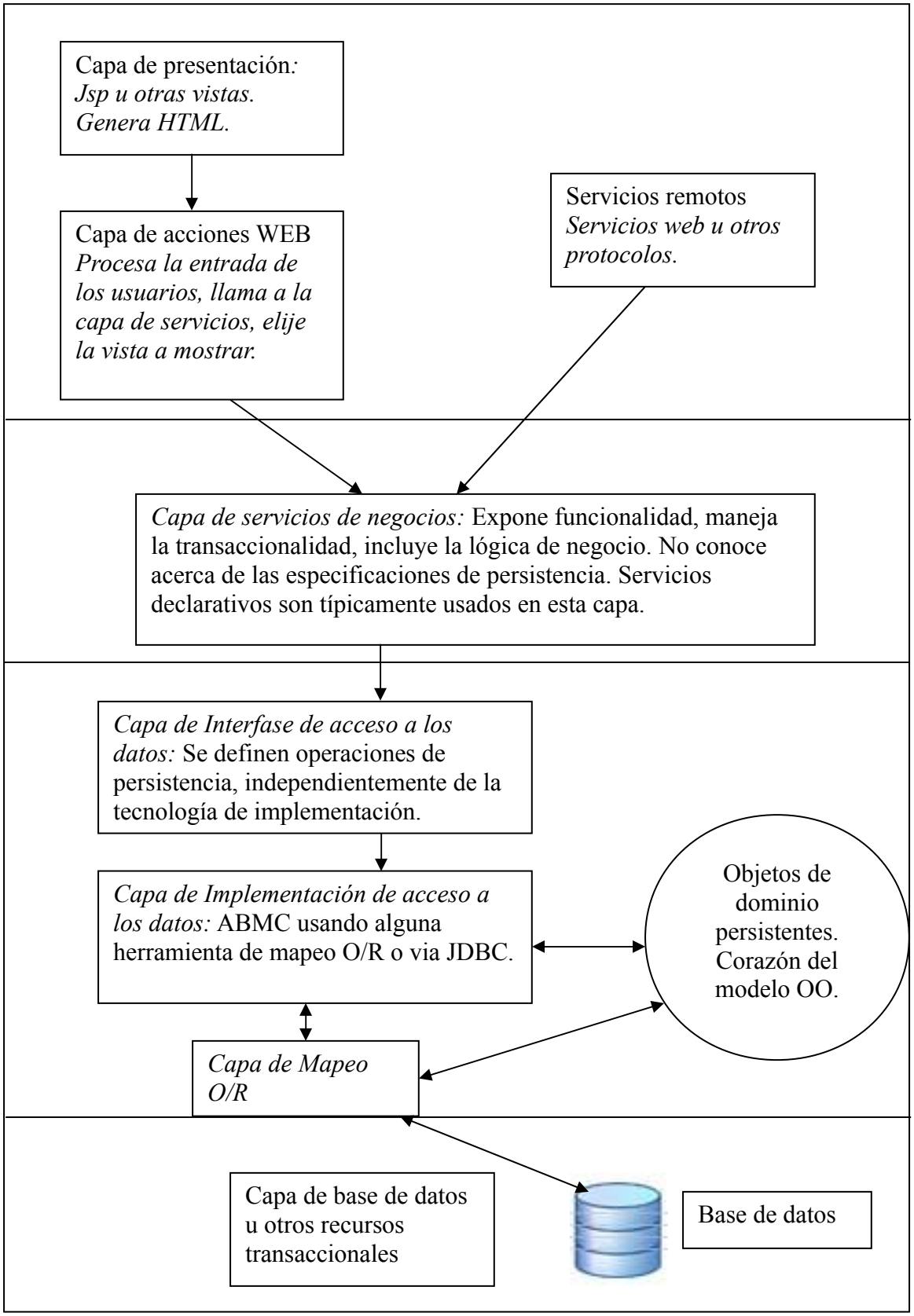

Figura 5. Arquitectura **Web con Lightweight Container**. [RJ05]

2.1.3.2 Inyección de Dependencias (ID) y el Control de Inversión (IoC)

Como se define en wikipedia [INT13], la ID es un patrón de diseño y modelo de arquitectura a veces referenciado como inversion of control - IoC - (inversión del control). Es una forma particular que tiene el container (contenedor) de realizar ò implementar la IoC.

Características [RJ05] y [RJ04]:

- Los objetos de una aplicación declaran cuales son sus colaboradores, es decir, cuales son las dependencias de los mismos.

- El container buscará cuales son las dependencias del objeto, y luego se invertirá el control, y el container inyectará dichas dependencias.

- El container es el responsable de la instanciación de los objetos.

- La ID es el proceso de inyectar a un objeto las dependencias que necesita.

- Las clases quedan autodocumentadas y hacen explícitas sus dependencias

- Permite centrarse en la lógica de negocio.

Para terminar de comprender bien estos conceptos se mostrará un ejemplo que fue extraído y modificado del sitio de Martin Fowler [INT12].

El ejemplo trata de una pequeñísima porción de código que permite verificar la existencia de una cuenta de mail de un usuario particular.

A continuación se irán definiendo las clases e interfases involucradas:

- La clase `WebUserDAOImpl` la cual actuará como finder (buscador) y sabrá como obtener los usuarios registrados de una base de datos.

- La clase `WebUserManager` que interactuará con el DAO anterior.

```
public class WebUserManager {
  .....
  .....
  .....
     private boolean mailExist(String email) throws
                                          BesyDAOException
     {
           return (null != webUserDaoImpl.findByMail(email) ?
                 true : false);
     }
 }
```

El problema aquí presente es la dependencia entre el objeto `WebUserManager` y el `WebUserDaoImpl`. ¿Y porqué decimos que es un problema?

Será un problema, ya que nuestro método `WebUserDaoImpl.findByMail(email)` queremos que sea completamente independiente de la forma en que son almacenados los usuarios. Lo ideal sería que ese método pueda, tanto acceder a una BD para verificar la existencia del mail, como así también, acceder a un archivo separado por comas, o bien a un xml, o conectarse a un web service, etc.

Es decir, si en un futuro queremos cambiar la implementación del método `webUserDaoImpl`, esto puede ser posible sin tocar el código de la clase `WebUserManager`.

Una buena solución a ésto es usar una interfase, de la siguiente manera:

- La Interfase `WebUserDAO` que actuará como finder (buscador) sin tener la implementación de los métodos.

```
public class WebUserDAO{

   public WebUser findByMail(String mail) throws
   BesyDAOException;
}
```

Y luego, nuestro `WebUserManager` quedaría de la siguiente manera:

```
public class WebUserManager {
     ....
     ....
    private WebUserDao webUserDao;


    public WebUserManager(){
              webUserDao = new WebUserDaoImpl();
    }

    private boolean mailExist(String email) throws
                                    BesyDAOException {
           return (null != webUserDao.findByMail(email) ?
                    true : false);
    }
}
```

Ahora bien, ¿qué sucedería si se desea cambiar la forma en que se almacenan los usuarios?. Necesitaríamos una clase diferente para acceder a los datos. Si bien, como se ha utilizado una interfase `WebUserDao` en el método `mailExist` de la clase `WebUserManager`, esto no alteraría el código en ese método, pero si estaría mal la implementación del finder (buscador).

Si vemos cuidadosamente, podríamos observar que la clase `WebUserManager` depende de la interfase `WebUserDao` como también de la implementación `WebUserDaoImpl`, por lo que se prefiere que solo dependa de la interfase.

Es aquí donde entra en juego la ID y la IoC, y es el Lightweight Container quien se encargará de inyectar las implementaciones de las dependencias (permitiendo a la clase `WebUserManager` depender solo de la interfase y luego en otro lado, indicar que dicha interfase tendrá tal o cual implementación). Este ejemplo continuará en la siguiente sección, y se utilizará al framework Spring como base para concluir esta explicación.

2.1.3.3 Formas de inyección de dependencias (ID)

Según Matt Raible, en su libro Spring Live [MR05], menciona 3 formas de ID:

- Setter Injection ó inyección por asignadores.

- Constructor injection o inyección por constructor.

- Method injection o inyección por método.

Según Rod Johnson [RJ04], además de mencionar solo las dos primeras formas de ID que Matt Raible menciona, hace una clasificación entre dos formas diferentes de implementación del IoC:

- Dependency Lookup (Búsqueda de Dependencia).

- Dependency Injection (Inyección de Dependencia).

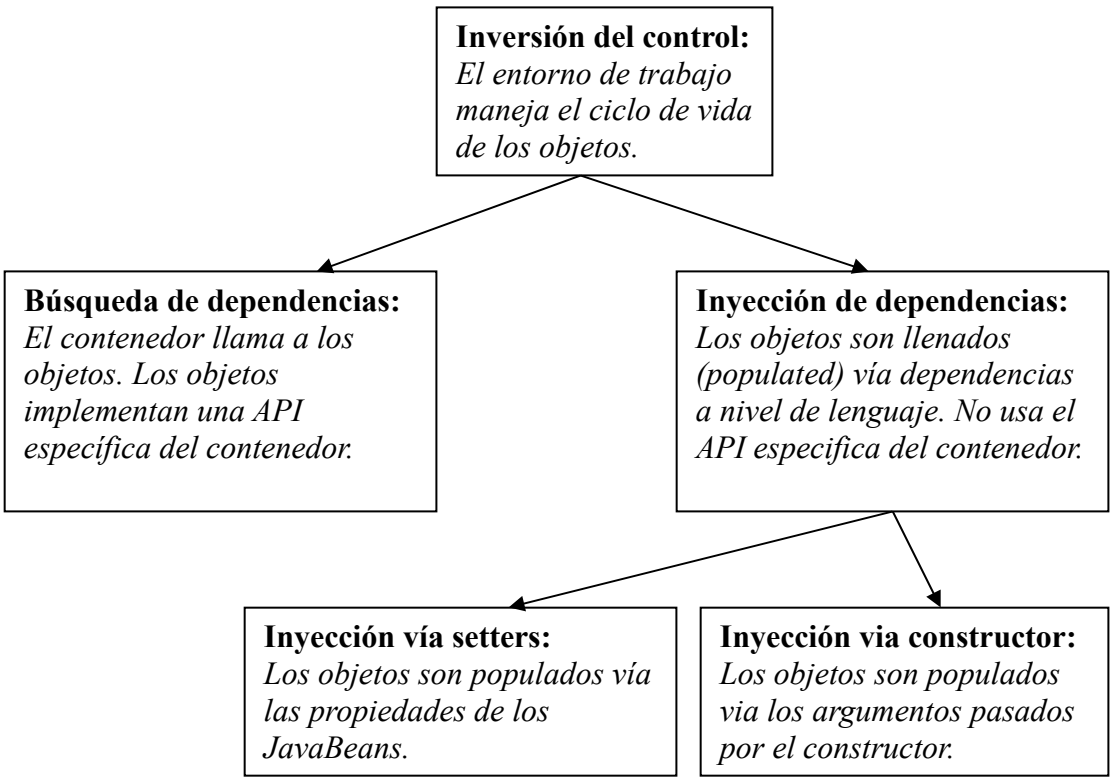

Figura 6. Formas de Inversión del Control. [RJ04]

Generalmente la inyección más utilizada por los desarrolladores es la setter

injection (inyección por los métodos asignadores) [RJ05]. A su vez, es la preferida por Rod Johnson [RJ04].

Continuaremos con el ejemplo de la sección anterior (tomando al framework Spring como contenedor ligero) para concluir la explicación de la ID y IoC.

En un xml se definirán las dependencias y sus implementaciones, quedando estas autodocumentadas y liberando al código de tener la implementación, haciéndolo más flexible:

```
<bean id="webUserManager"
class="besy.escuela.manager.WebUserManager">
        <property name="webUserDao"><ref bean="webUserDao"/>
          </property>
         ....
         ....
</bean>
<bean id="webUserDao"
class="besy.escuela.dao.hibernate.WebUserDaoImpl">
         ....
</bean>
```

Luego el código de la clase `WebUserManager` nos quedará de la siguiente manera:

```
public class WebUserManager {
     private WebUserDao webUserDao;

     public void setWebUserDAO (WebUserDAO webUserDao) {
          this.webUserDao = webUserDao;
     }

      private boolean mailExist(String email) throws
                              BesyDAOException {
            return (null != webUserDao.findByMail(email) ? true :
                   false);
       }
}
```

Como se puede apreciar, el código queda desacoplado de la implementación del buscador (finder), solo teniendo como dependencia la interfase `WebUserDAO`.

2.2 Arquitecturas distribuidas

2.2.1 Web con EJB remotos

Como se menciona en [RJ02] y [RJ04], esta arquitectura es la que se conoce como la Arquitectura J2EE clásica. Ofrece la posibilidad que la Middle tier corra en un servidor distinto al de la Web tier.

La capa Web o Web tier se comunica con la capa de negocio o Middle tier usando RMI (Remote Method Invocation, Invocación a Método Remoto).

Las implementaciones de las interfases de negocio tienen que manejar el acceso remoto a los EJB. La Middle tier está dividida en 2 partes (al igual que la arquitectura Web con EJB locales):

- En la capa web se utilizará algún framework MVC.

- En la capa de negocio se utilizará EJB de sesión con interfases remotas corriendo en un EJB container.

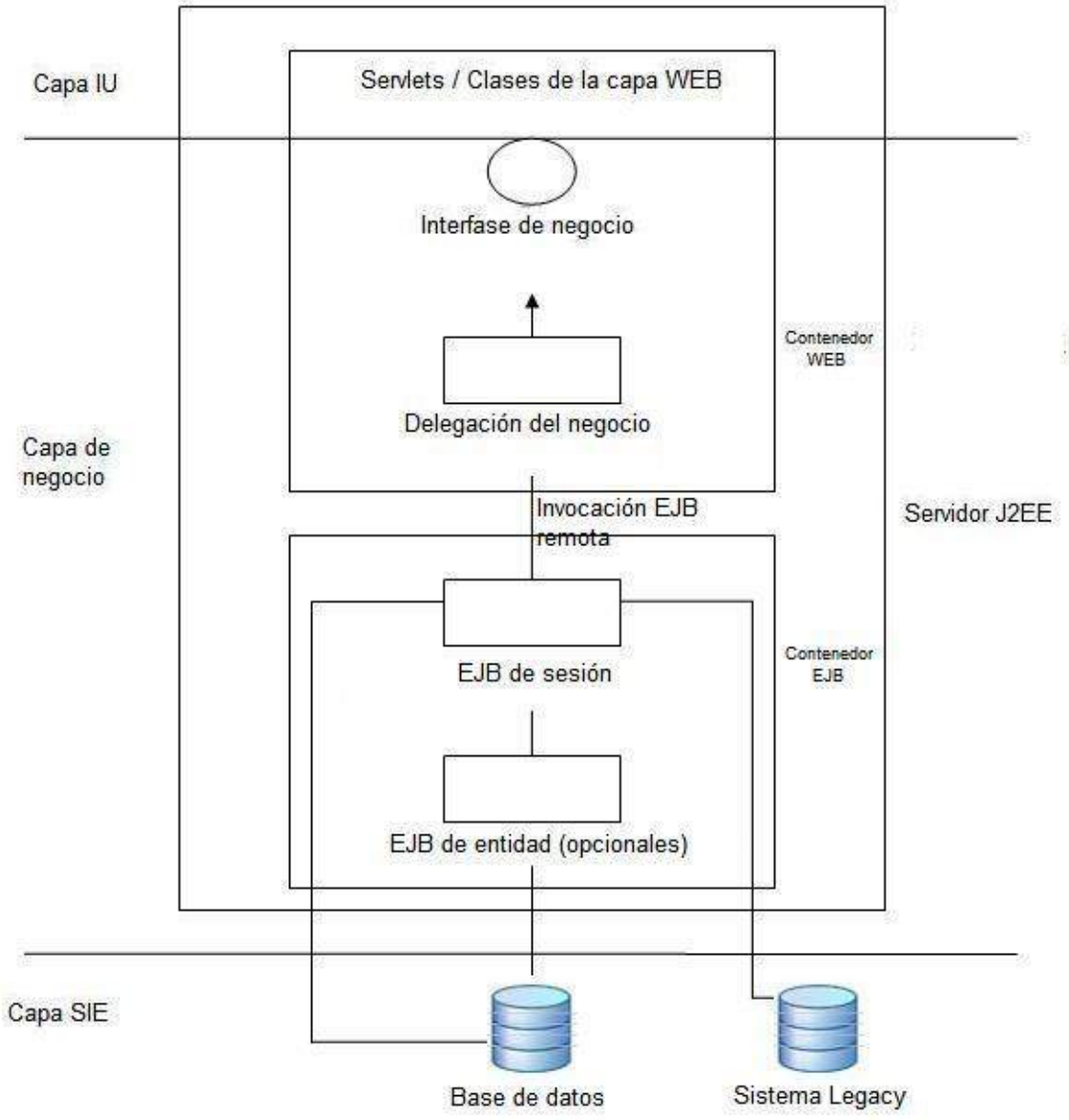

Figura 7. Arquitectura **Web con EJB remotos**. [RJ04]

2.2.2 Web con Web Services (servicios web)

Según Ramesh Nagappan et al. [RRR03], un web service está basado en el concepto de SOA (Sofware Oriented Architecture, Arquitectura Orientada por Servicios). Los web services exponen la lógica de negocio como servicios sobre Internet / intranet, por medio de interfases programables y utilizan los protocolos de Internet para proveer los mecanismos de búsquedas, suscripción e invocación de dichos servicios.

Por otra parte la W3C [INT15] (World Wide Web Consortium, Consorcio de la WWW) define a los web services como "un sistema de software diseñado para soportar interoperabilidad entre máquinas".

Usan un sistema estándar de mensajería basada en XML.

- XML-RPC (Remote Procedure Call, Llamadas a Procedimientos Remotos).
- SOAP (Simple Object Access Protocol, Protocolo Simple de Acceso a Objetos).
- Como alternativa HTTP GET/POST pasando XML.

Como se define en el sitio MSDN de Microsoft [INT14], “SOAP es un protocolo elaborado para facilitar la llamada remota de funciones a través de Internet”'.

Esta arquitectura agrega una capa de web services sobre las interfases de negocio. Si bien, J2EE no presenta un estándar para soportar los web services, si se puede integrar productos de terceros, los cuales proporcionan fácilmente el soporte SOAP a los servidores J2EE. Uno de los productos de terceros más usados es AXIS (framework de Apache que implementa SOAP).

Una de las principales ventajas que tiene SOAP sobre RMI/IOP - Remote Method Invocation / Insurance Object Protocol - (utilizados por las aplicaciones J2EE tradicionales) es que es más abierto. Pueden soportar tanto aplicaciones J2EE como también aplicaciones .NET o PHP [RJ02].

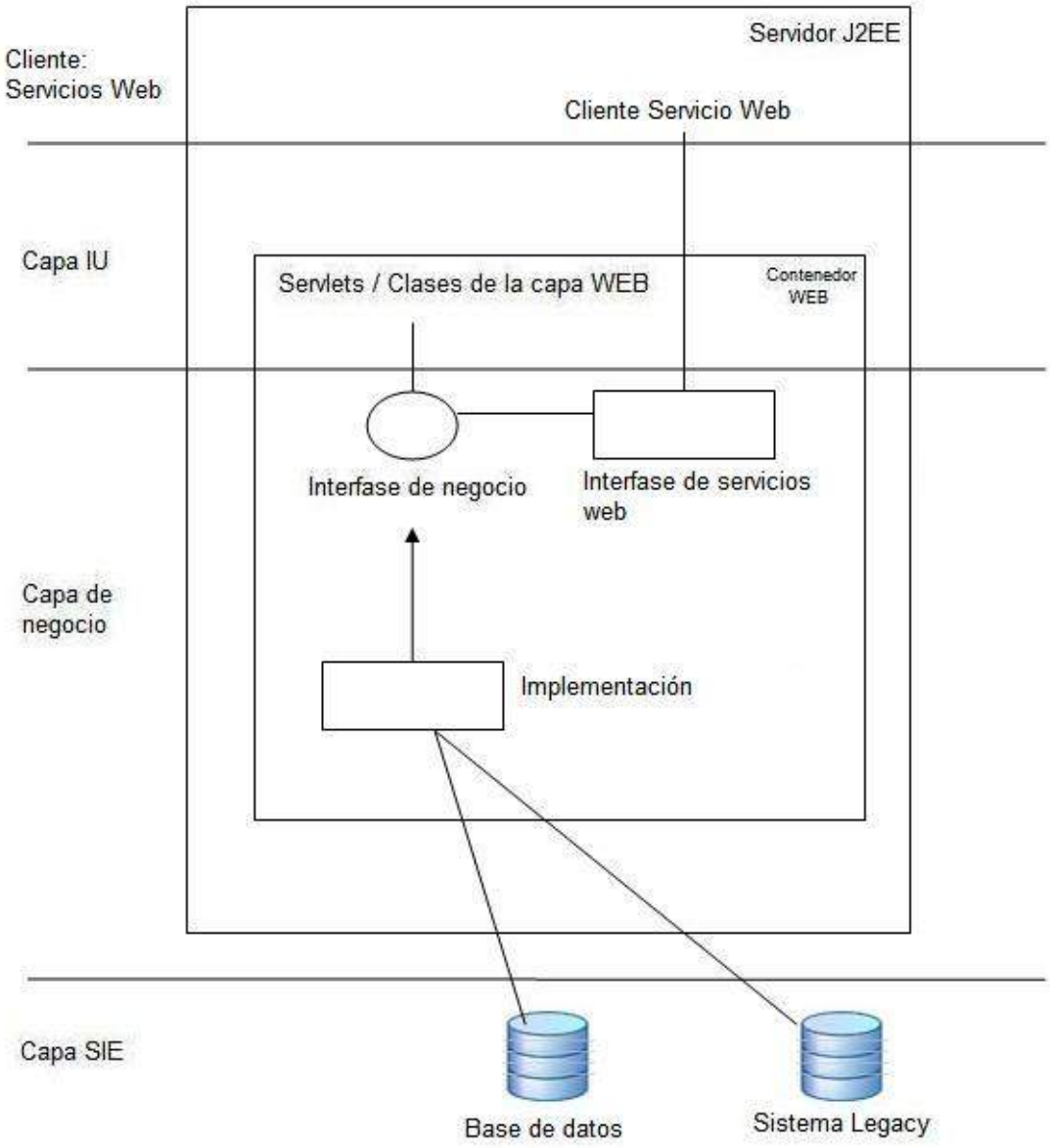

Figura 8. Arquitectura **Web con Web Services** . [RJ02]

2.3 Comparación de Debilidades y Fortalezas

La clasificación de los tipos de arquitecturas y la comparación de debilidades y fortalezas que se presentan a continuación están tomadas de [RJ02] y [RJ04].

Tipo de Arquitectura	Fortalezas	Debilidades
Web con Business Interfases (interfases de negocio).	- *Simplicidad* - *Velocidad*: - *No interfiere sobre OO.* El Diseño OO no se dificulta por las implicaciones de llamar a EJB's. - *Fácil de testear.* Con un diseño apropiado se pueden correr los test unitarios sobre las interfases de negocio sin necesidad de la capa Web.	- *Sólo soporta clientes web*. No puede soportar un cliente GUI. Sin embargo, se le puede agregar una capa de web services. - *Corre en una sola JVM*. Si bien esto aumenta la performance, no podemos distribuir los componentes en distintos servidores físicos. - *No tiene soporte de transacciones por medio del container EJB.* deberán ser manejadas por código. - ¿ Cómo se acceden a los objetos de negocio ?. ¿ Cómo se representa la configuración ?. Como podemos ver, quedan muchos interrogantes por resolver.
Web con EJB Locales.	- *Menos compleja que una arquitectura EJB distribuida que usa interfases remotas.* - *EJB no es tan intrusivo*. Se implementan como EJB's aquellos objetos que requieren servicios del Container EJB. - *Manejo de transacciones resuelto por EJB.*	- *Mas compleja que una arquitectura Web con Business Interfases (interfases de negocio).* - *Corre en una sola JVM.* - *Los EJB son difíciles de testear*. Es necesario correr los test unitarios desde el container EJB. - *Se introduce el overhead de los EJB.*
Web con Contenedores Ligeros.	- *Es una arquitectura simple pero poderosa.* - *La Escalabilidad Horizontal* (ver sección 1.7.6).Se puede lograr	- *No soporta clientes remotos.* Salvo que se agregue un componente especial. - *No hay un estándar para Contenedores Ligeros como si existe para*

	haciendo clustering de contenedores web. - *A los POJOs se les suministra servicios declarativos* (ej. administración de transacciones) a través de AOP (Aspect Oriented Programming, Programación Orientada a Aspecto). - *No requiere de un EJB container.* - *El uso de IoC y ID hace que el container le suministre a cualquier objeto, los colaboradores que necesite para trabajar.* Esto no sucede en el resto de las arquitecturas. - *Simplifica el test unitario*. Ya que no se necesita del Servlet Container para correr los test. - *Tiene todas las ventajas de la arquitectura Web con Buisines Interfase*. Elimina sus desventajas.	*Contenedores EJB.* - *Puede ser una arquitectura menos familiar que las arquitecturas EJB.*
Distribuida con EJB Remotos.	- *Soporta cualquier tipo de cliente* (web, GUI, etc.). - *Permite la distribución de los componentes en distintos servidores físicos.* - *Manejo de transacciones resuelto por EJB*.	- *Es una de las más complejas*. Esto puede aportar más costos que beneficios. - *Por ser distribuida es más difícil de testear y para la realización de debugging (eliminación de errores).* - *Sacrifica la OO por la distribución..* - *Manejo de excepciones más complejo.*
Distribuída con Web Services.	- *SOAP es más abierto que RMI/IIOP*. - *Soporta clientes multiplataformas*. Permite la interoperabilidad entre aplicaciones. - *EL protocolo SOAP puede ir sobre HTTP* (se evitan problemas de Firewalls).	- *Performance* .El overhead de parsear objetos sobre el protocolo basado en XML como SOAP. - *Serializar y des-serializar objetos complejos requiere código customizado..* - *J2EE no presenta un estándar para soportar los web services en*

		comparación con EJB. - *Falta de unidad en la parte de servicios de seguridad.*

Capítulo 3 – Spring –

Spring proporciona una solución ligera para el desarrollo de aplicaciones empresariales, apoyando la posibilidad de usar transacciones declarativas, acceso remoto a la lógica de negocio vía RMI o Web Servicies y varias opciones para persistir los datos en una base de datos. Entre otras funciones, Spring proporciona un marco completo en lo que respecta a MVC y caminos transparentes para integrar AOP en el desarrollo del software [SR07].

Antes de sumergirnos en el mundo de Spring y todas sus características, vamos a dejar en claro un concepto que se utilizará a lo largo de esta sección para evitar ambigüedades en el uso de algunos términos: Los terminos “bean” y “JavaBean”, Spring los utiliza liberalmente para referenciar a componentes de la aplicación lo cual no significa que dichos componentes tengan que cumplir con las especificaciones de JavaBeans a la perfección. Estos componentes serán cualquier tipo de POJO [WB05].

3.1 Introducción

Según Rod Jonson [RJ05], Spring es un entorno de trabajo de código abierto que apunta a realizar el desarrollo de aplicaciones J2EE de manera más fácil, rápida y productiva.

Como así lo menciona Seth Ladd et al. [SL07], Spring le ha dado una nueva vida a los desarrollos en Java. Las promesas iniciales de J2EE fueron enterradas, mientras que .NET se convertía en una fuerte amenaza. Las empresas comenzaron a querer más aplicaciones con menos dinero y esfuerzo y esto no podía ser resuelto con J2EE. Luego del libro de Rod Johnson [RJ02] y su evolución eventual en el entorno de trabajo llamado Spring, el mundo de Java tenía una nueva luz de esperanza.

3.1 Un poco de historia

Como se menciona en el libro de Matt Raible [MR05], Rod Johnson es el inventor de Spring. En el año 2002, Rod Johnson escribe el libro llamado “Expert One-on-One J2EE Design and Development” [RJ02] en el que explica sus experiencias con J2EE y como a menudo los EJB (Enterprise Java Beans, Objetos Empresariales de Java) no son una buena elección para el desarrollo de los proyectos. Rod Johnson en su libro [RJ05], destaca nuevamente a los entornos de trabajo ligeros, mencionando a Spring. Spring creció a partir de la experiencia de Rod Johnson como consultor en varios proyectos J2EE entre los años 1997 y 2002. Luego, el libro antes mencionado y las 30 líneas de código de ejemplo, causaron un impacto positivo en los lectores [RJ05] [INT50].

Por otra parte para Craigs Wall et al. [WB05], todo comienza con el nacimiento de los beans. Allá por el año 1996 Sun Microsystems publica la especificación 1.0 de Java-beans donde estos definen un modelo de componente de software.

Luego en marzo de 1998, Sun publicó la versión 1.0 de la especificación de Java-beans empresariales (EJB) [CR06]. Esta especificación amplió la noción de los componentes de Java del lado del servidor, proporcionando servicios empresariales necesarios para cualquier desarrollo, pero no ha podido continuar con la simplicidad de la especificación original. De hecho, salvo la semejanza del nombre, el resto es completamente distinta de la especificación original de Javabeans [WB05].

Como bien los indica Cris Richardson [CR06], la versión 1.0 de esta especificación proveyó dos tipos de beans empresariales: Los beans de sesión (representan los servicios) y los beans de entidad (representan los datos de la base de datos y fueron originalmente pensados para ser implementados como objetos de negocio). Luego surgió la version 2.0 que redefinió el modelo de programación EJB. Agregando un nuevo tipo de beans llamados Message-driven Beans los cuales procesan mensajes JMS (Java

Message Service – Servicio de Mensajeria de Java). Por ultimo la evolución llego a los EJB 3.0 que simplifico considerablemente el modelo de los EJB.

A pesar de que se han construido muchas aplicaciones empresariales exitosas basadas en EJB, EJB nunca alcanzó realmente su propósito previsto: Simplificar el desarrollo de aplicaciones de software. Esto llevo a que muchos desarrolladores dejaran de lado a los EJB para buscar otras soluciones más simples [WB05].

Mucho antes de que la especificación EJB 3.0 saliera a la luz, algunos desarrolladores desilusionados con los EJB de las versiones iniciales comenzaron a buscar otros entornos de trabajo que cubrieran sus espectativas. Los POJOs parecían la mejor solución, pero por si solos no eran suficientes. Es decir, una aplicación empresarial requiere de servicios tales como manejo de transacciones, seguridad y persistencia que previamente eran provistos por los contenedores EJB. Es así que comienza a nacer una solución que se empezó a utilizar cada vez más: Los entornos de trabajo de contenedores ligeros (lightweight frameworks). Estos frameworks comienzan a reemplazar a los frameworks pesados o llamados (heavyweight frameworks) [CR06].

Es por todo esto que Craigs Wall et al. [WB05], asegura que los próximos desarrollos serán basados en componentes Java. Las técnicas de AOP y IoC están dando a los Javabeans mucho del poder que antes tenían los EJB. Es decir, estas técnicas equipan a los Javabeans con el modelo de programación declarativo pero sin la complejidad de los EJB.

En febrero de 2003, Rod Johnson se junta con Juergen Hoeller (siendo uno de los mayores colaboradores ni bien Rod Johnson saca a la luz el libro [RJ02]). Ambos juntaron a otros desarrolladores (Thomas Risberg, Colin Sampaleanu y Alef Arendsen entre otros) para el desarrollo de este framework o entorno de trabajo. Por otro lado, escribieron juntos el conocido libro llamado “Expert One-on-One J2EE Development without EJB” [RJ04] que describe como Spring soluciona muchos de los problemas que existían en J2EE (en las versiones anteriores a la nueva especificación 3.0 de EJB) [MR05] [RJ05].

Spring fue originalmente conocido como un contenedor de inyección de dependencias (ID). Termino que llevo a la luz Martin Fowler y que luego fue conocido

como Inversión del Control (IoC), Una visión alternativa es que la ID es una estrategia de implementación de la IoC. Por otra parte, La mayoría de las explicaciones de ID se refirieron al principio de Hollywood: “No nos llames, nosotros te llamaremos.” [BSA06].

Rob Harrop y Jan Machacek [HM05], explican que este comportamiento de inyectar dependencias en tiempo de ejecución por parte del contenedor, conduce a la IoC a ser retitulado con el nombre de ID. Usar el termino de ID cuando se refiere a IoC es correcto. En el contexto de Spring, se pueden utilizar los términos alternativamente, sin ninguna pérdida de significado.

Los fundamentos arquitectónicos de Spring fueron desarrollados por Rod Johnson desde principios del 2000 (antes de Struts y otros frameworks) [MR05].

Luego en Agosto de 2004, Rod, Juergen, Colin y otros desarrolladores del core de Spring se juntaron para fundar los que se conoce como interfase21, una compañía dedicada al soporte y consultas de Spring. Luego del release de marzo de 2004 (Spring version 1.0), Spring fue totalmente adoptado por la comunidad de desarrolladores [RJ05].

3.2 La especificación EJB 3.0

Por otra parte, como lo menciona Rod Johnson en su libro [RJ05], es importante destacar como Java y J2EE fueron evolucionando y como Spring encaja dentro de este gran ecosistema J2EE.

Los principales cambios en el desarrollo de Spring se refieren a la especificación EJB 3.0 (desarrollada por un grupo de expertos JSR-220). Básicamente EJB 3.0 introduce dos cambios importantes para los usuarios de Spring y para el posicionamiento de este frente a J2EE.

- Inyección de dependencias: Provee un simple acceso a los objetos del contexto. Sin embargo Rod Johnson destaca que este manejo de inyección de dependencias es bastante limitado en comparación con Spring u otros

contendores IoC. Así mismo, Chris Richardson [CR06], señala que solo se pueden inyectar objetos JNDI en los EJBs

- Introduce una nueva especificación para la persistencia de POJOs: Estrictamente hablando, JSR-220 proporciona dos especificaciones: Una es la especificación EJB 3.0 y la otra es, la especificación para la persistencia de POJOs, que soporta la mayoría de los productos O/R mapping incluyendo TopLink, Hibernate, JDO, etc.

Según el sitio de especificaciones de la tecnología Java, conocido como www.jcp.org [INT47], como así también los sitios de Oracle [INT46] y JBOSS [INT49], coinciden en que dos de los principales logros del la especificación 3.0 de EJB son:

- La reducción de la complejidad de la API mejorando la arquitectura EJB (enfocando más en el desarrollo de POJOs)

- La estandarización de la API de persistencia.

Por otra parte, según [CR06], la nueva especificación agregó una serie de ventajas que vale la pena destacar:

- Los EJBs pueden correr fuera del contenedor EJB.

- No se requiere de archivos XML de configuración para describir la configuración de los beans. EJB 3.0 usa las nuevas características de Java 5 conocidas como “annotations” (anotaciones). Dichas anotaciones son como datos extras que se escriben en la clase de Java y le agregan funcionalidad. Dicha funcionalidad extra es interpretada por el contenedor de EJBs.

- Los EJB son POJOs. Es un cambio rotundo si tenemos en cuenta que según la especificación 2.0 para poder implementar un Session Bean (uno de los tres tipos de EJB) se necesitaba escribir una clase que implementará la interfase SessionBean, otra interfase que extendiera de EJBHome y un componente que extendiera de EJBObject o EJBLocaleObject.

- Se mejoró el API de persistencia, siendo similar a las APIs de JDO o Hibernate.

- "Desatachamiento" de los objetos. En la especificación 2.0, las aplicaciones manejaban el concepto de DTO (Data Transfer Object, Objeto de Transferencia de Datos) para intercambiar datos entre la capa de negocio y de presentación. Luego la nueva especificación eliminó el uso de los DTOs para manejar un nuevo concepto. Cuando una transacción finaliza, todos los entity beans (beans de entidad - uno de los tres tipos de EJB) que fueron cargados en la aplicación automáticamente son desatachados del contenedor y pasados a la capa de presentacion. Luego la capa de presentación puede operar sobre el bean desatachado, por ejemplo, actualiza cierta información, y éste es pasado nuevamente a la capa de negocio para ser nuevamente atachado y actualizado en la base de datos.

Así como también se destacan las siguientes desventajas:

- Soporte limitado para el manejo de las colecciones. La nueva especificación sólo soporta colecciones de entidades. No es posible tener colecciones de Enteros (Integer) o Strings. Lo que es muy común en un modelo de dominio de POJOs. Por otra parte, no elimina automáticamente los hijos huerfanos, es decir, elementos que no tienen asociado ningún objeto padre.

- La inyeccion de dependencias solo puede inyectar objetos JNDI en los EJB.

- Los SessionBeans y MessageBeans necesitan ser desplegados en un contenedor EJB. Sin embargo los EntityBeans no lo necesitan.

- Complejidad del ambiente. Los EJB añaden complejidad extra al incorporar un servidor de aplicaciones.

Para finalizar, según [RJ05], Java esta evolucionando por si mismo. Con el nuevo release J2SE 5.0 en septiembre de 2004, Java ha tenido uno de los mayores cambios de su historia.

3.4 Problemas de las arquitecturas J2EE tradicionales

Desde la implementación de J2EE en 1999/2000, J2EE no ha sido calificado como exitoso en la práctica. Uno de sus causantes es el excesivo esfuerzo que se requería para desarrollar y su baja performance, debido mayormente a su exceso de capas y objetos inútiles [RJ05].

Por otro lado, Rod Jonson [RJ05] señala que el framework EJB es incomodo y restrictivo, es decir, el trabajo que lleva implementar los EJB es excesivo y da la apariencia que toda la lógica de negocio es implementada mediante EJB.

A continuación se listaran los problemas que tenían las arquitecturas J2EE tradicionales y que ambos autores, Rod Jonson y Matt Raible, consideran en común. [RJ05] y [MR05]:

- Las aplicaciones J2EE tienden a contener código “plumbing”: JNDI lookups, bloques de try catch, Transfer Objects, etc.

- El modelo EJB es altamente complejo.

- Las aplicaciones J2EE son difíciles para la realización de los test unitarios. Como la lógica de negocio está en los EJB, entonces se necesita inicializarlos y su startup es bastante largo, lo que los hace ineficientes. El uso de los TDD es el mejor camino para producir código de alta calidad.

- Muchos patrones J2EE de diseño no son patrones de diseño sino más bien workarounds para las limitaciones de la tecnología**.**

Dando otro enfoque de este tema, según [CR06] si bien la especificación EJB 3.0 hizo un giro importante respecto las versiones anteriores, sigue teniendo ciertas limitaciones:

Fuerza a los componentes a una de las 3 categorías (beans de sesion, de entidad o manejados por mensajes). Donde todo objeto que no cumpla con alguna de estas 3 categorías no podrá utilizar los servicios proporcionados por los contenedores EJB. Por otra parte, no existen garantías, según Cris Richardson [CR06], de que los contenedores EJB prevean un rápido y confiable despliegue de los EJBs y como resultado, los EJB 3.0 dan la sensación de ser inferiores a las tecnologías de contenedores ligeros tales como Spring, PicoContainer, JDO, Hibernate, etc.

Sin embargo, Cris Richardson en su libro – POJOs in Action - [CR06], menciona que a pesar de sus limitaciones, es extremadamente probable que EJB 3.0 sea ampliamente utilizado por una razón simple, es parte del estándar de J2EE.

Es también importante recordar que los EJB son una tecnología apropiada para:

- Aplicaciones que usan transacciones distribuidas iniciadas por clientes remotos.

- Aplicaciones que requieren de un manejo de mensajeria como ser message-driven beans.

Y concluye explicando que existen otras alternativas superiores y ellas son entre otras las aplicaciones de contenedores ligeros.

3.5 EJB 3.0 VS Spring

Según Michael Juntao Yuan [INT48], el autor del libro "JBoss Seam: Simplicity and Power Beyond Java EE 5.0" y el autor del sitio http://www.michaelyuan.com, si bien existen numerosos libros y artículos que comparan Spring o EJB 3.0 contra EJB 2.1, no existe un estudio serio en el que se compare la tecnología EJB 3.0 y Spring. A continuación listaremos un conjunto de pros y contras extraídos de los siguientes autores Rod Johnson [RJ04], Craigs Wall et al. [WB05], Chris Richardson [CR06] y el sitio de Internet [INT48]:

Independencia del proveedor. Una de las razones para que los desarrolladores elijan la plataforma de Java es su independencia con los proveedores de APIs.

EJB 3.0 fue desarrollado y soportado para los principales desarrollos de código abierto y comerciales. Es decir, los desarrolladores no necesitan aprender Hibernate ni TopLink (dos de los principales O/R mappings del mercado actual) ya que la especificación se abstrae de la implementación. Pero por otra parte, no todos los servidores soportan EJB 3.0 de forma nativa. A su vez, según Craigs Wall et al. [WB05] y Rod Johnson [RJ04], los EJB son invasivos, es decir, para utilizar los servicios proporcionados por los contenedores EJB, se deben utilizar las interfases de Javax.ejb.*, atando el código fuente fuertemente a la tecnología EJB, haciendo casi imposible utilizarlo fuera del contenedor EJB**.**

En la otra vereda, encontramos a Spring, que si bien todavía no es un estándar como si lo es EJB 3.0, y es aquí donde queremos resaltar el concepto: Según [WB05] y haciendo hincapié en esto, EJB es una especificación definida por la JCP lo cual significa:

- Soporte de toda la industria
- Gran adopción a nivel empresarial
- Es un blanco perfecto para los proveedores que desarrollan herramientas que ayudan a los desarrolladores a desarrollar herramientas a su vez.

Spring permite correr las aplicaciones desarrolladas con el, en cualquier servidor tenga o no soporte para EJB. Las limitaciones en cuanto a la integración de servicios

estarán impuestas por el propio Spring. Según Michael Juntao Yuan, Spring proporciona una serie de clases llamadas “helpers (ayudantes)”. Por ejemplo, para los servicios de persistencia, Spring viene con diversos DAOs y templates de ayuda para JDBC, Hibernate, TopLink, etc.

Integración de servicios. El estandar EJB 3.0, se integra firmemente en los servidores de aplicaciones, lo que permite a los proveedores de APIs, para esta tecnología, optimizar fuertemente su performance. Mientras que en Spring es más difícil optimizar esta integración. Por ejemplo, para utilizar el servicio declarativo de manejo de transacciones de hibernate, se debe configurar explícitamente los objetos Spring TransactionManager y el Hibernate SessionFactory en un archivo xml de configuración.

XML vs. Anotaciones (Annotations). Las interfases de programación de Spring se basan en archivos XML de configuración mientras que EJB 3.0 hace un uso más extensivo de las anotaciones. Los archivos de XML pueden expresar relaciones complejas, son también muy prolijos y menos robustos. Las anotaciones son simples y sucintas, pero es dificil expresar las estructuras complejas o jerárquicas por medio de este mecanismo.

Servicios declarativos. EJB 3.0 configura servicios declarativos usando las anotaciones de Java, mientras que Spring utiliza los archivos XML de configuración. En la mayoría de los casos, el uso de las anotaciones de EJB 3.0 es la manera más simple y elegante para este tipo de servicios.

Inyección de dependencias (ID). Tanto Spring como EJB 3.0 proporcionan un soporte extensivo para este patrón, pero ambos tienen profundas diferencias. Por un lado Spring soporta la ID mediante el uso de archivos de configuración XML. Por otro lado EJB 3.0 apoya a las anotaciones para este fin. Básicamente la especificación EJB 3.0 define los recursos del servidor que pueden ser inyectados por las anotaciones pero no soporta la inyeccion de POJOs definidos por el usuario en otros POJOs. Es decir, con Spring se puede inyectar cualquier POJO en cualquier otro POJO.

Como conclusión, Michael Juntao Yuan expresa que ambos apuntan a proporcionar servicios empresariales débilmente acoplados usando diferentes caminos

para llegar a esta meta. La inyección de dependencias es un patrón muy usado en ambos entornos de trabajo. Por otra parte, Brian Sam-Bodden [BSA06] y Matt Raible [MR05] en sus libros, hacen mención a que Spring pone sobre la mesa la codificación de interfases (exponiendo contratos públicos – interfases – y pudiendo tener varias implementaciones de dichas interfases), una técnica muy conocida en la POO, lo que hace al código mucho más limpio y fácil de testear.

En EJB 3.0, el acercamiento a un estándar, el uso amplio de anotaciones y la integración con los servidores de aplicaciones resulta en una mayor independencia de los proveedores mientras que con Spring, el uso consistente de la inyección de dependencias y la centralización de la configuración en un archivo XML permite a los desarrolladores construir aplicaciones más flexibles y trabajar con varios proveedores de servicios a la vez.

Por otra parte, Craigs Wall et al. [WB05] disiente en varios aspectos cuando intenta comparar las dos tecnologías, arribando a la siguiente conclusión: “Una vez más, las características de los EJB pueden no ser las mejores respecto al desarrollo que uno desee llevar a cabo”. Básicamente menciona que Spring proporciona casi todos los servicios proporcionados por un contenedor EJB con la diferencia que permite escribir código mucho más simple. Por otra parte, Rod Johnson [RJ04], coincide en varios aspectos con Craigs Wall agregando que Spring proporciona mayor productividad, una mayor orientación a objetos y mayor facilidad en el testing de las aplicaciones, al no tener que depender de los servidores de aplicaciones. Así mismo [SL07], destaca que uno de los mayores fuertes de Spring es que permite desarrollar aplicaciones aplicando fuertemente los conceptos de OO permitiendo al desarrollador focalizar su esfuerzo en la lógica de negocio del modelo de objetos.

Pero por otra parte, Rod Johnson asegura que la verdadera fuerza de EJB es la fuerte integración que tienen los servicios empresariales con el contenedor, lo que significa que, los EJB brindan una única solución para la administración de los objetos de negocio. Otra de las ventajas que Rod Johnson le encuentra a los EJB es que es una especificación estándar y que la especificación 3.0 de los EJB introdujo cambios importantes.

3.6 Contenedores Ligeros

Rod Johnson en su libro [RJ04], utiliza el término container como framework o entorno de trabajo en donde el código de la aplicación corre o se ejecuta. Es decir, muchos objetos y en especial los objetos de negocio (contienen lógica propiamente del domino de la aplicación) que corren en el contenedor son manejados o administrados por el dicho contenedor. Existen numerosas arquitecturas y modelos de contenedores y tradicionalmente la más popular en J2EE es EJB para el manejo de los objetos de negocio.

Según Rod Johnson [RJ04], todo contenedor debe proporcionar los siguientes servicios:

- El contenedor deberá administrar el ciclo de vida de los objetos

- El contenedor deberá proveer una forma de búsqueda de las referencias de los objetos que maneja o administra.

- El contenedor deberá proporcionar una manera constante de configurar el funcionamiento de los objetos dentro de él y permitir la parametrización.

- El contenedor deberá manejar o administrar las relaciones entre los objetos administrados por dicho contenedor**.**

Por otra parte, existen una serie de valores potenciales que debería tener todo contenedor según lo explica Rod Johnson [RJ04]:

- Manejo de transacciones declarativas.

- Manejo de hilos de ejecución o comúnmente llamados threads para el acceso a los objetos administrados por el contenedor.

- Pool de objetos.

- Soporte de clustering.

- Servicios de remoting (acceso remoto de objetos). Exponer servicios remotos proveyendo acceso remoto a los objetos que están corriendo en el contenedor y consumo transparente para acceder remotamente a dichos objetos**.**

Para finalizar, Rod Johnson [RJ04] expone en su libro lo que él espera que caracterice a un contenedor ligero:

- Que pueda manejar el código de la aplicación pero que no imponga ninguna dependencia especial a dicho codigo, en otras palabras, que no sea intrusivo.

- Que tenga una rápido start-up

- Que no tenga pasos especiales para desplegar los objetos en el.

- Que pueda correr en varios ambientes como contenedores web, clientes standalone, etc.

3.7 La arquitectura de Spring

El BeanFactory representa el corazón de Spring, por lo cual es de suma importancia conocer como trabaja. A continuación se detallará como es el funcionamiento básico del las dos clases de Java más importantes del framework: BeanFactory y el ApplicationContext y del ciclo de vida de un objeto dentro del contenedor. Por otro lado se explicará la lógica necesaria para crear beans del tipo singleton. Para finalizar profundizaremos en la inversión de control, como es que trabaja y la simplicidad que brinda a los desarrolladores [MR05] [RJ05].

3.7.1 BeanFactory y ApplicationContext

Como bien lo indican Mat Raible [MR05], Rod Johnson [RJ05] y el sitio de Internet [INT112], el BeanFactory es el contenedor propiamente dicho, el cual instancia, configura y administra los beans de la aplicación. El BeanFactory es representado mediante una interfase de Java. El desarrollador, para poder interactuar con el BeanFactory deberá usar alguna implementación de la interfase ApplicationContext, siendo una subclase del BeanFactory. En otras palabras tanto el BeanFactory como el ApplicationContext son interfases: La primera proveerá un mecanismo avanzado de configuración capaz de administrar objetos de cualquier naturaleza. Es la interfase central que tiene como responsabilidades:

- Instanciar los objetos de la aplicación.
- Configurar los objetos.
- Inyectar las dependencias entre los objetos**.**

Esta interfase proporciona la configuración y funcionalidad básica [SR07]. Mientras que la interfase ApplicationContext (que extiende de BeanFactory), añade nueva funcionalidad extra (ej, integración con AOP entre otras).

Por otra parte, Craig Walls y Walls Ryan Breidenbach en su libro [WB05], resaltan que para obtener todo el poder del contenedor Spring, se deberá usar alguna implementación del ApplicationContext en lugar de alguna implementación del Bean Factory**.**

Matt Raible [MR05] explica que el BeanFactory es una implementación del patrón de diseño llamado Factory Pattern o Patrón Factoría (mencionado en la sección 1.7.4) que aplica la inversión del control, para separar la especificación de dependencias y configuración de la aplicación, del actual código de la misma. Según [SL06], define al ApplicationContext como una especialización del Bean Factory el cual lleva un registro de todos los objetos administrados por Spring.

Para finalizar, observamos que [WB05], también hace una distinción especial entre estos dos tipos de contenedores: BeanFactory o fabrica de Beans, el cual brinda el soporte básico para la inyección de dependencias. Y el ApplicationContext que al igual que la documentación [INT51] de dicha interfase, facilita entre otras cosas, la integración con la POA (Programación Orientada a Aspectos).

Mas allá de estos dos tipos básicos de contenedores, Spring brinda muchas más implementaciones de estas dos interfases (BeanFactory y ApplicationContext). Ambas (a menos que se diga lo contrario), serán utilizadas como sinónimos de la palabra contenedor.

Las implementaciones de la interfase ApplicationContext, más usadas según [WB05] son:

- **ClassPathXmlApplicationContext**: Cargará las definiciones del context desde un XML localizado en los recursos del class path.

- **FileSystemXmlApplicationContext**: Cargará las definiciones del context desde un XML localizado en el filesystem o sistema de archivos del ordenador.

- **XmlWebApplicationContext**: Cargará las definiciones del context desde un XML localizado en una aplicacion web.

Como conclusión, [SL07] explica la importancia del ApplicationContext como el principal objeto del framework que generalmente se configura vía XML, en el cual se especifican las definiciones de los beans y sus dependencias. En el ejemplo práctico se terminará de comprender su funcionamiento y como es que se trabaja con él.

3.7.2 El ciclo de vida de un objeto

Según [MR05], el ciclo de vida de un objeto en el contenedor ligero Spring es básicamente de la siguiente manera:

El BeanFactory lleva a cabo una serie de pasos que se listaran a continuación para que un bean esté listo para su uso [WB05]. Los siguientes pasos fueron extraídos de [MR05] y [WB05]:

- Mediante la IoC (Inversión del Control), el contenedor toma el control del bean. El contenedor definirá las reglas mediante las cuales el bean operará dentro del contenedor (las reglas son las definiciones del bean). Es decir, el contendor busca la definición del bean y lo instancia (crea una nueva referencia en memoria del objeto en cuestión).

- Luego se procede a la pre-inicializacion del bean a través de sus dependencias. Spring popula (llena) las propiedades del bean especificadas en las definiciones del bean.

- Luego, si el bean implementa la interfase BeanNameAware, el BeanFactory llamará al método setBeanName() pasandole el ID (identificador) del bean.

- Luego, si el bean implementa la interfase BeanFactoryAware, el BeanFactory llamará al método setBeanFactory() pasandole una instancia de si mismo.

- Si al bean se le asocia algún BeanPostProcessor, el BeanFactory llamará al método postProcessorBeforeInitialization().

- Si al bean se le asocia algún método de inicialización, el Bean Factory llamará a este.

- Finalmente, si al bean se le asocia algún BeanPostProcessors, el BeanFactory llamará a los métodos postProcessorAfterInitialization().

Una vez inyectadas las dependencias al bean y las llamadas a los métodos anteriormente descriptos, este pasa a un estado listo, donde el bean estará listo para ser usado en la aplicación y estará disponible en el BeanFactory.

Cuando el contenedor no lo necesite más, este será removido del BeanFactory (contenedor) mediante:

- Si el bean implementa la interfase DisposeableBean, el contenedor (BeanFactory) llamará al método destroy().

- Si al bean se le ha especificado algún método personalizado (custom method), el contenedor llamará a este.

3.7.3 Breve descripción de la arquitectura de spring

Como bien lo explica [SR07], el framework Spring esta comprendido (entre otros) por los siguientes paquetes bien definidos:

`org.springframework.core`

Aquí se encontrará el BeanFactory el cual provee mecanismos sofisticados de implementación del patron Factory que nos permitirá remover la necesidad de generar objetos singleton y nos brindará la posibilidad de desacoplar la configuración y especificación de dependencias de los objetos de negocio. Según [HM05], este paquete contiene las clases necesarias para el acceso a los archivos de configuración, creación y administración de los beans y ejecución de la ID.

`org.springframework.context`

Provee soporte entre otras cosas para la internacionalizacion (I18N), carga de recursos, etc. Asi mismo, [HM05] agrega que este paquete contiene todas las clases necesarias para la ejecucion del ApplicationContext.

`org.springframework.dao`

Provee una capa de abstracción JDBC que remueve la necesidad de codificar la tediosa tarea de conexion a la base de datos [SR07]. Básicamente agrega las clases necesarias para el acceso a la BD y una capa de abstracción para las transacciones de

Spring. Permite configurar las transacciones de forma declarativa (sin programar una sola linea de código) [HM05].

`org.springframework.orm`

Provee una capa de integración con los populares O/R mappings APIs, como Hibernate, JPA, JDO, etc.

`org.springframework.aop`

Provee un soporte para POA (Programacion Orientada a Aspectos) permitiendo al desarrollador poder definir métodos interceptors y puntos de corte (pointcuts). Como bien lo indican Rob Harrop y Jan Machacek en su libro [HM05], este paquete es de gran utilidad para la definición de transacciones declarativas.

`org.springframework.web`

Provee caracteristicas Web básicas tales como funcionalidades para el upload de formularios multipart, inicialización del contenedor IoC mediante listeners (oyentes). Cuando se utilice Spring con algun otro framework como Struts o WebWork, este es el paquete que se integrará con ellos.

`org.springframework.beans`

Este paquete al igual que org.springframework.context, nos ofrece las clases básicas y necesarias para el framework. Entre otras clases se encuentra la ya conocida BeanFactory.

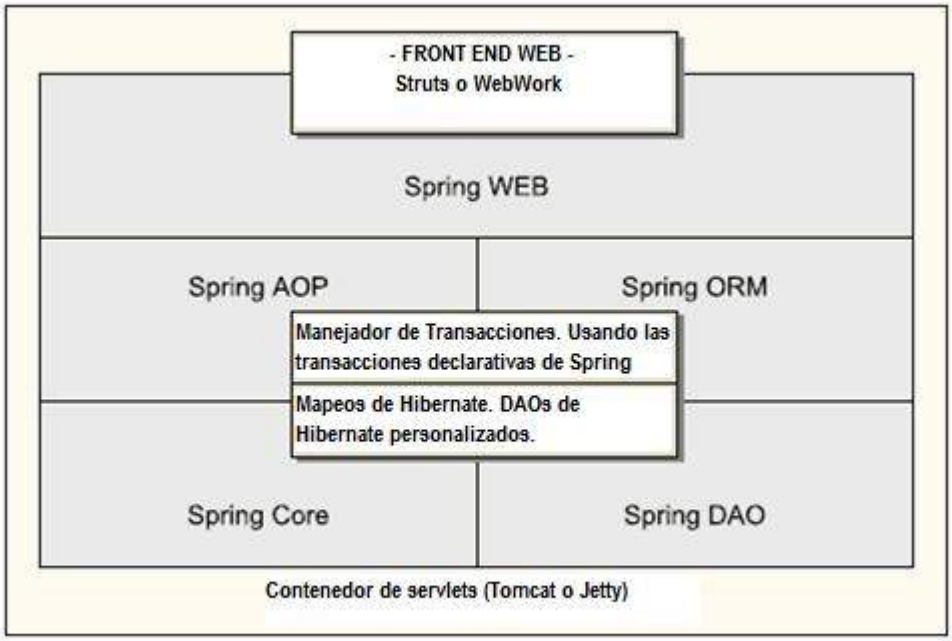

Figura 13. Módulos de Spring. [SR07]

3.7.4 Como trabaja la Inversión del Control y la Inyección de Dependencia

Según [HM05] y Rod Jonson [RJ05], la inversión del control o en su defecto la inyección de dependencias, ofrecen un simple mecanismo para inyectar a los componentes sus dependencias o colaboradores.

Al mismo tiempo, [SL06], hace hincapié, al igual que [HM05], en que la Inversión del Control es un término mucho más general que la Inyección de Dependencias y que puede ser expresada en muchos sentidos. La Inyección de Dependencias es una implementación concreta de la Inversión del Control.

Básicamente Rob Harrop y Jan Machacek en su libro [HM05] hacen una fuerte distinción en dos subtipos de Inversión del Control: Inyección de Dependencias (ID) y Búsqueda de Dependencias. Ambas hacen a la implementación de la Inversión del Control por lo cual, queda claro que, si se habla de ID, estaremos hablando de Inversión del Control pero si estamos hablando de la Inversión del Control, no necesariamente nos referiremos a la ID.

Según [SR07], este trabajo del contenedor, de inyectar las dependencias a los objetos (beans), es fundamentalmente lo contrario de lo que habitualmente se desarrollaba (donde el desarrollador tenia que llamar – inyectar - el mismo a los colaboradores de una clase), por lo cual lleva el nombre de Inversión del Control.

Como corolario, [WB05], hace mención a un artículo que Martin Fowler escribió por el año 2004 en el que se preguntaba que aspecto del control es invertido. Y concluyó que lo que se invierte es la adquisición de dependencias de los objetos. Basándose en esto, Martin Fowler adoptaria un nombre más exacto para este termino y lo llamo Inyección de Dependencias

Todos los autores coinciden que el termino Inyección de Dependencias es un termino mucho más especifico y concreto que el termino Inversión del Control.

3.8 Ejemplo práctico - Spring, arquitectura de capas y la ID

En el capitulo 5 de este trabajo podremos encontrar un ejemplo práctico que servirá para terminar de comprender el funcionamiento de Spring, la arquitectura de capas y la inyección de dependencias.

Capítulo 4 – Testing en aplicaciones de contenedores ligeros –

En los últimos años, los test unitarios han ido tomando cada vez más fuerza en el proceso de desarrollo de software [RJ04]. En la actualidad, el término test unitario es usado comúnmente para referirse a un tipo de test que aísla una unidad de código y testea su funcionalidad interna así como también sus contratos externos. Si los test unitarios son el primer paso en el proceso de desarrollo, estos se convierten en herramientas que testean código funcional que todavía no se ha escrito, siguiendo la filosofía TDD [BSA06].

4.1 La importancia de los test unitarios y de la filosofía TDD

A continuación se detallaran algunas ventajas de los test unitarios extraídos de los libros de Rod Johnson [RJ04]:

- Los test unitarios son el mejor camino para encontrar rápidamente errores en la codificación de aplicaciones.

- El desarrollo de los test ayuda a definir los requerimientos en cada una de las clases.

- El desarrollo de suite de test (conjunto de test unitarios) es importante ya que sirven como una forma de documentación del código.

- Las suite (conjuntos) de test unitarios facilitan el refactoring de las aplicaciones. Es decir, brindan seguridad en cuanto a que los cambios realizados en el código no afectan la lógica y el normal funcionamiento del sistema.

- Los test unitarios ayudan a detectar bugs (errores) de forma temprana, donde el costo de reparación es mucho más bajo.

- TDD ayuda a escribir código más simple.

- TDD ayuda a los desarrolladores a tener un constante feedback sobre el progreso de su desarrollo, bajo la siguiente idea: El desarrollador va a ir escribiendo los test, el código funcional asociado al mismo y ejecutándolos. Cuando las pruebas pasan exitosamente (demostrando la funcionalidad requerida), causan un efecto muy satisfactorio en el programador.

El termino “suite de test” implica que una porción notable del código de la aplicación este cubierta por los test unitarios. Esto a su vez garantiza un efectivo test de regresión.

Por otra parte, Seth Ladd et al. [SL06] hace mención de que los test unitarios:

- Se ejecutan rápidamente, permitiendo encontrar potenciales bugs.

- No requieren configuración externa.

- Un test unitario se ejecuta de forma totalmente independiente del resto de los test.

- Los test o pruebas unitarios testean pequeñas porciones de código como por ejemplo un método de una clase, logrando reducir el número de líneas de código evitando un potencial bug.

Como corolario de esta sección, cabe destacar que la herramienta por excelencia en el mundo de Java es conocida como JUnit (http://www.junit.org). Como bien lo indican [BSA06] y [MR05], JUnit es una de las herramientas pioneras de testeo de código y el estándar para la técnica de TDD. Es un simple framework basado en Java y en los conceptos de:

- Testeo de las clases Java

- Testeo de métodos de las clases Java

- Suite de test (testeo de varias clases Java).

4.2 Buenas prácticas

A continuación se detallaran algunas de las buenas prácticas que todo desarrollador debería emplear al momento de codificar [BSA06]:

- Escribir primero los test y luego desarrollar el código que pase correctamente el test (TDD).

- Desarrollar clases pequeñas y usar una convención de nombres apropiados lo que facilita el desarrollo del test del componente en cuestión.

- Testear todo. Ya que un simple componente puede causar errores en otros más complejos.

- Configurar los test en la etapa de building (construcción) de la aplicación.

- Mapear los test con los casos de usos. Es decir, lograr un trace o mapeo entre los requerimientos del sistema y el conjunto de test definido.

4.3 Categorías de las Herramientas de testeo

Según [BSA06], es importante destacar las diferentes categorías de herramientas de testeo. Estos conceptos también fueron extraídos de la enciclopedia digital online wikipedia [INT59] [INT60]:

- Frameworks de testing: Tales como JUnit (http://junit.org/) y TestNG (http://testng.org/). JUnit es un entorno de pruebas para Java creado por Erich Gamma y Kent Beck. Fue el primero y actualmente es el más usado.

- Herramientas de Testing de los contenedores: Permiten testear los componentes con todas las dependencias que el contenedor le agrega a dicho componente.

- Herramientas de testeo Web: Usualmente son una extensión de los frameworks de testing que permiten desarrollar test basados en el protocolo http.

- Herramientas de Coverage testing o Code Coverage (Código cubierto por los test): Estas herramientas permiten determinar que cantidad del código esta cubierto por test unitarios.

- Frameworks de objetos Mocks (objetos de maqueta): Proveen la habilidad de crear objetos Mock dinámicos para ser usados en los test unitarios.

4.4 Mock Testing (pruebas con objetos Mocks) VS. Pruebas de integración

Según Matt Raible [MR05], existen dos tipos de test: Mock test y test de integración. A continuación se realizara una breve descripción de cada uno. A lo largo del trabajo utilizará el término mock para referirse a los objetos mocks u objetos de maquetas.

4.4.1 Mock Test

Es el proceso de aislar los casos de test en una simple unidad de trabajo, es decir, en una clase. Usando un objeto mock (maqueta), se podrá testear una clase independientemente de sus dependencias.

Por ejemplo, en la instancia de una fachada de negocio (manager), donde esta tiene como dependencia a un DAO, el mock será el mismo DAO, lo que permite testear dicho componente sin su dependencia (DAO).

4.4.2 Test de integración

Es el proceso por el cual se testea una clase en un ambiente normal, es decir, con todas sus dependencias. La desventaja que esto acarrea es la velocidad y el no aislamiento del test. Aquí podemos apreciar que Matt Raible coincide con las buenas prácticas descriptas por Brian Sam-Bodden [BSA06].

Siguiendo con el ejemplo anterior, para testear una fachada de negocio (manager) que depende de un DAO, el test al incluir el DAO, deberá establecer la conexión con la base de datos.

Según Brian Sam-Bodden [BSA06], a los test de integración los llama test funcionales. Es decir, todo test que se comunica con otro sistema (como ser base de datos, otros test) no son considerados test unitarios pero si test funcionales. Aquí vemos una clara coincidencia entre los autores aunque difieren en el nombre.

Matt Raible llega a la conclusión de que los tipos de test que se deben realizar dependerán mucho del equipo de desarrollo y de la complejidad de las capas del sistema. Básicamente un buen diseño deberá tener test de integración en las capas de acceso a los datos (DAOs), mock test en la capa de servicio y capa web (especialmente para los controladores MVC) y test de integración para la capa UI (User Interfase, Interfase de Usuario) [MR05].

Por otro lado, Seth Ladd et al. [SL06] agrega que se deberá usar test unitarios simples con JUnit para la capa de dominio o negocio que a diferencia de Matt Raible no menciona dicha capa en el desarrollo de los test. Luego coincide con Matt Raible en la utilización de objetos mocks para la capa de servicio y la capa web (especialmente para los controladores).

4.5 Testeando con Spring

Según Rod Jonhson [RJ04], desarrollar los test unitarios con Spring es mucho más simple, usando directamente la herramienta JUnit. Al igual que todas las implementaciones de inyección de dependencia, Spring está diseñado para alentar a los desarrolladores a implementar los objetos de dominio como POJOs parametrizables a través de sus propiedades o constructores por medio de archivos de configuración.

Es decir, la construcción de esos objetos fuera del contenedor es simple. Se crearan nuevos test usando la herramienta JUnit sobre los POJOs del dominio directamente y se setearán las propiedades o argumentos del constructor necesarios. Gracias a esto evitaremos el uso de objetos mocks o stubs para los colaboradores del POJO al cual estamos testeando. Básicamente esto se logra ya que las dependencias de los objetos son declarados como interfases y no como clases [RJ04].

4.6 Testeando las diferentes capas

A continuación se detallarán las diferentes estrategias para testear las diferentes capas lógicas de una aplicación bien diseñada.

4.6.1 Testeando la capa de acceso a los datos (DAO) con DbUnit

La mejor manera de testear la capa de acceso a los datos es hacerlo directamente contra la base de datos. Esto asegura que el framework de persistencia (como ser Hibernate, Top Link, etc.) utilizado funcione correctamente [MR05].

Según el sitio oficial [INT57], Matt Raible [MR05] y Brian Sam-Bodden [BSA06], DbUnit es una extensión del conocido framework de pruebas de Java llamado JUnit que entre otras funcionalidades, coloca a la base de datos en un estado bien conocido entre las ejecuciones de diferentes corridas de testing. Es una buena opción cuando una prueba unitaria deja a la base de datos en un estado inconsistente o corrupto causando que las demás pruebas fallen.

Básicamente DbUnit usará archivos XML para cargar datos de prueba en la base de datos (dataset).

A continuación se mostrará un ejemplo paso a paso extraído del libro de Matt Raible [MR05] que intentará ilustrar el funcionamiento de esta herramienta:

Como ejemplo tomaremos una clase llamada **`UserTest.java`** y en particular se testeará el método llamado **`testGetUsers()`** y **`testSaveUsers()`**.

```
public class UserDAOTest extends BaseDAOTestCase {

      private User user = null;
      private UserDAO dao = null;

      protected void setUp() throws Exception {
            // inyectamos la dependencia UserDao
            dao = (UserDAO) ctx.getBean("userDAO");
      }

      protected void tearDown() throws Exception {
            dao = null;
      }
```

```
        public void testGetUsers() {
              user = new User();
              user.setFirstName("Rod");
              user.setLastName("Johnson");
              dao.saveUser(user);
              List users = dao.getUsers();
              assertTrue(users.size() >= 1);
              assertTrue(users.contains(user));
        }

        public void testSaveUser() throws Exception {
              user = new User();
              user.setFirstName("Rod");
              user.setLastName("Johnson");
              dao.saveUser(user);
              assertTrue("primary key assigned", user.getId() !=
              null);
              log.info(user);
              assertTrue(user.getFirstName() != null);
        }
  }

  public class BaseDAOTestCase {

        protected ApplicationContext ctx = null;

        public BaseDAOTestCase() {
              String[] paths = { "/WEB-INF/applicationContext*.xml"
                               };
              // Inyectamos la dependencia ApplicationContext
              ctx = new ClassPathXmlApplicationContext(paths);
        }
  }
```

Nota: Este ejemplo representa a un test de integración (no es un Mock test) ya que depende del **`ApplicationContext`** de Spring para que le inyecte la implementación del **`UserDAO`**

1. Descargamos del sitio oficial el JAR (dbunit-2-2.jar)

2. Creamos un archivo xml llamado **`simple-data.xml`** el cual emulará una tabla de la base de datos.

```
        <?xml version="1.0" encoding="utf-8"?>
        <dataset>
              <table name='app_user'>
                    <column>id</column>
                    <column>first_name</column>
```

```
                <column>last_name</column>
                <row>
                      <value>1</value>
                      <value>Rod</value>
                      <value>Johnson</value>
                </row>
          </table>
    </dataset>
```

3. Agregamos las siguientes variables de instancia a la clase

`BaseDAOTestCase.java`

```
    private IDatabaseConnection conn = null;
    private IDataSet dataSet = null;
```

4. Agregar los métodos `setUp()` y `tearDown()` a la clase

`BaseDAOTestCase.java`

```
    protected void setUp() throws Exception {
          DataSource ds = (DataSource) ctx.getBean("dataSource");
          conn = new DatabaseConnection(ds.getConnection());
          dataSet = new XmlDataSet(new FileInputStream(
          "test/data/sample-data.xml"));
          // Limpiamos la tabla e insertamos un registro.
          DatabaseOperation.CLEAN_INSERT.execute(conn, dataSet);
    }


    protected void tearDown() throws Exception {
          // Limpiamos la tabla.
          DatabaseOperation.DELETE.execute(conn, dataSet);
          conn.close();
          conn = null;
    }
```

5. Cambiamos los métodos `setUp()` y `tearDown()` de la clase

`UserDAOTest.java`

```
    protected void setUp() throws Exception {
          super.setUp();
          dao = (UserDAO) ctx.getBean("userDAO");
    }

    protected void tearDown() throws Exception {
          super.tearDown();
          dao = null;
    }
```

6. Cambiamos el método `testGetUsers()` de la clase `UserDAOTest.java`

```
public void testGetUsers() {
      List users = dao.getUsers();
      assertTrue(users.size() == 1);
      User user = (User) users.get(0);
      assertEquals(user.getFullName(), "Rod Johnson");
}
```

Para aumentar la velocidad de ejecución de los test, se deberá cargar una sola vez el contexto de Spring **"ApplicationContext".** En el ejemplo antes descrito, el contexto es cargado cada vez que se llama al método **setUp().** Este método se llamará antes de que cada método **testXXX()** sea ejecutado. Por lo que es muy buena práctica, colocar la inicialización del contexto de Spring en bloques estáticos.

Como corolario de esta sección, según Matt Raible [MR05], un buen diseño deberá tener test de integración en las capas de acceso a los datos (DAOs). También encontramos cierta coincidencia con Brian Sam-Bodden [BSA06], donde en su libro, también hace mención a esta herramienta para testear esta capa lógica.

4.6.2 Testeando la capa de servicios (Managers)

Para realizar los test en esta capa lógica, Matt Raible [MR05] propone aislar esta capa de la capa de acceso a los datos (DAOs) para lo cual hace uso de una de las técnicas de la sección 4.4: Mock Test. Es decir, se hará fuerte hincapié en como testear esta capa lógica y se simulará la implementación de los DAOs (sin acceso la base de datos).

A este nivel, se parte de la premisa que la capa de acceso a los datos está bien testeada (como vimos en la sección anterior) por lo que Matt Raible [MR05] no encuentra razón alguna para usar las implementaciones de los DAOs (con la base de datos) en esta etapa de testeo. Bajo esta misma premisa, Seth Ladd et al. [SL06] sostiene el mismo criterio que Matt Raible.

Otra razón que expone Matt Raible [MR05] de usar mock objects (objetos falsos) para simular la dependencia de una clase (como en este caso las clases de la capa de servicio) es: Si un desarrollador está en un equipo de desarrollo y cada uno es el

responsable de las diferentes capas lógicas, no es buena idea esperar a que a la implementación de las capas subsiguientes para comenzar a escribir los test unitarios.

Por otra parte, Seth Ladd et al. [SL06] agrega que el usar objetos falsos aísla la clase a testear ayudando a tener un mejor control de las dependencias e influencias externas.

Existen dos tipos de mock objects (objetos falsos) [SL06], [MR05], [RJ04]:

- Los objetos llamados stubs: Objetos que el desarrollador crea por si mismo

- Los Objetos Mocks dinámicos: Objetos que tienen más “inteligencia” y programación por detrás. Típicamente estos objetos implementaran la interfase de la dependencia de la que se desee simular su comportamiento.

Existen 3 herramientas que ayudan a crear los mock objects:

- MockObjects: http://www.mockobjects.com

- jMock: http://www.jmock.org

- EasyMock: http://www.easymock.org

Tanto Matt Raible [MR05] como Seth Ladd et al. [SL06], recomiendan jMock o EasyMock; de hecho, Spring utiliza EasyMock.

4.6.2.1 Testeando la capa de servicio (manager) con EasyMock

Como bien lo indica el sitio oficial [INT61], EasyMock es un proyecto que provee mock objects (objetos falsos) para las interfases utilizadas en los test unitarios creándolas “on the fly” (al vuelo).

A continuación se mostrará un ejemplo paso a paso extraído del libro de Matt Raible [MR05] que intentará ilustrar el funcionamiento de esta herramienta como se hizo en la seccion anterior:

Como ejemplo tomaremos una clase llamada **UserManagerTest.java** que usará mock objects (falso objeto) para referenciar al **UserDAO.java**

```
public class UserManagerTest extends TestCase {

      private ApplicationContext ctx;
      private User user;
      private UserManager mgr;

      protected void setUp() throws Exception {
            String[] paths = { "/WEB-
            INF/applicationContext*.xml" };
            // Inyectamos la dependencia ApplicationContext
            ctx = new ClassPathXmlApplicationContext(paths);
            mgr = (UserManager) ctx.getBean("userManager");
      }

      protected void tearDown() throws Exception {
            user = null;
            mgr = null;
      }

      public void testAddUser() throws Exception {
            user = new User();
            user.setFirstName("Easter");
            user.setLastName("Bunny");
            user = mgr.saveUser(user);
            assertNotNull(user.getId());
      }

}
```

Como podemos ver, este ejemplo nos muestra un típico caso de test de integración. Además podemos ver que este test no solo cargará la dependencia del **UserDao.java** de la clase **UserManager.java** sino que también cargará al **ApplicationContext.java** con toda la API de Spring. A continuación transformaremos esto en un caso de test unitario, en donde la dependencia del **UserManager.java** con **UserDao.java** será simulada con EasyMock.

```
public class UserManagerEMTest extends TestCase {

      private UserManager mgr = new UserManagerImpl();
      private MockControl control;
```

```
private UserDAO mockDAO;

protected void setUp() throws Exception {

      // Inyectamos el Controlador de EasyMock que
      // simulará la implementación de la interfase
      // UserDao.java
      control= MockControl.createControl(UserDAO.class);
      mockDAO= (UserDAO) control.getMock();
      mgr.setUserDAO(mockDAO);
}

public void testAddUser() throws Exception {

      User user = new User();
      user.setFirstName("Easter");
      user.setLastName("Bunny");

      // Seteamos el comportamiento para el objeto mock //
      DAO
      mockDAO.saveUser(user);
      control.setVoidCallable();

      // Activamos el objeto mock
      control.replay();

      // testeamos el método en cuestion.
      user = mgr.saveUser(user);
      assertEquals(user.getFullName(), "Easter Bunny");
      control.verify();
}
}
```

Matt Raible en su libro [MR05] realizó una comparación en cuanto a los tiempos de ejecución de ambos test demostrando que el test con mock objects `UserManagerEMTest.java` es mucho más rápido que el test de integración `UserManagerTest.java`.

4.6.3 Testeando la capa Web (testeando la vista y los controladores)

Según Matt Raible [MR05], el desarrollo de las pruebas en dicha capa es la parte más importante en la suite de test. Al testear los controladores, se verificará el flujo de control de las aplicaciones y se determinará si las entradas y salidas son las esperadas.

Por otro lado, si bien testear los controladores es importante, testear la vista interactuando con la UI (interfase de usuario) es usualmente el mejor camino para asegurar que la aplicación funcione correctamente. Para esto se deberá disponer de un apropiado departamento de QA (Quality Assurance, Aseguramiento de la Calidad). Sin embargo es fácil escribir test unitarios para la UI usando jWebUnit y Canoo WebTest, ambos son simplificaciones de HttpUnit. [MR05]. No se entrará en detalle sobre estas herramientas.

En caso de usar Struts como web framework, se podrá utilizar la herramienta llamada StrutsTestCase para facilitar el testeo. En caso de usar el framework MVC de Spring, necesitaremos la librería spring-mock.jar [MR05]

A continuación se mostrará un ejemplo extraído del libro de Matt Raible [MR05] que intentará ilustrar el funcionamiento de la herramienta StrutsTestCase. Como ejemplo tomaremos una clase llamada **`UserActionTest.java`** que pertenece al grupo de los test de integración.

```
public class UserActionTest extends MockStrutsTestCase {

      public UserActionTest(String testName) {
            super(testName);
      }

      public void addUser() {
            // Seteamos el action a ejecutar. En este caso
            // vamos a ejecutar la accion que persiste los
            // usuarios llamada user.do
            setRequestPathInfo("/user");
            // A esta accion, vamos a pasarle parámetros
            por                                          //
            request. Donde la vida de esos parámetros sera
            // lo que dure la petición HTTP.
            addRequestParameter("method", "save");
            addRequestParameter("user.firstName", "Juergen");
            addRequestParameter("user.lastName", "Hoeller");
            // Ejecuta el método execute() de la accion
            // user.java. Esto simularia la siguiente URL:
            // http://localhost:8080/listUsersAction.do
            actionPerform();
            // Se verifica que este action llame correctamente // a
            la jsp definida en list
```

```
            verifyForward("list");
            // Se verifica que no haya habido errores.
            verifyNoActionErrors();
      }

      public void testAddAndEdit() {
            addUser();
            addRequestParameter("method", "edit");
            addRequestParameter("id", "1");
            actionPerform();
            verifyForward("edit");
            verifyNoActionErrors();
      }

      public void testAddAndDelete() {
            addUser();
            setRequestPathInfo("/user");
            addRequestParameter("method", "delete");
            addRequestParameter("user.id", "1");
            actionPerform();
            verifyForward("list");
            verifyNoActionErrors();
      }


      public void testList() {
            addUser();
            setRequestPathInfo("/user");
            addRequestParameter("method", "list");
            actionPerform();
            verifyForward("list");
            verifyNoActionErrors();
            // Pide al request (petición HTTP) el atributo "users"
            que es dejado por la accion /user)
            List users = (List)
            getRequest().getAttribute("users");
            assertNotNull(users);
            assertTrue(users.size() == 1);
      }
}
```

Así también, mostraremos otro ejemplo extraído del libro de Matt Raible [MR05] que intentará ilustrar el funcionamiento de la herramienta MVC de Spring. Como ejemplo tomaremos una clase llamada **`UserControllerTest.java`** que también pertenece al grupo de los test de integración.

```
public class UserControllerTest extends TestCase {

      private XmlWebApplicationContext ctx;

      public void setUp() {
            String[] paths = {"/WEB-INF/applicationContext.xml",
```

```
            "/WEB-INF/action-servlet.xml"};
            ctx = new XmlWebApplicationContext();
            ctx.setConfigLocations(paths);
            ctx.setServletContext(new MockServletContext(""));
            ctx.refresh();
      }

      public void testGetUsers() throws Exception {
            UserController c = (UserController)
            ctx.getBean("userController");
            ModelAndView mav =
            c.handleRequest((HttpServletRequest) null,
            (HttpServletResponse) null);
            Map m = mav.getModel();
            assertNotNull(m.get("users"));
            assertEquals(mav.getViewName(), "userList");
      }
}
```

Vale la pena aclarar que el método **`testGetUsers()`**, invocará al método **`handleRequest()`**, el cual devolverá una clase llamada **`ModelAndView.`** Básicamente esta clase tendrá la próxima página web (ej., html o jsp) a desplegarse en el browser (la vista) y los datos necesarios de dicha pagina (el modelo). Haciendo una analogía con Struts, este método **`handleRequest()`** es el equivalente en Struts al **`execute().`** Por otra parte, en Struts la vista y el modelo están separados. Típicamente el modelo se coloca en el request (petición) o session (sesión) del HTTP y los Actions retornaran la vista (**`ActionForwards`**) [MR05].

Como corolario de esta sección, podemos observar que tanto para los Actions de Struts como para los Controladores de Spring MVC no se usaron mock objects.

Capítulo 5 – Ejemplo práctico

En este apartado, se tratará de demostrar mediante un ejemplo práctico todo lo explicado en este trabajo. La idea es que sirva como ejemplo concreto de lo que se explica como arquitectura de capas y la utilización de Spring como entorno de trabajo e implementación de la inyección de dependencias.

En este ejemplo demostraremos el uso de Struts como framework MVC para la capa de presentación o front-end, Spring como framework de aplicación y Hibernate como framewok para la capa de acceso a los datos o de persistencia.

Por ultimo se usará la técnica de desarrollo dirigida por test o comúnmente llamada TDD. Este ejemplo, cubrirá los siguientes grandes aspectos:

- Desarrollo y configuración de los módulos y sus dependencias.

- Organización de los fuentes Java.

- Configuración de Spring con Hibernate y Struts mediante los archivos ApplicationContext.xml, struts-config.xml y servlet-action.xml. Minimamente se requiere de algunos conocimientos básicos de Struts y Hibernate para lograr una compresión adecuada de este ejemplo.

- Desarrollo de los test unitarios.

- Desarrollo de las clases funcionales y la Inyección dependencias entre los POJOs.

A continuación, se desarrollará un ejemplo básico de administración de usuarios – CRUD (Create, Retrive, Update and Delete, Alta, Consulta, Actualización y Baja) -. Este ejercicio recibirá el nombre de **BesyApp**. El diagrama presentado a continuación nos presenta una breve descripción de uno de los casos de uso: *Consulta de los usuarios del sistema* como asi también, como será la arquitectura lógica de este ejemplo:

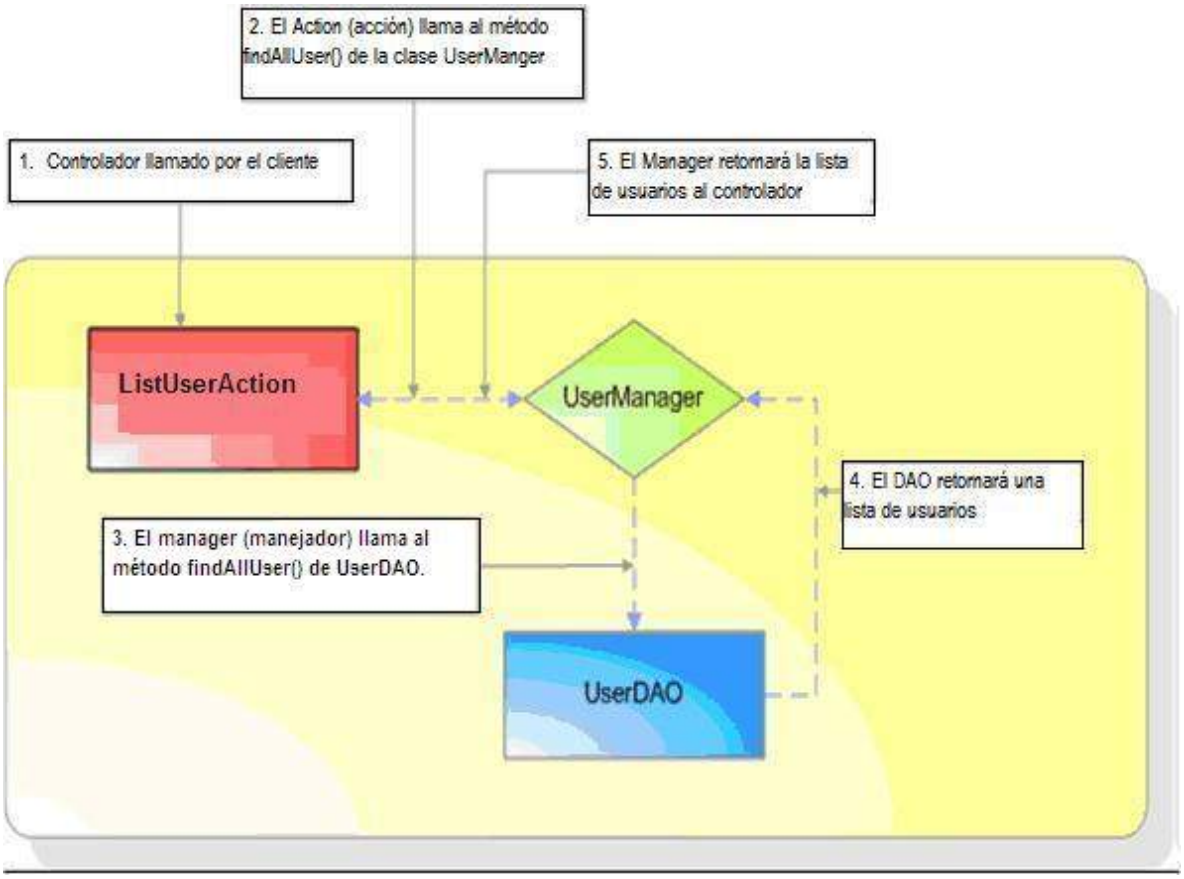

Figura 14. Arquitectura del ejemplo práctico. [MR05]

5.1 Módulos y dependencias

El proyecto BesyApp consistirá en los módulos que serán descriptos a continuación:

- *BesyAppWeb (web module, módulo web):*

Representa al Front end- de la aplicación. Integra los siguientes Frameworks o entornos de trabajo:

- Struts: implementación del patrón MVC - framework de presentación-
- Spring: framework de aplicacion - framework multi capa -

Librerias de terceras partes necesarias:

/BesyThird-party/libraries/aspectj-1.5.2a/aspectjweaver.jar
/BesyThird-party/libraries/jakarta-commons/commons-beanutils-1.7.jar
/BesyThird-party/libraries/jakarta-commons/commons-collections-3.2.jar
/BesyThird-party/libraries/jakarta-commons/commons-digester-1.7.jar
/BesyThird-party/libraries/jakarta-commons/commons-fileupload-1.1.1.jar
/BesyThird-party/libraries/jakarta-commons/commons-lang-2.1.jar
/BesyThird-party/libraries/jakarta-commons/commons-logging-1.1.jar
/BesyThird-party/libraries/jakarta-commons/commons-validator-1.3.0.jar
/BesyThird-party/libraries/struts/jakarta-oro.jar
/BesyThird-party/libraries/struts/struts.jar
/BesyThird-party/libraries/Junit/junit.jar
/BesyThird-party/libraries/Spring2.0/spring.jar
/BesyThird-party/libraries/strutstestcase/strutstest-2.1.3.jar
/BesyThird-party/libraries/Displaytag-1.0/displaytag-1.0.jar

- *BesyAppService (Java module, módulo Java):*

Representa el punto de entrada a la lógica de la aplicación expuesta en forma de servicios. Aquí se encontrarán todos los managers (manejadores) - services (servicios) - a los que podremos acceder desde el módulo anterior-

Librerias de terceras partes necesarias:

Vemos que en este módulo no necesitamos de las librerías de struts (framework de presentación), pero sí necesitamos de las librerías de Spring (framework multicapa)

/BesyThird-party/libraries/mail/activation.jar
/BesyThird-party/libraries/mail/mail.jar

/BesyThird-party/libraries/jakarta-commons/commons-beanutils-1.7.jar
/BesyThird-party/libraries/jakarta-commons/commons-collections-3.2.jar
/BesyThird-party/libraries/jakarta-commons/commons-digester-1.7.jar
/BesyThird-party/libraries/jakarta-commons/commons-fileupload-1.1.1.jar
/BesyThird-party/libraries/jakarta-commons/commons-lang-2.1.jar
/BesyThird-party/libraries/jakarta-commons/commons-logging-1.1.jar
/BesyThird-party/libraries/jakarta-commons/commons-validator-1.3.0.jar
/BesyThird-party/libraries/Firebird/firebirdsql-full.jar
/BesyThird-party/libraries/Junit/junit.jar
/BesyThird-party/libraries/Log4J/log4j-1.2.8.jar
/BesyThird-party/libraries/MySql/mysql-connector-java-3.1.11-bin.jar
/BesyThird-party/libraries/Spring2.0/spring.jar

- *BesyAppDomain (Java module, módulo Java):*

  Representa al módulo del modelo o entidades de negocio.

Librerias de terceras partes necesarias:

/BesyThird-party/libraries/jakarta-commons/commons-collections-3.2.jar

- *BesyAppHibernate (Java module, módulo Java):*

Representa al módulo de acceso a los datos. Aquí encontraremos la implementación de las interfases de los DAOs. La implementación de los DAOs será llevada a cabo, como el nombre del módulo lo indica, mediante el framework Hibernate.

Librerias de terceras partes necesarias

/BesyThird-party/libraries/Hibernate3.2rc2/ant-antlr-1.6.5.jar
/BesyThird-party/libraries/Hibernate3.2rc2/antlr-2.7.6.jar
/BesyThird-party/libraries/Hibernate3.2rc2/asm-attrs.jar
/BesyThird-party/libraries/Hibernate3.2rc2/asm.jar

/BesyThird-party/libraries/Hibernate3.2rc2/cglib-2.1.3.jar
/BesyThird-party/libraries/Hibernate3.2rc2/dom4j-1.6.1.jar
/BesyThird-party/libraries/Hibernate3.2rc2/ehcache-1.2.jar
/BesyThird-party/libraries/Hibernate3.2rc2/commons-collections-2.1.1.jar
/BesyThird-party/libraries/Hibernate3.2rc2/hibernate3.jar
/BesyThird-party/libraries/Hibernate3.2rc2/jta.jar
/BesyThird-party/libraries/Hibernate3.2rc2/log4j-1.2.11.jar
/BesyThird-party/libraries/hibernateAnnotation/ejb3-persistence.jar
/BesyThird-party/libraries/hibernateAnnotation/hibernate-annotations.jar

- *BesyCore (Java module, módulo Java):*

Este módulo representará las clases que serán de uso diario para cualquier aplicación. Son clases de utilidades que escapan al dominio en cuestión y servirán para cualquier proyecto.

Es muy importante que cuando se agreguen clases en dicho módulo, se haga su correspondiente test unitario no solo porque es una buena práctica, sino también, por ser un mecanismo de auto documentación para comprender su uso.

5.2 Grafico de dependencias entre los módulos

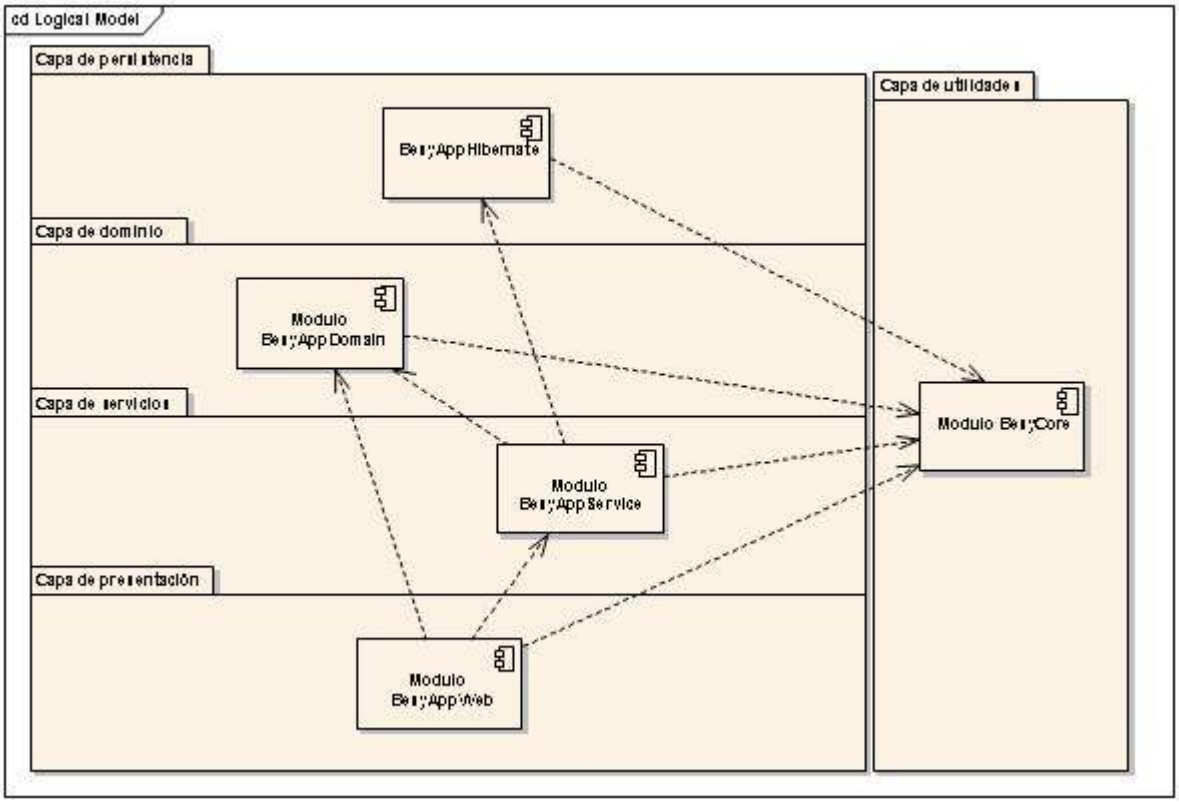

Figura 15. Dependencias entre módulos

5.3 Organización lógica de los fuentes Java

En la carpeta de fuentes de cada módulo, encontraremos los elementos típicos de una aplicacion J2EE: fuentes Java que forman parte del código a desplegar, test unitarios, recursos de distintos tipos (Ej. Archivos de propiedades (properties), xml, imágenes, etc.).

Dependiendo del tipo de módulo de que se trate, es posible llegar a tener cuatro directorios lógicos (solo en los módulos web, mientras que en los módulos Java tendremos solamente dos).

- src (siempre): fuentes Java.

- test (siempre): fuentes de los test unitarios.

- resources (solo en el módulos web): archivos de propiedades y xml de configuración.

- web (solo en el módulo web): ídem al resources más las JSPs.

Packages (Paquetes)

A grandes rasgos podemos dividir los fuentes en:

- Actions o Controllers (Acciones o Controladores): Punto de entrada desde la capa de presentación. No contiene lógica de negocio. Solo actuará como dispatcher.

- Forms o Models (Formularios o Modelos): Datos ingresados por el usuario o enviados al usuario mediante la capa de presentación. Representación de los datos de la vista.

- Manager o Services (Manejadores o Servicios): Punto de entrada a la lógica de negocio. Servicios que son accedidos desde la capa de presentación.

- DAO: Punto de entrada a los datos. Acceso a la BD. Finders y métodos CRUD que son accedidos desde los objetos de negocio y/o servicios.

- Entities o Domain (Entidades o Dominio): Objetos reales persistentes de negocio.

5.4 Configuración de Spring con Hibernate y Struts.

Luego de entender como estará diseñada arquitectónicamente la aplicación, procederemos a configurar Spring, específicamente lo que se conoció en el capitulo 3 sección 3.7.1 como **`ApplicationContext`**. Toda aplicación con Spring deberá tener un archivo en donde se configura los beans y sus dependencias (entre otras cosas).

A continuación detallaremos el archivo **`ApplicationContext.xml`**, examinaremos como se configuran los beans de la aplicación y como se integra Struts con Spring y Hibernate mediante este archivo**.**

ApplicationContext.xml

… /BesyWebApp/web/WEB-INF/applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE beans PUBLIC
"-//SPRING//DTD BEAN//EN"
                                  "http://www.springframework.org/dtd/spr
ing-beans.dtd">
<beans>
    <!-- **************** 1 POJOs ****************  -->
    <bean id="restrictionsBuilder" class="ar.dao.builder.
                                              RestrictionsBuilderImpl">
    </bean>
    <bean id="user" class="ar.besy.model.User" scope="prototype">
    </bean>
   <!-- **************** 2 DAOS ****************  -->
    <bean id="userDAO" class="ar.besy.dao.hibernate.UserDAOHibernate">
        <property name="restrictionsBuilder">
           <ref bean="restrictionsBuilder"/>
        </property>
        <property name="sessionFactory">
           <ref local="sessionFactory"/>
        </property>
    </bean>
   <!-- **************** 3 MANAGERS ****************  -->
```

```
 <bean id="userManager" class="ar.besy.manager.cmsUser.
                                                    UserManagerImpl">
     <property name="userDAO"><ref bean="userDAO"/></property>
     <property name="transactionManager">
        <ref bean="transactionManager"/>
     </property>
 </bean>
<!-- *************** 4 TRANSACCIONES Hibernate *************** -->
 <bean id="transactionManager" class= "org.springframework.orm.
                              hibernate3.HibernateTransactionManager">
     <property name="sessionFactory">
        <ref local="sessionFactory"/>
     </property>
 </bean>
<!-- ***************** 5 HIBERNATE SessionFactory **************  -->
 <bean id="sessionFactory" class="org.springframework.orm.hibernate3
                                              .LocalSessionFactoryBean">
     <property name="dataSource">
         <ref local="dataSourceMySqlTest"/>
     </property>
     <property name="mappingResources">
         <list>
             <value>ar/besy/model/user.hbm.xml</value>
         </list>
     </property>
     <property name="hibernateProperties">
         <props>
             <prop key="hibernate.dialect">
                   org.hibernate.dialect.MySQLDialect
              </prop>
             <!--prop key="hibernate.hbm2ddl.auto">create</prop-->
         </props>
     </property>
 </bean>
<!-- ***************** 6 MySql Datasource ***************** -->
 <bean id="dataSourceMySql" class="org.springframework.jdbc.
                                datasource.DriverManagerDataSource">
     <property name="driverClassName">
```

```
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost/CMS_ADMIN</value>
        </property>
        <property name="username">
            <value>root</value>
        </property>
        <property name="password">
            <value></value>
        </property>
    </bean>
</beans>
```

En el `ApplicationContext.xml` encontramos 6 secciones bien identificadas que pasaremos a detallar a continuación:

Antes de sumergirnos en las secciones, explicaremos algunos tags (etiquetas) utilizadas en este xml.

El tag bean es el tag que más se usa en este archivo, que junto con otros dos atributos que veremos a continuación, simulan la creación de una instancia de una clase de Java. Es decir, estos tags tienen un **`id`** y un **`class`** que al setear ambos, es similar a crear una nueva instancia de ese objeto, cuyo nombre de instancia será el definido por el atributo **`id`** y cuya clase sera la definida por el atributo **`class`** [SR07].

En el archivo xml encontramos entre otros beans definidos, uno que tiene el siguiente formato: **`<bean id="user" class="ar.besy.model.User" scope= "prototype">`** el cual nos indica que estaríamos creando una nueva instancia de la clase **`User`** llamada **`user`**. Al mismo tiempo, vemos que también tiene un atributo llamado scope el cual puede tomar dos valores **`prototype`** o **`singleton`**. Si optamos por el primero, el framework nos devolverá una nueva instancia del bean por cada llamada, mientras que si optamos por el segundo, Spring retornará siempre la misma instancia del bean en cuestión.

Cabe destacar que toda referencia a una instancia de un bean (mediante el atributo id del tag **`bean`**) debe estar previamente definida en este archivo.

El tag **`property`** nos indica que ese bean tiene una dependencia como variable de instancia que sera inyectada mediante la IoC. En el archivo xml podemos ver el siguiente bean con dos dependencias [SR07]:

```
<bean id="userManager" class="ar.besy.manager.cmsUser.
                                                UserManagerImpl">
    <property name="userDAO"><ref bean="userDAO"/></property>
    <property name="transactionManager">
       <ref bean="transactionManager"/>
    </property>
</bean>
```

El tag **`ref`** nos indica que la propiedad de un bean (dependencia) hace referencia a otro bean dentro del mismo container (puede que este o no en el mismo xml) [SR07].

Recordamos que todo lo que definamos en este archivo será visto como un bean para Spring.

1 POJOS:

En esta sección y para este ejemplo práctico, definimos dos clases (beans para el framework Spring) del dominio de nuestro sistema: La clase User y `RestrictionsBuilderImpl` que a su vez no tienen dependencia alguna (no tienen propiedades).

2 DAOS:

En esta sección, definiremos a los objetos que tendrán la responsabilidad de acceder a los datos o comúnmente llamados DAOs. En este ejemplo definiremos la clase llamada `UserDAOHibernate` con una propiedad o dependencia como variable de instancia: `sessionFactory` que el contenedor inyectará mediante la IoC.

3 MANAGERS:

En esta sección, definiremos a los objetos que tendrán la responsabilidad de actuar como servicios, o comúnmente llamados Managers o Services (Manejadores o Servicios). En este ejemplo definiremos la clase llamada `UserManager` con dos propiedades o variables de instancia llamadas `userDAO` y `transactionManager`.

4 TRANSACCIONES Hibernate:

En esta sección, definiremos a los objetos que tendrán la responsabilidad de administrar las transacciones contra la base de datos. En este ejemplo definiremos la clase llamada `HibernateTransactionManager`. Esta clase ya viene implementada con la API de Spring para el manejo de transacciones con Hibernate. Por este motivo no se necesitará desarrollo alguno. A su vez, definiremos una propiedad o variable de instancia llamada **`sessionFactory`**.

Cabe destacar la importancia de la clase `HibernateTransactionManager` que usaremos para el manejo de transacciones. Como bien lo explica Matt Raible [MR05], esta implementación permite vincular las sesiones de Hibernate para un mismo thread (hilo) de ejecución.

Por otra parte, nada impide el uso de JTA como manejador de transacciones, siempre y cuando necesitemos administrar transacciones distribuidas sobre dos o mas bases de datos. En este caso, utilizaremos otra implementación que viene con la API de Spring llamada **`JtaTransactionManager`**.

5 HIBERNATE SessionFactory:

Spring facilita la búsqueda (lookup) de los recursos de una aplicación, permitiendo definir recursos como beans por ejemplo: Una fuente de datos (datasource) JDBC o como en este caso, un SessionFactory de Hibernate [SR07].

En esta sección, encontraremos a los objetos que tendrán la responsabilidad de interactuar con la base de datos y realizar los mapeos entre las entidades y las tablas. En

este ejemplo definiremos la clase llamada `LocalSessionFactoryBean` la cual ya viene como parte de la API del framework Spring y a la que le tendremos que setear básicamente 3 propiedades. No hará falta implementar dicho bean ni sus propiedades programaticamente vía código Java, sino que se definen declarativamente en este xml:

- `dataSourceMySqlTest`: Es una implementación de un Datasource que deberá estar definido en el `applicationContext.xml`.

- `mappingResources`: En donde configuramos los archivos de hibernate conocidos como *.hbm.xml. En estos archivos haremos el mapeo objeto – tabla. Aquí encontraremos: `User.hbm.xml`.

- `hibernateProperties`: En donde definiremos la configuración propia de Hibernate como por ejemplo, qué dialecto deberá usar Hibernate para traducir las consultas SQL y si recrea el esquema de BD "on-the-fly" cada vez que iniciamos la aplicación, entre otras cosas.

6 MySql Datasource

En esta sección, definiremos los objetos que tendrán la responsabilidad de conectarse directamente con la base de datos mediante algún driver (manejador) de conectividad, como se hace en cualquier aplicación Java que desea conectarse a una base de datos.

Aquí encontraremos la clase llamada `DriverManagerDataSource` a la que le tendremos que setear 4 propiedades. No hará falta implementar dicho bean y sus propiedades vía código Java, sino que se definen declarativamente en este xml:

- **`driverClassName`**: En donde indicaremos la ruta del nombre de la clase Java que actúa como Driver de conexión con la base de datos. Aquí es donde registramos el controlador con el DriverManager**.**

- **`url`**: Una vez especificado el driver, se deberá precisar la ruta de la fuente de datos. En JDBC una fuente de datos se determina por medio de una

URL con el prefijo de protocolo JDBC y un subprotocolo que será el tipo de controlador como ser db2, mysql, oracle**.**

- **username**: En donde especificaremos el nombre de usuario de la base de datos.

- **password**: En donde especificaremos el password de la base de datos**.**

struts-config.xml

… /BesyWebApp/web/WEB-INF/struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC"-//Apache Software Foundation//DTD Struts
Configuration 1.2//EN""http://jakarta.apache.org/struts/dtds/struts-
config_1_2.dtd">

<struts-config>

     <!-- ****************Form Bean Definitions *************** -->
     <form-beans>
         <form-bean name="userForm"  type="ar.besy.cliente.web.usuario.
                                                    view.UserForm"/>
     </form-beans>
     !-- *************** Global Forward Definitions *************** -->
     global-forwards>
        <forward name="Welcome" path="index.jsp"/>
        <forward name="globalFail" path="home.global.error"/>
     </global-forwards>
     !-- *************** Action Mapping Definitions *************** -->
     <action-mappings type="ar.security.SecureActionMapping">

       <!-- ************* CRUD USUARIO ************* -->
       <action path="/userAction" name="userForm" type="org.
                springframework.web.struts.DelegatingActionProxy"
                      scope="session">
            <forward name="success" path="user.form" redirect="false"/>
            <forward name="failed" path="/listUserAction.do"
                                                    redirect="false"/>
       </action>
        <action path="/listUserAction" type="org.
                      springframework.web.struts.DelegatingActionProxy"
                      scope="session">
            <forward name="success" path="user.list" redirect="false"/>
        </action>

        <action path="/addOrUpdateUserAction" name="userForm"
                            type="org. springframework.web.struts.
```

```
                            DelegatingActionProxy" scope="session">
            <forward name="success" path="/listUserAction.do"
                                                    redirect="false"/>
        </action>
        <action path="/deleteUserAction" type="org.
                    springframework.web.struts.DelegatingActionProxy"
                    scope="session">
            <forward name="success" path="/listUserAction.do"
                                                      redirect="false"/>
        </action>
    </action-mappings>
     !-- **************** Controller Configuration *************** -->
    <controller
       processorClass="org.apache.struts.tiles.TilesRequestProcessor"/>
    <!-- **************** Plug Ins Configuration *************** -->
    <!-- **************** Tiles plugin *************** -->
    <plug-in className="org.apache.struts.tiles.TilesPlugin" >
        <!-- Path to XML definition file -->
        <set-property property="definitions-config"
                    value="/WEB-INF/tiles-defs.xml" />
        <!-- Set Module-awareness to true -->
        <set-property property="moduleAware" value="true" />
    </plug-in>
    <!-- **************** Validator plugin *************** -->
    <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames"
                    value="/WEB-INF/validator-rules.xml,/WEB-
                     INF/validation.xml"/>
    </plug-in>
    <!-- **************** Spring plugin *************** -->
    <plug-in className="org.springframework.web.
                                        struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
                    value="/WEB-INF/applicationContext.xml,
                           /WEB-INF/action-servlet.xml"/>
    </plug-in>
</struts-config>
```

Según la enciclopedia Wikipedia [INT115] y el sitio de Internet [INT114], al momento de desarrollar aplicaciones web, siguiendo el patrón MVC, surge la típica duda si tener varios controladores o un solo, para atender las peticiones http del usuario. Si optamos por tener un solo controlador, este va a tener encapsulado toda la lógica del manejo de peticiones, convirtiéndose en lo que se conoce como “fat controller (controlador pesado)”. Como solución a este problema surge Struts, implementando un solo controlador llamado ActionServlet que evalúa las peticiones http del usuario mediante un archivo de configuración llamado **`struts-config.xml`.** Es importante aclarar que esta clase es concreta, con lo cual, el desarrollador no necesita extenderla para comenzar a utilizar el framework, sino que indicará al mismo las acciones, los forms (formularios) y otros atributos mediante este archivo de configuración que por convención se llama

struts-config.xml. Aquí podemos apreciar todas las acciones pertinentes a CRUD (Create, Retrive, Update, Delete – Alta, Consulta, Actualización, Eliminación).

action-servlet.xml

… /BesyWebApp/web/WEB-INF/action-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- Actions mapeados como beans para ser usados por Spring -->
     <bean name="/userAction" class="ar.besy.cliente.web.
         usuario.controller.UserAction" singleton="false">
         <property name="userManager">
                 <ref bean="userManager"/>
         </property>
     </bean>
    <bean name="/listUserAction" class="ar.besy.cliente.web.
              usuario.controller.ListUserAction" singleton="false">
         <property name="userManager">
               <ref bean="userManager"/>
         </property>
    </bean>
   <bean name="/addOrUpdateUserAction" class="ar.besy.cliente.web.
          usuario.controller.AddOrUpdateUserAction" singleton="false">
         <property name="userManager">
           <ref bean="userManager"/>
         </property>
    </bean>
    <bean name="/deleteUserAction" class="ar.besy.cliente.web.
                usuario.controller.DeleteUserAction" singleton="false">
         <property name="userManager">
           <ref bean="userManager"/>
           </property>
    </bean>
</beans>
```

Este archivo de configuración forma parte del **ApplicationContext.xml** explicado anteriormente, ya que toda aplicación con Spring deberá tener un archivo en donde se configura los beans y sus dependencias (entre otras cosas). Para este ejemplo, encontramos dos archivos para este propósito: El **ApplicationContext.xml** y **action-servlet.xml.** En el **action-servlet.xml** tendremos los beans que mapearán con los controladores o acciones de struts.

5.5 Desarrollo de los test unitarios.

5.5.1 Pruebas unitarias para la capa de persistencia (módulo BesyAppHibernate)

Para realizar las pruebas unitarias de los DAOs utilizaremos, como se explicó en el capitulo 4, la herramienta DbUnit, para lo cual creamos un archivo llamado `simple-data.xml` el que emulará una tabla de la base de datos.

```
<?xml version="1.0" encoding="utf-8"?>
<dataset>
      <table name='app_user'>
            <column>id</column>
            <column>username</column>
            <column>password</column>
            <column>enabled</column>
            <row>
                  <value>1</value>
                  <value>Rod Johnson</value>
                  <value>12345678</value>
                  <value>true</value>
            </row>
      </table>
</dataset>
```

Creamos la clase `BaseDAOTestCase.java` de la cual podrán extender todas las pruebas unitarias que se hagan con los DAOs. Esto permitirá resolver en una única vez varias cuestiones que tienen que ver con Spring y DbUnit y que gracias a la herencia de POO podrán ser reutilizadas sin tener la necesidad de duplicar código.

```
public class BaseDAOTestCase {

      protected ApplicationContext ctx = null;
      private IDatabaseConnection conn = null;
      private IDataSet dataSet = null;


      public BaseDAOTestCase() {
            String[] paths = { "/WEB-INF/applicationContext*.xml"
                             };
```

```
            // inyectamos la dependencia del container:
            // AplicationContext
            ctx = new ClassPathXmlApplicationContext(paths);
      }

      protected void setUp() throws Exception {
            // Le pedimos al applicationContext.xml el datasource
            // referenciado con el ID  "dataSourceMySql"
            DataSource ds = (DataSource)
            ctx.getBean("dataSourceMySql");
            conn = new DatabaseConnection(ds.getConnection());
            dataSet = new XmlDataSet(new FileInputStream(
            "test/data/sample-data.xml"));
            // Limpiamos la tabla e insertamos los registros
            // definidos en simple-data.xml.
            DatabaseOperation.CLEAN_INSERT.execute(conn, dataSet);
      }


      protected void tearDown() throws Exception {
            // Limpiamos la tabla.
            DatabaseOperation.DELETE.execute(conn, dataSet);
            conn.close();
            conn = null;
      }

}
```

Luego creamos el `UserDaoTest` que extenderá de `BaseDAOTestCase`. A modo de ejemplo práctico y para que no se haga muy difícil la demostración, se testeará el método `findAllUser()` de los métodos CRUD de `UserDAO`.

```
public class UserDAOTest extends BaseDAOTestCase {

      private User user = null;
      private UserDAO dao = null;

      protected void setUp() throws Exception {
            // llamamos al setUp() de la super clase.
            super.setUp();
            // Le pedimos al applicationContext.xml el dao
            // refernciado con el atributo id "userDAO"
            dao = (UserDAO) ctx.getBean("userDAO");
      }

      protected void tearDown() throws Exception {
            super.tearDown();
            dao = null;
      }

      public void testFindAllUsers() {
            List users = dao.findAllUser();
            assertTrue(users.size() == 1);
            User user = (User) users.get(0);
```

```
            assertEquals(user. userLastName(), "Rod Johnson");
        }

    }
```

5.5.2 Pruebas unitarias para la capa de servicio (módulo BesyAppService)

Para realizar las pruebas unitarias de los MANAGERs utilizaremos, como se explicó en el capitulo 4, la herramienta JUnit. A modo de ejemplo práctico, se testearán los métodos `save(User user)`, `removeUserById(long id)` y `findAllUsers()` de la clase `UserManager`.

```
    public class UserManagerTest extends TestCase {

        private ApplicationContext ctx;
        private User user;
        private UserManager mgr;

        protected void setUp() throws Exception {
            String[] paths = { "/WEB-INF/applicationContext*.xml"
                             };
            // Inyectamos la dependencia: ApplicationContext
            ctx = new ClassPathXmlApplicationContext(paths);
            mgr = (UsersManager) ctx.getBean("UserManager");
        }

        protected void tearDown() throws Exception {
            user = null;
            mgr = null;
        }

        public void testAddAndRemoveUser() throws Exception {
            // agregamos un nuevo usuario y comprobamos que el ID
            // no sea null
            user = new User();
            user.setUserName("Easter");
            user.setLastName("Bunny");
            user = mgr.save (user);
            assertNotNull(user.getId());
            // borramos el usuario agregado anteriormente y
            // verificamos que cuando lo volvamos ir a buscar,
            no // exista más.
            long userId = user.getId();
            mgr.removeUserById(userId);
            user = mgr.retriveUserById(userId);
            assertNull("User object found in database", user);
        }

        public void testfindAllUsers() throws Exception {
            // Buscamos los usuarios del sistema y verificamos que
            // no sea nulo
```

```
            List usuarios = cmsManager.findAllUsers();
            assertNotNull(usuarios);
      }


}
```

5.5.3 Pruebas unitarias para la capa de presentación (módulo BesyAppWeb)

Para realizar las pruebas unitarias de los controladores o acciones utilizaremos, como se explicó en el capitulo 4, la herramienta StrutsTestCase. A modo de ejemplo práctico, solo se realizará el test unitario para la funcionalidad de consulta de usuarios.

```
public class UserActionTest extends MockStrutsTestCase {

      public UserActionTest(String testName) {
            super(testName);
      }

      public void testExecute() {
            // seteamos al action a ejecutar. En este caso vamos a
            // ejecutar la acción que lista los usuarios llamada
            // /listUserAction.do
            setRequestPathInfo("/listUserAction");
            // Este método dispara el método execute() de la acción
            // simulando que se ha ingresado a la siguiente URL:
            // http://localhost:8080/listUserAction.do
            actionPerform();
            // Se comprueba que todo haya salido bien.
            verifyForward("success");
            // Se comprueba que no haya habido errores.
            verifyNoActionErrors();
      }
 }
```

5.6 Desarrollo de las clases funcionales.

5.6.1 Clases funcionales para la capa de persistencia (módulo BesyAppHibernate)

A continuación, implementaremos los objetos DAOs que surgen de los test unitarios. En primer lugar, se desarrollará una clase genérica, la cual podrá ser extendida por cualquier DAO. El objetivo de esta clase genérica es evitar duplicar código por cada DAO nuevo que se desarrolle. Luego se procederá a desarrollar el DAO para la entidad `User`.

```
public interface GenericDaoInterface {

        public Collection findAll() throws ObjectNotFoundException;

        public Object insert(Object obj) throws
                                        ObjectNotCreatedException;

        public void remove(Object obj) throws
                                        ObjectNotRemovedException;

        public void removeByPrimaryKey(Serializable key) throws
                                        ObjectNotRemovedException;

        public Object findByPrimaryKey(Serializable key) throws
                                          ObjectNotFoundException;

        public List find(final List restrictions) throws
                                          ObjectNotFoundException;

        public Object findUnique(final List restrictions) throws
                                          ObjectNotFoundException;

        public Object update(Object obj) throws
                                        ObjectNotUpdatedException;
}


public abstract class GenericDAO extends HibernateDaoSupport
                            implements GenericDaoInterface{

    private static Log log = LogFactory.getLog(GenericDAO.class);

    /**
     * Permite saber que clase esta utilizando este DAO generico.
     * @return Class
     */
    protected abstract Class getObjectClass();

    /**
     * Busca todos los objectos segun getObjectClass()
     * @return Collection
     */
    public Collection findAll() throws ObjectNotFoundException {

        try {
            return getHibernateTemplate().loadAll
                                        (this.getObjectClass());

        } catch (DataAccessException dae) {
            throw new ObjectNotFoundException(dae);
        } catch (Exception ex) {
            throw new UnexpectedException(ex);
        }
    }
```

```
/**
 * Inserta un objeto
 * @param obj
 */
public Object insert(Object obj) throws
                                ObjectNotCreatedException {

    try{
        return getHibernateTemplate().save(obj);
    } catch (DataAccessException dae) {
        throw new ObjectNotCreatedException(dae);
    } catch (Exception ex) {
        throw new UnexpectedException(ex);
    }
}

/**
 * remueve un objeto
 * @param obj
 */
public void remove(Object obj) throws ObjectNotRemovedException
  {
    try{
        getHibernateTemplate().delete(obj);
    } catch (DataAccessException dae) {
        throw new ObjectNotRemovedException(dae);
    } catch (Exception ex) {
        throw new UnexpectedException(ex);
    }

}

/**
 * remueve un objecto según la clave primaria
 * @param key
 */
public void removeByPrimaryKey(Serializable key) throws
                                  ObjectNotRemovedException {

    try{
        Object object = this.findByPrimaryKey(key);
        getHibernateTemplate().delete(object);
    } catch (DataAccessException dae) {
        throw new ObjectNotRemovedException(dae);
    } catch (Exception ex) {
        throw new UnexpectedException(ex);
    }
}

/**
 * busca un objeto por clave primaria
 * @param key
 * @return Object
 *
 */
public Object findByPrimaryKey(Serializable key) throws
                                  ObjectNotFoundException {
```

```
    try{
        return getHibernateTemplate().load(getObjectClass(),
                                                        key);
    } catch (DataAccessException dae) {
        throw new ObjectNotFoundException(dae);
    } catch (Exception ex) {
        throw new UnexpectedException(ex);
    }
}

/**
 * Actualiza un objeto
 * @param obj
 */
public Object update(Object obj) throws
                                ObjectNotUpdatedException {

    try{
        getHibernateTemplate().saveOrUpdate(obj);
        return obj;
    } catch (DataAccessException dae) {
        throw new ObjectNotUpdatedException(dae);
    } catch (Exception ex) {
        throw new UnexpectedException(ex);
    }

}

/**
 * Busca una lista de objetos según una lista de restricciones.
 * Para armar esta lista de restricciones se utilizará la clase
 * RestrictionsBuilderImpl.java
 * @param restrictions
 * @return List
 */
public List find(final List restrictions) throws
                                    ObjectNotFoundException {

    try{
        final DetachedCriteria criteria = DetachedCriteria.
                            forClass (this.getObjectClass());

        CollectionUtils.forAllDo(restrictions, new Closure(){
            public void execute(Object object) {
                criteria.add((Criterion) object);
            }
        });

        return getHibernateTemplate().findByCriteria(criteria);
    } catch (DataAccessException dae) {
        throw new ObjectNotFoundException(dae);
    } catch (Exception ex) {
        throw new UnexpectedException(ex);
    }
}
```

```
    /**
     * Buscar un solo objeto segun una lista de restricciones
     * Para armar esta lista de restricciones se utilizará la clase
     * RestrictionsBuilderImpl.java
     * @param restrictions (Lista de Criterion)
     * @return Object
     */
    public Object findUnique(final List restrictions) throws
                                        ObjectNotFoundException {

        List objects = this.find(restrictions);
        return (!objects.isEmpty() ? objects.get(0) : null);
    }

}
```

Podemos observar que esta clase extiende de `HibernateDaoSupport`**:** Dicha clase viene con la API de Spring y de la cual se debe extender para obtener el soporte para Hiberante. Esta clase, como bien lo explica el libro de Matt Raible [MR05] y el manual de referencia de Spring [SR07], se encarga de manejar las sesiones de hibernate para interactuar con la base de datos. Por una lado, requiere de un `SessionFactory` que es definido en el `applicationContext.xml` y por otro lado, tiene un método llamado `getHibernateTemplate()` el cual devuelve un objeto `HibernateTemplate` que tiene varios de los métodos que posee la interfase `HibernateSession`**.**

Al mismo tiempo, existen otros soportes de DAOs de los cuales se pueden extender:

- JdbcDaoSupport
- JdoDaoSupport
- JpaDaoSupport

Cabe destacar que esta clase genérica tiene un método abstracto llamado `getObjectClass()`**,** el cual debe ser implementado por los DAOs que extiendan de esta clase. Gracias a este método, esta clase genérica tiene conocimiento de cual es la clase que la esta extendiendo, y así, sabe como usar los métodos CRUD que tiene definidos.

```
public interface UserDAO {

    User saveUser(User user) throws BesyDAOException;

    List findAllUser() throws BesyDAOException;

    User retriveUserById(long id);

    User updateUser(User user);

    void removeUserById(Long id);
}

public class UserDAOHibernate extends GenericDAO implements
                                                    UserDAO {

    private static Log log = LogFactory.getLog(
                                    UserDAOHibernate.class);
    private RestrictionsBuilder builder;

    // ***************************Injection dependencies
    public void setRestrictionsBuilder(
                              RestrictionsBuilder builder) {
        this.builder = builder;
    }

    public User saveUser(User user) throws BesyDAOException {
        User userPersisted = (User) insert(user);
        log.info("Inserted..: " + userPersisted);
        return userPersisted;
    }

    public List findAllUser() throws BesyDAOException {
        return (List) findAll();
    }

    public User retriveUserById(long id) {
        User userRetrived = (User) findByPrimaryKey(id);
        log.info("Retrived by PK..: " + userRetrived);
        return userRetrived;
    }

    public User updateUser(User user) {
        User userUpdated = (User) update(user);
        log.info("Updated..: " + userUpdated);
        return userUpdated;
    }

    public void removeUserById(Long id) {
        removeByPrimaryKey(id);
        log.info("Removed User by PK: " + id);
    }

    // Implementación del método abstracto de la superclase.
    protected Class getObjectClass() {
```

```
            return User.class;
        }
    }
```

En el `ApplicationContext.xml` ha sido referenciado como un bean cuyo `id` es `userDAO` y tiene una propiedad o dependencia definida como variable de instancia llamada `SessionFactory`. Esta propiedad está definida en la clase llamada `HibernateDaoSupport` de la cual extiende, tanto como variable de instancia, como también su método setter, por lo que no es necesaria su implementación.

```
public final void setSessionFactory(org.hibernate. SessionFactory
                                                    sessionFactory)
```

5.6.2 Clases funcionales para la capa de servicio (módulo BesyAppService)

Luego, implementaremos los objetos MANAGERs que surgen de los test unitarios. En primer lugar, desarrollaremos la interfase, y luego la implementación a la cual le setearemos, vía inyección de dependencia, el DAO `UserDAO`. Aquí podemos apreciar que lo que le estamos inyectando al manager es una interfase (en tiempo de compilación) pero que luego en tiempo de ejecución se le inyectará la implementacion de esa interfase definida en el `applicationContext.xml`.

```
public interface UserManager {

    User update(User user) throws ServiceException;

    List findAllUsers() throws ServiceException;

    User retriveUserById(long id) throws ServiceException;

    User save(User newUser) throws ServiceException;

    void removeUserById(Long aLong) throws ServiceException;
}
```

```
public class UserManagerImpl implements UserManager {

    private UserDAO dao;
    private TransactionTemplate txTemplate;
```

```
// *********** IoC: método setter usado por Spring para
// *********** setear el valor de la variable de instancia dao.
public void setUserDAO (UserDAO dao) {
    this.dao= dao;
}

// *********** IoC: método setter usado por Spring para setear
// *********** el valor de la variable de instancia txTemplate.
public void setTransactionManager(PlatformTransactionManager
                                                    txManager) {
    this.txTemplate = new TransactionTemplate(txManager);
}

public User update(final User user) throws ServiceException {
        return (User) txTemplate.execute(new
                                    TransactionCallback()   {
                public Object doInTransaction(TransactionStatus
                                        transactionStatus) {
                    return dao.updateUser(user);
                }
        });
 }

public List findAllUsers() throws ServiceException {
    return (List) txTemplate.execute(new
                                    TransactionCallback() {
              public Object doInTransaction(TransactionStatus
                                        transactionStatus) {
                  return dao.findAllUser();
              }
    });
}

public User retriveUserById(final long id) throws
                                         ServiceException {
    return (User) txTemplate.execute(new
                                     TransactionCallback()  {
        public Object doInTransaction(TransactionStatus
                                        transactionStatus) {
            return dao.retriveUserById(id);
        }
    });
}

public User save(final User newUser) throws ServiceException {
    return (User) txTemplate.execute(new
                                      TransactionCallback() {
        public Object doInTransaction(TransactionStatus
                                       transactionStatus) {
            return dao.saveUser(newUser);
        }
    });
}

public void removeUserById(final Long id) throws
                                           ServiceException {
  txTemplate.execute(new TransactionCallbackWithoutResult() {
```

```
        protected void doInTransactionWithoutResult(TransactionStatus
                                                              status) {
                dao.removeUserById(id);
            }
        });
    }
}
```

Inyección de dependencia: Es aquí donde entra en juego la ID y la IoC, y es el Lightweight Container (Spring) quien se encargará de inyectar las implementaciones de las dependencias permitiendo a la clase `UserManagerImpl` depender solo de la interfase que en este caso, será `UserDAO` e indicar que dicha interfase tendrá como implementación (en tiempo de ejecución) a `UserDAOHibernate`. Por lo que cuando se ejecute el método `setUserDAO(…)`, lo que se recibirá por parámetro será la implementación de la interfase `UserDAO`, permitiendo tener varias implementaciones del DAO. Por ejemplo, sera suficiente con solo modificar el `applicationContext.xml`, indicandole que la implementación de `UserDAO` no sera más `UserDAOHibernate`, sino que a partir de ahora será `UserDAOJdbc`.

Por otro lado, a este manager también se le inyecta otra dependencia: `TransactionTemplate`, que nos ayudará a demarcar el comienzo y fin de una transacción con la base de datos. Este tipo de transacciones son las que se conocen como transacciones programáticas de Spring. Existe otra forma de crear transacciones de forma declarativa directamente en el `applicationContext.xml`, pero que no se detalla en este ejemplo práctico.

5.6.3 Clases funcionales para la capa de presentación (módulo BesyAppWeb)

Ahora pasaremos a desarrollar los controladores de Struts, también conocidos como los actions o acciones.

```
// Clase controladora que solo tiene lógica de delegar al
                    // controlador correspondiente.
public class UserAction extends Action {

    public static final String ADD_ACTION = "add";
    public static final String EDIT_ACTION = "edit";
    public static final String REMOVE_ACTION = "remove";
    public static final String ACTION = "action";
```

```
public static final String USER_ID_PARAMETER = "userId";


private UserManager userManager = null;

// ********** IoC: método setter usado por Spring para setear
// ********** el valor de la variable de instancia userManager.
public void setUserManager(UserManager userManager) {
    this.userManager = userManager;
}

public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                            HttpServletRequest req,
                             HttpServletResponse res) throws
                                        ServiceException {

    UserForm userForm = (UserForm) form;
    String userId = req.getParameter(USER_ID_PARAMETER);
    if (ADD_ACTION.equals(req.getParameter(ACTION))) {
        addUserActionPressed(userForm);
        return mapping.findForward(UtilConstantes.SUCCESS);
    } else if (EDIT_ACTION.equals(req.getParameter(ACTION))) {
        if (editUserActionPressed(userId, req, userForm)) {
            return mapping.findForward(UtilConstantes.SUCCESS);
        } else {
            return mapping.findForward(UtilConstantes.
                                                VALIDATE_ERROR);
        }
    } else if (REMOVE_ACTION.equals(req.
                                       getParameter(ACTION))) {
        if (removeUserActionPressed(userId, req, mapping)) {
            return mapping.findForward(UtilConstantes.
                                                    SUCCESS_2);
        } else {
            return mapping.findForward(UtilConstantes.
                                                 VALIDATE_ERROR);
        }
    } else {
        throw new AssertionError("Unexpected URL parameter");
    }
}

private boolean removeUserActionPressed(String userId,
                                        HttpServletRequest req,
                                        ActionMapping mapping) {
    return validateUserId(userId, req);
}

private boolean editUserActionPressed(String userId,
                                      HttpServletRequest req,
                                      UserForm userForm) throws
                                             ServiceException {
    ActionMapping mapping;
    if (validateUserId(userId, req)) {
        User userPersisted = userManager.retriveUserById(
                                             new Long(userId));
```

```
            populateUserForm(userForm, userPersisted);
            return true;
        }
        return false;
    }

    private void addUserActionPressed(UserForm userForm) {
        cleanUserForm(userForm);
    }

    private boolean validateUserId(String userId,
                                   HttpServletRequest req) {
        ActionMapping mapping;
        if (userId == null) {
            req.setAttribute("MSG", "Debe seleccionar
                                                un elemento.");
            return false;
        }
        return true;
    }

    private void populateUserForm(UserForm form, User
                                                userPersisted) {
        form.setId(userPersisted.getId());
        form.setUserName(userPersisted.getUserName());
        form.setUserLastName(userPersisted.getUserLastName());
        form.setPassword(userPersisted.getPassword());
        form.setMail(userPersisted.getMail());
        form.setAdress(userPersisted.getAdress());
        form.setPhone(userPersisted.getPhone());
        form.setEstadoRegistracion(userPersisted.
                                getEstadoRegistracion());
        form.setDateRegistration(userPersisted.
                                getDateRegistration());
        form.setSuscripcionNews(userPersisted.isSuscripcionNews());
    }

    private void cleanUserForm(UserForm form) {
        form.setId(null);
        form.setUserName(null);
        form.setUserLastName(null);
        form.setPassword(null);
        form.setMail(null);
        form.setAdress(null);
        form.setPhone(null);
        form.setEstadoRegistracion(null);
        form.setDateRegistration(new Date());
        form.setSuscripcionNews(false);
    }
}
```

```
// Clase controladora que tienen la lógica de listar a los
// usuarios del sistema.
public class ListUserAction extends Action {

    private UserManager userManager = null;
    private final static String USUARIOS = "usuarios";


    // ********** IoC: método setter usado por Spring para setear
    // ********** el valor de la variable de instancia userManager
    public void setUserManager(UserManager userManager) {
        this.userManager = userManager;
    }

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                HttpServletRequest req,
                                 HttpServletResponse res) throws
                                              ServiceException {

       Collection usuarios = userManager.findAllUsers();
        if (!usuarios.isEmpty())
            req.setAttribute(USUARIOS, usuarios);

        return mapping.findForward(UtilConstantes.SUCCESS);
    }

}

// Clase controladora que tienen la lógica de dar de alta nuevos
// usuarios como así también el proceso de actualización de los
        // mismos.
public class AddOrUpdateUserAction extends Action {

    private UserManager userManager = null;
    private static final Long OK = 1L;

    public void setUserManager(UserManager userManager) {
        this.userManager = userManager;
    }

    // ********** IoC: método setter usado por Spring para setear
    // ********** el valor de la variable de instancia userManager
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                HttpServletRequest req,
                                 HttpServletResponse res) throws
                                                ServiceException {

        UserForm userForm = (UserForm) form;
        addOrUpdateUser(userForm);
        return mapping.findForward(UtilConstantes.SUCCESS);
    }

    private void addOrUpdateUser(UserForm userForm) throws
                                                ServiceException {
        Long userId = userForm.getId();
```

```
        if (userId != null) {
            User userFromDatabase = userManager.
                                              retriveUserById(userId);
            if (userFromDatabase != null) {
                User userToUpdate = updateWebUserFromDatabase(
                                          userFromDatabase, userForm);
                userManager.update(userToUpdate);
            } else {
                throw new AssertionError("Unexpected value.
                                                 User ID: " + userId +
                                    " should be exist in database.");
            }
        } else {
            User userToPersit = createWebUserFrom(userForm);
            userManager.save(userToPersit);
        }
    }

    private User updateWebUserFromDatabase(User userFromDatabase,
                                                UserForm userForm) {

        userFromDatabase.setUserName(userForm.getUserName());
        userFromDatabase.setUserLastName(
                                          userForm.getUserLastName());
        userFromDatabase.setPassword(userForm.getPassword());
        userFromDatabase.setMail(userForm.getMail());
        userFromDatabase.setAdress(userForm.getAdress());
        userFromDatabase.setPhone(userForm.getPhone());
        userFromDatabase.setSuscripcionNews(
                                       userForm.isSuscripcionNews());
        return userFromDatabase;
    }

    private User createWebUserFrom(UserForm form) {
        return new User(form.getId(),
                form.getUserName(),
                form.getUserLastName(),
                form.getPassword(),
                form.getMail(),
                form.getAdress(),
                form.getPhone(),
                OK,
                new Date(),
                form.isSuscripcionNews());
    }
}


// Clase controladora que tiene la lógica de eliminar a los
// usuarios del sistema.
public class DeleteUserAction extends Action {

    private UserManager userManager = null;
```

```
     // ********** IoC: método setter usado por Spring para
     setear // ********** el valor de la variable de instancia
     userManager
    public void setUserManager(UserManager userManager) {
        this.userManager = userManager;
    }
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                HttpServletRequest req,
                                 HttpServletResponse res) throws
                                                 ServiceException {

        String[] userIds = req.getParameterValues(UserAction.
                                                 USER_ID_PARAMETER);
        for (String userId : userIds) {
            userManager.removeUserById(new Long(userId));
        }
        return mapping.findForward(UtilConstantes.SUCCESS);
    }
}

// Clase que automaticamente sera populado (suministrado) con datos
// desde el servidor para ser mostrados del lado del cliente.
public class UserForm extends ActionForm {

    private Long id;
    private String userName;
    private String userLastName;
    private String password;
    private String mail;
    private String adress;
    private String phone;
    private Long estadoRegistracion;
    private Date dateRegistration;
    private boolean suscripcionNews;

    public Long getId() {
        return id;
    }

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

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getUserLastName() {
        return userLastName;
    }

    public void setUserLastName(String userLastName) {
```

```
    this.userLastName = userLastName;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getMail() {
    return mail;
}

public void setMail(String mail) {
    this.mail = mail;
}

public String getAdress() {
    return adress;
}

public void setAdress(String adress) {
    this.adress = adress;
}

public String getPhone() {
    return phone;
}

public void setPhone(String phone) {
    this.phone = phone;
}

public Long getEstadoRegistracion() {
    return estadoRegistracion;
}

public void setEstadoRegistracion(Long estadoRegistracion) {
    this.estadoRegistracion = estadoRegistracion;
}

public Date getDateRegistration() {
    return dateRegistration;
}

public void setDateRegistration(Date dateRegistration) {
    this.dateRegistration = dateRegistration;
}

public boolean isSuscripcionNews() {
    return suscripcionNews;
}

public boolean getSuscripcionNews() {
    return suscripcionNews;
```

```
    }

    public void setSuscripcionNews(boolean suscripcionNews) {
        this.suscripcionNews = suscripcionNews;
    }
}
```

5.6.4 Clases funcionales para la capa de dominio (módulo BesyAppDomain)

Por último, para finalizar con esta demostración, crearemos las entidades persistentes o también conocidas como entidades de dominio. Algunas de estas clases son las que se mapearán con las tablas de la base de datos mediante Hibernate y los archivos *.hbm.xml. A continuación desarrollaremos la entidad persistente junto con el archivo de configuración para hibernate. No se entrará en detalles sobre el framework Hibernate.

```
public interface RestrictionsBuilder {
    ListBuilder clear();

    ListBuilder with(Criterion criterion);

    List build();
}

public class RestrictionsBuilderImpl implements RestrictionsBuilder
{

    private ListBuilder listBuilder;


    public RestrictionsBuilderImpl() {
        listBuilder = new ListBuilder();
    }

    public ListBuilder clear() {
        listBuilder.clear();
        return listBuilder;
    }

    public ListBuilder with(final Criterion criterion) {
        return listBuilder.with(criterion);
    }

    public List build() {
        return listBuilder.build();
    }

}
```

```
public class User {

    private Long id;
    private String userName;
    private String userLastName;
    private String password;
    private String mail;
    private String adress;
    private String phone;
    private Long estadoRegistracion;
    private Date dateRegistration;
    private String tempNumber;
    private boolean suscripcionNews;


    protected User() {
    }

    public User(Long id, String userName, String userLastName,
                String password, String mail, String adress, String
                phone, Long estadoRegistracion, Date
                dateRegistration, boolean suscripcionNews) {
        this.id = id;
        this.userName = userName;
        this.userLastName = userLastName;
        this.password = password;
        this.mail = mail;
        this.adress = adress;
        this.phone = phone;
        this.estadoRegistracion = estadoRegistracion;
        this.dateRegistration = dateRegistration;
        this.suscripcionNews = suscripcionNews;
    }

    public Long getId() {
        return id;
    }

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

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
```

```
public String getUserLastName() {
    return userLastName;
}

public void setUserLastName(String userLastName) {
    this.userLastName = userLastName;
}


public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getMail() {
    return mail;
}

public void setMail(String mail) {
    this.mail = mail;
}

public String getAdress() {
    return adress;
}

public void setAdress(String adress) {
    this.adress = adress;
}

public Long getEstadoRegistracion() {
    return estadoRegistracion;
}

public void setEstadoRegistracion(Long estadoRegistracion) {
    this.estadoRegistracion = estadoRegistracion;
}

public Date getDateRegistration() {
    return dateRegistration;
}

public void setDateRegistration(Date dateRegistration) {
    this.dateRegistration = dateRegistration;
}

public String getTempNumber() {
    return tempNumber;
}

public void setTempNumber(String tempNumber) {
    this.tempNumber = tempNumber;
}
```

```
    public boolean isSuscripcionNews() {
        return suscripcionNews;
    }

    public void setSuscripcionNews(boolean suscripcionNews) {
        this.suscripcionNews = suscripcionNews;
    }
}
```

`user.hbm.xml`: Podemos apreciar que este archivo esta en la sección 5 HIBERNATE SessionFactory del `applicationContext.xml` de Spring. Mediante este archivo logramos el mapeo objeto/relacional entre el objeto `User` y la tabla `WEB_USER`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
     PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
            "http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">

<hibernate-mapping>
    <class name="ar.besy.model.User" table="WEB_USER"
                                                lazy="false">

        <id name="id" column="id" unsaved-value="0">
            <generator class="increment"/>
        </id>
        <property name="userName" column="userName" />
        <property name="userLastName" column="userLasrName"/>
        <property name="password" column="password"/>
        <property name="mail" column="mail"/>
        <property name="adress" column="adress"/>
        <property name="phone" column="phone"/>
        <property name="estadoRegistracion"
                                    column="estadoRegistracion"/>
        <property name="dateRegistration"
                                    column="dateRegistration" />
        <property name="tempNumber" column="tempNumber" />
        <property name="suscripcionNews"
                                     column="suscripcionNews" />

    </class>
</hibernate-mapping>
```

Como corolario de esta demostración práctica, se explicará sintétiticamente el flujo de trabajo (workflow) del caso de uso - *Consulta de los usuarios del sistema*:

1. En primer lugar se recibe por URL:
   http://localhost:8080/listUserAction.do

2. La acción `/listUserAction.do` será mapeado por Spring por medio de Struts con la clase de Java correspondiente (lo que se conoce como Action o Acción). Esto lo realizará a partir del `struts-config.xml` que nos indicará como serán mapeadas las acciones a clases de Java bien definidas. Es decir, este mapeo lo realizará Spring mediante el Proxy `DelegatingActionProxy` delegando al manejador de acciones de Struts definido en el `ContextLoaderPlugIn` del `applicationContext`.xml. Es decir, cada accion (*.do que se reciba por la URL) definida en el struts-config.xml tendrá su bean (acción asociada) definido en el `action-servlet.xml`. En otras palabras, el contendor (Spring) busca la definición del bean.

3. Luego, como bien se explicó en el capitulo 3, y como así lo explica [MR05] y [WB05], mediante la IoC (Inversión del Control), el contenedor toma el control del bean `ListUserAction.java`. asociado a la acción `listUserAction.do` y lo instancia (crea una nueva referencia en memoria del objeto en cuestión).

4. Se procede a la pre-inicialización del bean a través de sus dependencias. Spring popula (llena) las propiedades del objeto especificadas en las definiciones del bean (`servlet-action.xml`) pasándolo a un estado listo. El bean estará disponible para ser utilizado en la aplicación. Como podemos apreciar en el `servlet-action.xml`, este bean tiene una sola

dependencia definida que es `UserManager.java` y la cual se encuentra definida en el archivo `applicationContext.xml`.

5. Una vez que el bean queda apto para ser utilizado por el framework, Spring por medio de Struts, llama al método `execute(…..)` para ejecutar la acción de listar los usuarios por pantalla. Para lo cual, deberá pasar por cada una de las etapas descriptas en todo este trabajo. Entonces, del action pasará a la capa de servicios, la cual a su vez llamará a la capa de persistencia para realizar la consulta pertinente y asi el resultado de la consulta retornará al action o acción el cual despachará el flujo a la vista o JSP.

Conclusiones

El estándar J2EE no ha sido calificado como exitoso en la práctica debido a los excesivos esfuerzos de los desarrolladores y su baja performance, que radica en el exceso de capas y objetos inútiles. La especificación EJB 2.0 es un claro ejemplo de ello. Sin embargo, la nueva especificación de EJB 3.0, introdujo numerosos cambios a favor respecto de la anterior.

Este trabajo destaca, cómo gracias a estas deficiencias, del estándar J2EE, surgen otras alternativas para el desarrollo de software empresarial y es así como nace, entre otros, el contenedor ligero Spring como framework de ID. En este tipo de arquitecturas, tanto los servicios como los objetos de dominio son objetos POJOs (Plain Old Java Object, Objetos planos de Java) y éstos, en lugar de correr sobre un contenedor EJB, lo hacen sobre un Lightweight Container o contenedor ligero.

Estos dos entornos de trabajo – Spring y el estándar EJB 3 (J2EE) -, comparten un mismo objetivo: ambos tienen como propósito prestar servicios de middleware o de capa de servicio para los POJOs. En otras palabras, ambos resuelven la inyección de dependencias en tiempo de ejecución, evitando el uso de Service Locators, permitiendo a los POJOs no preocuparse por como obtener las dependencias. Por otra parte, tienen bajo acoplamiento con el framework, logrando que los desarrolladores puedan concentrarse más en la lógica de negocio. Esta es una de las principales cuestiones en las que varios autores, entre otros, Rod Johnson, Martin Fowler y Eric Evans hacen constantemente hincapié: Un buen framework no debe ser intrusivo y el dominio debe estar aislado, desacoplando los objetos de dominio del resto del sistema.

Por otro lado, se llega a la conclusión que la ID es la inversa de JNDI, es decir, en lugar de que los EJB o POJOs hagan un lookup (Dependency Lookup) de un Datasource, el Datasource será inyectado (Dependency Injection) al EJB o POJO. Simplificando la

complejidad de búsqueda de recursos vía JNDI y facilitando el desarrollo de aplicaciones empresariales.

Spring brinda varias alternativas de inyección de dependencias, como ser, inyección por constructor (quizás sea la más recomendable aunque no sea la preferida por Rod Johnson) e inyección por metodos setter. Mientras que el estándar EJB 3.0 no soporta la inyección de POJOs definidos por el usuario en otros POJOs.

Si bien, la utilización de POJOs conlleva una serie de ventajas, cabe destacar que por si solos no tienen ningún sentido, ya que toda aplicación empresarial (y la experiencia así lo demuestra) requiere de una infraestructura lógica que resuelva temas como el manejo de transacciones, seguridad y persistencia entre otros.

Por otro lado, y como corolario de este trabajo, los contenedores ligeros alientan el uso de las interfases, siendo un recurso muy poderoso en la POO. La inyección de dependencia es un claro ejemplo de ello: En tiempo de compilación, la ID se llevará a cabo por medio de interfases, mientras que en tiempo de ejecución el contenedor inyectará la implementación correspondiente permitiendo reducir el acoplamiento entre las capas y más aún, se podrán inyectar objetos falsos o mock objects en lugar de la verdadera implementación y así realizar test unitarios sin la necesidad de tocar nada del código funcional.

Anexo

A.1 Estudio comparativo entre Wikipedia y la Enciclopedia Británica

egún un estudio comparativo realizado por la famosa revista científica Nature [INT38] sobre la exactitud de Wikipedia, respecto de la enciclopedia británica, Wikipedia sale favorecida.

ikipedia es una enciclopedia online (en línea) libre que cualquier persona pueda corregir, siendo un recurso cada vez más usado. Pero es también polémico ya que si cualquier persona puede corregir artículos, ¿cómo los usuarios saben si Wikipedia es tan exacta como otras fuentes establecidas como ser la enciclopedia Británica?.

Una investigación conducida por la revista Nature permite revelar que en ambas enciclopedias existen numerosos errores. El ejercicio revelo que sobre 42 entradas testeadas en ambas enciclopedias, la diferencia no fue tan grande: En Wikipedia se encontraron alrededor de 4 inexactitudes, mientras que en la enciclopedia Británica alrededor de 3.

Por otro lado, Wikipedia esta creciendo rápidamente. La enciclopedia ha agregado 3.7 millones de artículos en 200 idiomas desde que fue fundada en 2001, lo que la ha convertido, según Alexa [INT39], un servicio que nos muestra los sitios más visitados, en el décimo sitio web más visitado y en continuo crecimiento.

Michael Twidale, documentalista en la universidad de Illinois, dice que el juego más fuerte de Wikipedia es la velocidad a la cual puede actualizarse, un factor no considerado por los revisores de la revista Nature.

Glosario

AOP (Aspect Oriented Programming, Programación Orientada a Aspectos): Es un nuevo paradigma cuyo objetivo es por un lado separar las funcionalidades comunes utilizadas a lo largo del sistema (cross-cutting) también llamada aspecto. Un ejemplo claro es el loggin (registro de sucesos del sistema) que va a estar en todos los módulos de la aplicación. Y por otro lado, separar las funcionalidades propias de cada módulo [INT94].

API (Application Programming Interfase, Interfaz de Programación de aplicación): Conjunto de métodos que ofrece cierta librería para ser utilizado por otro software como una capa de abstracción. En otras palabras, es una interfase de comunicación entre componentes de software [INT97].

Aplicación Standalone: Es una aplicación que no corre bajo un servidor de aplicaciones o servlet container.

AXIS: Es una implementación de Apache del protocolo SOAP (Simple Object Access Protocol) [INT69]

Balanceo de carga: Es un concepto informático que se refiere a la tecnica de compartir o distribuir el trabajo entre varios procesos, ordenadores, discos u otros recursos [INT93].

Business Delegate ó Delegar a la capa de negocio: De “Core J2EE patterns”: Para poder reducir el acoplamiento entra la capa de presentación y los servicios de negocio y así evitar el acceso directo a los componentes de negocio (ya que cambios en estos se verán reflejados en la capa de presentación), se usa este patrón. Permite ocultar todos los detalles de la implementación de los servicios de negocio agregando una clase intermedia llamada Business Delegate. [INT10] [INT11].

Cliente fino o delgado (Thin Client): Es básicamente lo opuesto a los clientes pesados. Depende del servidor para llevar a cabo el procesamiento de la funcionalidad [INT108] [INT109].
Cliente rico o pesado (rich client o fat client): En el mundo del software y en las arquitecturas clientes-servidor, típicamente corresponde a clientes con gran funcionalidad independiente del servidor [INT108] [INT109].

Clustering o Cluster: Conjunto de ordenadores que se comportan como si fuese un único ordenador. Estos ordenadores están unidos en red de alta velocidad de tal forma que el conjunto es visto como un único ordenador. Básicamente de un cluster se espera alto rendimiento, alta disponibilidad, equilibrio de carga o balanceo de carga y escalabilidad [INT92].

CORBA (Common Object Request Broker Architecture — arquitectura común de intermediarios en peticiones a objetos). Estándar que establece una plataforma de desarrollo de para sistemas distribuidos facilitando la invocación de métodos remotos. Fue definido por la OMG. Permite la interoperabilidad entre aplicaciones de diferentes tecnologías [INT87].

DAO (Data Access Object, Objeto de Acceso a los Datos): Es conocido como un patrón de diseño J2EE. Esto nos permite abstraernos y encapsular el acceso a los datos. Los DAOs manejan las conexiones con la fuente de datos para obtener y actualizar datos [INT106].

Dispatcher (Despachador): Es un patrón J2EE que actúa como un punto de entrada entre las peticiones de la vista (request que llegan mediante la capa de presentación) y la lógica de negocio. Básicamente es como el patrón de diseño MVC con el controlador actuando como un Front Controller.

Display PostScript: (DPS) Sistema que permite visualizar imágenes en la pantalla. DPS utiliza PostScript como lenguaje para el modelado y generación de los gráficos en la pantalla.

EJB (Enterprise Java Beans, Java Beans Empresariales): Son una de las API que forman parte del estándar de construcción de aplicaciones empresariales J2EE de Sun. Estos son

objetos del lado del servidor provistos por el servidor de aplicaciones. Al ser objetos que nos brinda el servidor de aplicaciones, ayudan al desarrollador a abstraerse de los problemas generales de una aplicación empresarial (seguridad, concurrencia, transaccionabilidad, persistencia) [INT70].

EJB de Sesión o Session EJBs: Generalmente sirven a la capa cliente como una fachada de los servicios proporcionados por otros componentes disponibles en el servidor.
Existen dos tipos:

- Con estado (stateful): Estos beans poseen un estado y el acceso a los mismos se limita a un solo cliente
- Sin estados (stateless) Estos beans no poseen estado, permitiendo que sean accedidos concurrentemente. En otras palabras, no garantizan que el contenido de las variables de instancia se conserve entre llamadas a los métodos [INT70].

EJB de entidad o Entity EJBs: Estos tipos de beans poseen la característica fundamental de la persistencia [INT70].

EJBs dirigidos por mensajes o Message-driven EJBs: Son beans con funcionamiento asincrónico, es decir, se suscriben a un tema (topic) o cola (queue) de mensajes y se activan al recibir un mensaje dirigido a dicho tema o cola [INT70].

EJB Container: Como su nombre lo indica un contenedor EJB proporciona un entorno de ejecución para los EJB [INT71].

Fortran: Es un lenguaje de programación desarrollado en los años 50. Se utiliza principalmente en aplicaciones científicas y análisis numérico. Desde 1958 ha pasado por varias versiones, entre las que se destacan FORTRAN II, FORTRAN IV, FORTRAN 77, Fortran 90, Fortran 95 y Fortran 2003 [INT104].

Front Controller (Controlador Frontal): Es un patrón J2EE que acepta todos los requerimientos de un cliente, los autentica y redirecciona la petición a los manejadores (managers) apropiados. Básicamente, el Front Controller podría dividir esta funcionalidad en dos objetos bien definidos. El Front Controller propiamente dicho, que se encargaría de

recibir las peticiones y autenticarlas y el Dispatcher, que seria el redireccionador hacia los manejadores o managers de las capas siguientes.

GUI (Graphical User Interfase, Interfaz Grafica de Usuario): Es el artefacto tecnológico que posibilita mediante el uso y la representación del lenguaje visual, una interacción amigable con el sistema informático [INT90].

Hibernate: Herramienta de O/R mapping. Para la plataforma Java. También existe una version para la plataforma .Net llamado NHibernate. Es una implementación de la técnica de O/R mapping [INT99].

HTTP (HyperText Transfer Protocol, Protocolo de Transferencia de Hipertexto): Es el protocolo de transferencia usado en cada transacción de la web. Es un protocolo sin estado, es decir, no guarda información sobre conexiones anteriores [INT95].

J2EE: Actualmente conocido como Java EE o como se llamaba antes (hasta la version 1.4 de Java) J2EE (Java 2 Enterprise Edition, Java 2 Edición Empresarial). Es una plataforma de programación para desarrollar y ejecutar software en lenguaje Java. J2EE o Java EE incluye varias especificaciones tales como JDBC, RMI, JMS, XML, EJB, Servlets, portlets y define como coordinarlos [INT75].

JAR (Java ARchive): Es un tipo de archivo que contiene muchas clases Java agrupadas en un solo archivo [INT100].

JavaServer Pages o JSP: Es una tecnología Java que permite generar código web dinámico. Básicamente permiten la utilización de código Java mediante scripts [INT74].

JDBC (Java Database Connectivity, Conectividad Java a Base de datos): Es un API de Java que permite la ejecución de operaciones sobre la base de datos independientemente del sistema operativo y utilizando el dialecto SQL del modelo de base de datos que se necesite [INT107].

JOTM: Implementación open source del estándar JTA para el manejo de transacciones.

JSF (Java Server Faces): Es un framework para aplicaciones web que simplifica el desarrollo de interfases de usuarios en el mundo de J2EE o Java EE. Básicamente es un conjunto de APIs para representar componentes de interfase de usuario, administrar su estado, manejar eventos, validar entradas, definir esquemas de navegación. Utiliza la tecnología JSP como soporte para desplegar las paginas [INT82].

JTA: Establece una serie de Interfases Java entre el manejador de transacciones y las partes involucradas en el sistema de transacciones distribuidas. Es conocida como el API de transacciones de Java [INT63].

JVM (Java Virtual Machine, Maquina Virtual de Java): Es la encargada de ejecutar el código compilado usando el lenguaje Java. Es un sistema que se sitúa entre el sistema operativo y la aplicación actuando como traductor del pseudo-código de máquina Java a código de máquina y permitiendo a Java ser multiplataforma [INT76].

Mock objects (Objetos Mocks u Objetos de Maqueta): Es una técnica de programación en la que se reemplazan los objetos de negocio por implementaciones “dummy” (simuladas) que emulan el código real.

MVC (Model-View-Controller, Modelo-Vista-Controlador): Patrón de arquitectura de software que separa los datos, la interfase de usuario y la lógica de control en tres componentes bien distintos [INT77] [INT78].

NeXT Software, Inc. Compañía de computación en California que desarrolló y manufacturó dos estaciones de trabajo durante su existencia, NeXTcube y NeXTstation
Fue fundado después en 1985 por Steve Jobs después de su dimisión de Apple [INT32].

O/R mapping (Object-Relational mapping, mapeo objeto-relacional - o sus siglas O/RM, ORM, y O/R mapping): Consiste en una técnica de programación que permite convertir datos entre el paradigma orientado a objetos y el relacional (utilizado en una base de datos). En otras palabras, esta técnica permite el mapeo entre objetos y tablas de una manera transparente [INT98].

Overhead: Es generalmente considerado como un exceso en los tiempos de procesamiento, memoria, ancho de banda u otro recurso utilizado para alcanzar una meta [INT91].

PARC centro de investigación de Palo Alto de Xerox, es una compañía de investigación y de desarrollo en Palo Alto, California que comenzó como división de Xerox Corporation (Corporacion).

POJO: Acrónimo de Plain Old Java Object (Objeto Viejo Plano de Java). Sigla creada por Martin Fowler, Rebecca Parsons y Josh MacKenzie y utilizada por desarrolladores para enfatizar el uso de clases simples y que no dependen de un framework en especial [INT68].

Refactoring: “Es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo” [INT64].

RMI (Remote Method Invocation, Invocación de Métodos Remota). Mecanismo ofrecido por Java para invocar remotamente a un método. La invocación consiste en la serialización de los objetos en la red. Ideal para aplicaciones distribuidas. Ahora bien, si se quiere interactuar con otras tecnologías se deberá usar SOAP o CORBA [INT88].

RMI / IIOP (Remote Method Invocation / Internet Inter-Orb Protocol, convertido en GIOP - por General -, Invocación de Metodos Remota / Protocolo Entre ORBs General): Se lee RMI sobre IIOP. Denota la interfase RMI de Java sobre el sistema CORBA [INT86] [INT113].

Ruby on Rails: Es un framework para el desarrollo de aplicaciones web siguiendo el patron MVC. Utiliza Ajax para la vista [INT110].

Serialización / Des-serializacion: También conocido como marshalling en ingles. Consiste en el proceso de codificación de un objeto en un medio de almacenamiento con el fin de transmitirlo a través de una conexión de red como una serie de bytes o en xml. Es un proceso para hacer persistente un objeto. La des-serializacion es el proceso inverso [INT96].

Servicie Locator ó localizador de servicio: Del conjunto de "Core J2EE patterns, patrones J22E base"): Como los clientes J2EE interactúan con componentes de servicios como los EJB que proveen servicios de negocio y persistencia, estos clientes deberán localizar a dichos componentes por medio de una operación de lookup (búsqueda) usando JNDI (Java Naming and Directory interfase. Interfase Java de Nombres y Directorios). Centraliza los lookups de los servicios distribuidos, puntos de control y crea un cache eliminando los lookup redundantes. [INT08] [INT09].

Servlet Container (Contenedor de Servlets): Es el ambiente donde se ejecutan los servlets. Básicamente es usado para administrar los servlets y JSP [INT72].

Servlet: Son objetos Java que corren dentro de un servlet container. También pueden ejecutarse en un servidor de aplicaciones. Son programas que se ejecutan del lado del servidor [INT73].

Smalltalk: Fue el primer lenguaje de programación que se corresponde con el paradigma orientado a objetos. Todo en smalltalk es un objeto y todo el desarrollo es mediante mensajes que son enviados entre los objetos [INT103].

SOA (Service Oriented Architecture, Arquitectura Orientada a Servicios): Permite que una aplicación no sea desarrollada desde cero sino una integración con servicios ya desarrollados y publicados [INT89].

SOAP (Simple Object Access Protocol, Protocolo Simple de Acceso a los Datos): Es un protocolo basado puramente en XML que permite intercambiar información en ambientes descentralizados y distribuidos [INT69].

Solaris: Sistema operativo desarrollado por Sun Microsystems. Certificado como una version de UNIX.

Spring MVC: Módulo del framework de Spring que implementa el patrón MVC para la construcción de aplicaciones web. Es un framework de capa de presentación al igual que Struts y JSF [INT85].

Spring: Framework de código abierto para el desarrollo de aplicaciones Java. Existe una version para la plataforma .NET llamado Spring.net. En los últimos años se ha convertido en un sustituto del modelo EJB. Una de las características más notables de este framework es la inyección de dependencias permitiendo al desarrollador concentrarse en la lógica de negocio [INT83] [INT84].

Stakeholders: Es un término que se utiliza para referirse a quienes pueden afectar o son afectados por las actividades de una empresa. Estos grupos o individuos son considerados como un elemento esencial en la planeación estratégica [INT102].

Struts: Es un entorno de trabajo o framework open source para el desarrollo de aplicaciones web. Struts era parte del proyecto Jakarta de Apache pero actualmente es un proyecto independiente. Struts se basa en el patrón de diseño MVC el cual se utiliza ampliamente [INT79].

Sun Microsystems: Empresa informática de Silicom Valley. Tiene entre sus productos más destacados, la plataforma de programación JAVA.

Tansfer Objects (Objetos de Transferencia) ó DTOs: Es un patrón de diseño J2EE que resuelve el problema de la transferencia de información sobre las capas. Básicamente la capa cliente necesita información para actualizar su estado, por lo que usa objetos DTOs para obtener esos datos y visualizarlos [INT66].

Test de regresión: Es un tipo de pruebas de software que trata de encontrar fallas (bugs) en la funcionalidad del software. Métodos comunes de pruebas de regresión incluyen volver a ejecutar casos de pruebas y comprobar si fallas que surgieron previamente vuelven a suceder. [INT58].

Tomcat: Es un contenedor de servlets desarrollado bajo el proyecto Jakarta. Implementa las especificaciones de los servlets y de JavaServer Pages (JSP) [INT72].

UML (Unified Modeling Language, Lenguaje Unificado de Modelado): Es un lenguaje grafico de modelado para diseño de sistemas de información orientado a objetos [INT105].

Value Objects (Objetos con valor) o VOs: Al igual que los DTOs, los VOs aplican a un patrón de diseño J2EE en donde resuelven el problema de la transferencia de información sobre las capas de la misma manera que los DTOs. Estos objetos son creados por la capa de servicios (EJB) encapsulando la lógica de negocio y pasándolos a la capa cliente [INT67].

WAR (Web Application aRchive): Es un tipo de archivo que contiene no solo clases Java, sino que también contiene archivos XML, JSPs y otros recursos relacionados con las aplicaciones web. [INT101].

WebWork: Es un entorno de trabajo o framework como Struts, también open source ideal para el desarrollo de aplicaciones web basado en el patrón MVC. Permite al usuario poder concentrarse más en la lógica de negocio que en cuestiones típicas del mundo web. Tanto WebWork como Struts 2.0 en la actualidad hacen uso de XWork un framework que provee entre otras cosas la inyección de dependencias [INT80] [INT81].

Bibliografía

Internet:

[INT01] http://en.wikipedia.org/wiki/Software_architecture

[INT02] http://www.dcs.st-and.ac.uk/research/architecture

[INT03] http://www.awprofessional.com/bookstore/product.asp?isbn=0321112296&rl=1

[INT04] http://ciberaula.com/curso/lamp/que_es/

[INT05] http://www.onlamp.com/pub/a/onlamp/2001/01/25/lamp.html

[INT06] http://www.martinfowler.com/bliki/LayeringPrinciples.html

[INT07] http://www.stevenblack.com/PTN-Layers.asp

[INT08] http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html

[INT09] http://java.sun.com/blueprints/patterns/ServiceLocator.html

[INT10] http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html

[INT11] http://java.sun.com/blueprints/patterns/BusinessDelegate.html

[INT12] http://www.martinfowler.com/articles/injection.html

[INT13] http://en.wikipedia.org/wiki/Dependency_injection

[INT14] http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art51.asp

[INT15] http://www.w3.org/TR/ws-gloss/

[INT16] http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_3317.asp

[INT17] http://agilemanifesto.org/principles.html

[INT18] http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=grasp

[INT19] http://www.martinfowler.com/bliki/AnemicDomainModel.html

[INT20] http://www-128.ibm.com/developerworks/rational/library/3100.html

[INT21] http://www.omg.org/mda/

[INT22] http://en.wikipedia.org/wiki/Service-oriented_architecture

[INT23] http://www.extremeprogramming.org/rules/unittests.html

[INT24] http://www.agiledata.org/essays/tdd.html
[INT25] http://www.subbu.org/articles/jts/JTS.html
[INT26] http://java.sun.com/products/jta/
[INT27] http://www.adobe.com/cfusion/knowledgebase/index.cfm?id=tn_17974
[INT28] http://www.martinfowler.com/eaaCatalog/modelViewController.html
[INT29] http://en.wikipedia.org/wiki/Connection_pool
[INT30] http://en.wikipedia.org/wiki/SmallTalk
[INT31] http://lowendmac.com/orchard/06/1220.html
[INT32] http://en.wikipedia.org/wiki/NeXT
[INT33] http://en.wikipedia.org/wiki/NEXTSTEP
[INT34] http://rpbouman.blogspot.com/search/label/Database%20Stored%20Procedures
[INT35] http://www.unixlibre.org/articulos.jsp?cve=35
[INT36] http://lowendmac.com/orchard/05/0705.html
[INT37] http://www.oracle.com/technology/products/ias/toplink/doc/10131/main/_html/
uowund001.htm#sthref5638
[INT38] http://www.nature.com/news/2005/051212/full/438900a.html
[INT39] http://www.alexa.com/
[INT40] http://es.wikipedia.org/wiki/Crisis_del_Software
[INT41] http://java.sys-con.com/read/180347.htm
[INT42] http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm
[INT43] http://en.wikipedia.org/wiki/Unified_Modeling_Language
[INT44] http://www.awprofessional.com/articles/article.asp?p=30432&seqNum=4&rl=1
[INT45] http://en.wikipedia.org/wiki/Object-oriented_software_engineering
[INT46] http://www.oracle.com/technology/tech/java/oc4j/ejb3/fov_ejb3.html
[INT47] http://jcp.org/en/jsr/detail?id=220
[INT48] http://www.onjava.com/pub/a/onjava/2005/06/29/spring-ejb3.html
[INT49] http://labs.jboss.com/jbossejb3/
[INT50] http://blog.interfase21.com/main/2006/11/08/
spring-framework-the-origins-of-a-project-and-a-name/
[INT51] http://www.springframework.org/docs/api/org/springframework/
context/ApplicationContext.html
[INT52] http://www.bloginformatico.com/historia-de-la-computacion-ii-parte.php
[INT53] http://en.wikipedia.org/wiki/Jackson_Structured_Programming
[INT54] http://es.wikipedia.org/wiki/NEXTSTEP

[INT55] http://www.maestrosdelweb.com/editorial/web2/
[INT56] http://martinfowler.com/articles/mocksArentStubs.html
[INT57] http://www.dbunit.org/
[INT58] http://en.wikipedia.org/wiki/Regression_testing
[INT59] http://en.wikipedia.org/wiki/Code_coverage
[INT60] http://es.wikipedia.org/wiki/JUnit
[INT61] http://www.easymock.org/
[INT62] http://java.sun.com/features/2002/11/appserver_influences.html#features
[INT63] http://java.sun.com/products/jta/
[INT64] http://www.onjava.com/pub/a/onjava/2003/07/30/jotm_transactions.html
[INT65] http://es.wikipedia.org/wiki/Refactorizaci%C3%B3n
[INT66] http://www.corej2eepatterns.com/Patterns2ndEd/TransferObject.htm
[INT67] http://java.sun.com/j2ee/patterns/ValueObject.html
[INT68] http://es.wikipedia.org/wiki/Plain_Old_Java_Object
[INT69] http://ws.apache.org/axis/index.html
[INT70] http://es.wikipedia.org/wiki/Enterprise_JavaBeans
[INT71] http://publib.boulder.ibm.com/infocenter/wsdoc400/v6r0/index.jsp?topic=/com.ibm.websphere.iseries.doc/info/ae/ae/cejb_ecnt.html
[INT72] http://tomcat.apache.org/
[INT73] http://es.wikipedia.org/wiki/Java_Servlet
[INT74] http://es.wikipedia.org/wiki/JSP
[INT75] http://es.wikipedia.org/wiki/Java_EE
[INT76] http://es.wikipedia.org/wiki/M%C3%A1quina_Virtual_de_Java
[INT77] http://www.cica.es/formacion/JavaTut/Apendice/arq_mvc.html
[INT78] http://es.wikipedia.org/wiki/Modelo_Vista_Controlador
[INT79] http://es.wikipedia.org/wiki/Apache_Struts
[INT80] http://www.opensymphony.com/webwork/wikidocs/WebWork%20Overview.html
[INT81] http://www.opensymphony.com/xwork/
[INT82] http://es.wikipedia.org/wiki/JavaServer_Faces
[INT83] http://es.wikipedia.org/wiki/Spring_Framework
[INT84] http://www.springframework.org/
[INT85] http://www-128.ibm.com/developerworks/web/library/wa-spring3/
[INT86] http://es.wikipedia.org/wiki/RMI-IIOP
[INT87] http://es.wikipedia.org/wiki/CORBA

[INT88] http://es.wikipedia.org/wiki/RMI
[INT89] http://es.wikipedia.org/wiki/SOA_vs._web_2.0
[INT90] http://es.wikipedia.org/wiki/Interfaz_gr%C3%A1fica_de_usuario
[INT91] http://en.wikipedia.org/wiki/Computational_overhead
[INT92] http://es.wikipedia.org/wiki/Cluster_de_computadores
[INT93] http://es.wikipedia.org/wiki/Balance_de_carga
[INT94] http://es.wikipedia.org/wiki/Programaci%C3%B3n_Orientada_a_Aspectos
[INT95] http://es.wikipedia.org/wiki/HTTP
[INT96] http://es.wikipedia.org/wiki/Serializaci%C3%B3n
[INT97] http://es.wikipedia.org/wiki/Application_Programming_Interfase
[INT98] http://es.wikipedia.org/wiki/Mapeo_objeto-relacional
[INT99] http://es.wikipedia.org/wiki/Hibernate
[INT100] http://en.wikipedia.org/wiki/JAR_(file_format)
[INT101] http://en.wikipedia.org/wiki/WAR_%28file_format%29
[INT102] http://en.wikipedia.org/wiki/Stakeholder
[INT103] http://swiki.agro.uba.ar/small_land/65
[INT104] http://es.wikipedia.org/wiki/Fortran
[INT105] http://es.wikipedia.org/wiki/Lenguaje_Unificado_de_Modelado
[INT106] http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
[INT107] http://es.wikipedia.org/wiki/JDBC
[INT108] http://en.wikipedia.org/wiki/Fat_client
[INT109] http://en.wikipedia.org/wiki/Thin_client
[INT110] http://www.rubyonrails.org/
[INT111] http://technet.microsoft.com/es-es/library/ms188929.aspx
[INT112] http://static.springframework.org/spring/docs/1.2.x/reference/beans.html
[INT113] http://es.wikipedia.org/wiki/IIOP
[INT114] http://www.myjavaserver.com/~aldosolis/tutorial_struts/struts_tutorial.html
[INT115] http://es.wikipedia.org/wiki/Apache_Struts

Libros:

[BCK03] Software Architecture in Practice, Second Edition - Len Bass, Paul Clements,

Rick Kazman – 2003 - (http://books.google.es/books?vid=ISBN0321154959&id=mdiIu8Kk1WMC&dq=Software+Architecture+in+Practice,+Second+Edition)

[RJ02] Wrox - Expert One-On-One - j2Ee Design And Development - Rod Johnson - 2002 – (http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764543857.html)

[RJ04] John.Wiley.and.Sons.Expert.One.on.one.J2EE.Development.Without.EJB - Rod Johnson -2004 – (http://www.amazon.com/Expert-One-on-One-J2EE-Development-without/dp/0764558315)

[RJ05] Wrox - Professional Java Development with the Spring Framework - Rod Johnson 2005 – (http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764574833.html)

[MF02] Patterns of Enterprise Application Architecture - Martin Fowler – 2002.- (http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Fowler/dp/0321127420)

[MF97] Addison-Wesley Professional - Analysis Patterns, Reusable Object Models – Martin Fowler - 1997 -(http://books.google.com.ar/books?vid=ISBN0201895420&id=4V8pZmpwmBYC&pg=PP6&lpg=PP6&ots=x5z_erJHu6&dq=analysis+patterns+martin+fowler&sig=to3raXSQqVKspg1_IRAGhWLqjtU)

[EE03] Addison-Wesley - Domain-Driven Design - Tackling Complexity in the Heart of Software. - Eric Evans - 2003 - (http://www.awprofessional.com/bookstore/product.asp?isbn=0321125215&rl=1)

[MR05] Spring Live - Matt Raible - 2005 – (http://www.amazon.com/Spring-Live-Matt- Raible/dp/0974884375)

[RRR03] Wiley - Developing Java Web Services - Architecting and Developing Secure Web Services Using Java -Ramesh Nagappan, Robert Skoczylas, Rima Patel Sriganesh - 2003 -. (http://www.amazon.com/Developing-Java-Web-Services-Architecting/dp/0471236403)

[SJM02] Addison Wesley - Design Pattern java work book - Steven John Metsker – 2002- http://www.amazon.com/Design-Patterns-Workbook-Steven-Metsker/dp/0201743973

[GHJV95] Design Patterns Elements of Reusable Object Oriented Software - Gamma, Helm, Johnson y Vlissides - 1995. (http://www.amazon.com/Design-Patterns-Object- Oriented-Addison-Wesley-Professional/dp/0201633612)

[CL03] UML y patrones 2da edición - Craig Larman – 2003-

(http://www.amazon.ca/UML-y-Patrones-Larman-Craig/dp/8420534382/sr=1-11/qid=1162385601/ref=sr_1_11/701-1058756-8598755?ie=UTF8&s=books)

[MLC01] Anaya Multimedia / Wrox - Fundamentos de base de datos con java, Kevin Mukhard, Todd Lauinger y John Carnel (http://www.barataria.com/Merchant2/merchant.mvc?Screen=PROD&Store_Code=SBLRS&Product_Code=84-415-1362-7&Category_Code=AMULTI)

[BMRSS96] Wiley - Pattern Oriented Software Architecture, A System of Patterns - Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal - 1996 – (http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471958697.html)

[SA03] Wiley - The Art of Software Architecture: Design Methods and Techniques - Stephen T. Albin – 2003 – (http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471228869.html)

[PK04] Software architecture design patterns in java - Partha Kuchana - 2004 - (http://books.google.es/books?vid=ISBN0849321425&id=FqfKFl4Bm7UC&dq=Software+architecture+design+patterns+in+java+-+Partha+Kuchana)

[SR07] Spring reference - Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Rick Evans – 2004/2007 - (http://static.springframework.org/spring/docs/2.0.x/reference/index.html)

[SL06] Expert Spring MVC and Web Flow - Seth Ladd, Darren Davison, Steven Devijver, Colin Yates – 2006

[WB05] Spring in Action - Craig Walls, Walls Ryan Breidenbach - 2005 (http://www.amazon.com/Spring-Action-Craig-Walls/dp/1932394354)

[CR06] Pojos in Action – Developing Enterprise Applications with Lightweight Frameworks – Chris Richardson – (http://www.amazon.com/Spring-Action-Craig-Walls/dp/1932394354)

[BSA06] Beginning Pojos – From novices to Professional - Brian Sam-Bodden – 2006 (http://www.preciomania.com/search_getprod.php/masterid=/isbn=9781590595961)

[HM05] Pro Spring - Rob Harrop, Jan Machacek – 2005 (http://www.amazon.com/Pro-Spring-Rob-Harrop/dp/1590594614)

[BG] La nueva interacción en Internet, Web 2.0 - Bertol Belloso Garitano