Arquitectura del Módulo de Administración y Control de Recursos del Sistema de Gestión de Emergencia y Seguridad Ciudadana 171

Trabajo de Diploma presentado en opción al título de Ingeniero en Ciencias Informáticas

Autor: Rammel Maestre Sánchez

Tutores: Ing. Carlos Molina Villalobos
Ing. Michel Arias Arias

Ciudad de La Habana, junio de 2009

DECLARACIÓN DE AUTORÍA

Declaro ser autor de la presente tesis y se le reconoce a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma con carácter exclusivo.

Para que así conste firmo la presente a los ___ días del mes de _____ del año _____.

_____

Rammel Maestre Sánchez

Autor

_____ _____

Ing. Carlos Molina Villalobos

Tutor

Ing. Michel Arias Arias

Tutor

Dedicatoria

A mis padres por ser mi inspiración y guía.

Agradecimientos

A mami y papi por darme todo el apoyo del mundo.

A Any por estar tanto tiempo a mi lado.

A los tutores.

A todos los coleguitas que tiraron líneas de código a mi lado y que me impusieron metas más altas que me han hecho crecer.

RESUMEN

El Sistema de Gestión de Emergencia y Seguridad Ciudadana 171 (SIGESC 171) es un sistema informático para los Centros de Gestión de Emergencias 171 de la República Bolivariana de Venezuela que automatiza el proceso de atención a emergencias en los mismos. A su vez la administración y el control de los recursos que los Órganos de Seguridad Ciudadana ponen a disposición de estos centros es un factor decisivo en la solución satisfactoria de emergencias ciudadanas.

En el presente trabajo se especifica la arquitectura sobre la cual se desarrolló el Módulo de Administración y Control de Recursos del SIGESC 171, definiéndose los criterios de diseño, la organización lógica y física, los mecanismos de comunicación, seguridad y tratamiento de errores. Además brinda un estilo de desarrollo y un flujo de trabajo para guiar al equipo de desarrollo durante el ciclo de vida del proyecto.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La República Bolivariana de Venezuela está inmersa en un proceso de profundos cambios en todos los órdenes de la sociedad. Dentro de las prioridades, la modernización de las instituciones del sector público ocupa un lugar privilegiado.

El Ministerio del Poder Popular para las Relaciones Interiores y Justicia (MPPRIJ) promueve la formulación y puesta en marcha del Sistema de Gestión de Emergencia y Seguridad Ciudadana 171 (SIGESC 171) y su implantación en los Centros de Gestión de Emergencias 171.

Estos centros implementan y ejecutan políticas y estrategias orientadas a educar a la población y salvaguardar la vida humana, los bienes, los servicios, el medio ambiente y los recursos naturales, minimizando el impacto social y ambiental; pero los mismos no cuentan con herramientas tecnológicas que posibiliten la solución de los acontecimientos que se presentan, en un período de tiempo breve y de forma efectiva.

Todo este proceso es una actividad compleja y requiere de esfuerzos por el volumen de datos que es necesario manejar, por el empleo de tecnologías novedosas, por la interacción con otros sistemas, por el manejo de información en tiempo real, por lo trabajoso de los procesos de negocio, por la prestación de servicios de forma continua, por la capacidad de incrementarse, por la seguridad de la información entre otras.

La administración y el control de los recursos[1] que los Órganos de Seguridad Ciudadana ponen a disposición de los Centros de Gestión de Emergencias es un factor decisivo, que está estrechamente relacionado a la solución satisfactoria de los acontecimientos que se puedan presentar. El mismo implica, principalmente, mantener un registro de la disponibilidad técnica, así como, del estado de los recursos que se ponen a disposición de estas entidades, mantener un control constante de los recursos, de manera que en el instante de hacer frente a una situación se conozcan los que están disponibles para solucionar el problema.

[1] A los efectos de la presente investigación entiéndase como recurso todos aquellos elementos que intervienen en la solución de una situación de emergencia. Principalmente se incluyen aquellos recursos materiales de apoyo, dígase desde un carro patrullero y una ambulancia hasta un edificio que pueda servir de punto de referencia o como centro de evacuación.

Uno de los mecanismos más generalizados adoptado por la industria de software para enfrentar la complejidad implícita en los sistemas informáticos es la arquitectura base, aportando elementos que contribuyan a la toma de decisiones y, al mismo tiempo, aportar conceptos y lenguaje común que haga compatible y fluida la comunicación entre los equipos que participen en el proyecto.

La plataforma J2EE (Java 2 Platform, Enterprise Edition) es una de las alternativas más atrayentes para el desarrollo de aplicaciones empresariales debido a que ya es un producto maduro y con una amplia variedad de especificaciones, herramientas y técnicas.

El desarrollo de aplicaciones con J2EE nunca ha sido fácil o rápido, lo que motiva a los miembros de la comunidad internacional de Java a enfocarse en proveer un conjunto de poderosas tecnologías y herramientas con el principal objetivo de reducir el tiempo y la complejidad del proceso de desarrollo y mejorar el comportamiento de las aplicaciones.

Cuando se va a desarrollar una aplicación web sobre J2EE se puede escoger una arquitectura escalable conformada por un extenso número de soluciones libres y de código abierto. Este es un hecho que aporta beneficios extras muy importantes a los arquitectos de software, ya que obtienen un amplio espectro de alternativas. Pero precisamente esto, el amplio margen de elección de posibles soluciones, puede conllevar a una mala selección en dependencia de las necesidades reales.

Es muy grande el número de soluciones libres reutilizables orientadas a la web existentes para crear arquitecturas empresariales dentro de la plataforma J2EE, sin embargo el conocimiento y la experiencia acerca de las posibles tecnologías y combinaciones de frameworks capaces de brindar una arquitectura sólida para desarrollar software es pobre y no existe un estudio detallado sobre el tema

Dada esta **SITUACIÓN PROBLÉMICA** se impone el siguiente **PROBLEMA CIENTÍFICO:** ¿Qué especificaciones requiere la arquitectura del Módulo de Administración y Control de Recursos del Sistema de Gestión de Emergencias de Seguridad Ciudadana (171) de manera que favorezca su desarrollo?

Se define como **OBJETO DE ESTUDIO** la arquitectura de software y el **CAMPO DE ACCIÓN** la arquitectura de software para aplicaciones web sobre J2EE.

Como **OBJETIVO GENERAL** de esta investigación se establece: Definir e implementar la arquitectura del Módulo de Administración y Control de Recursos del Sistema de Gestión de Emergencias de Seguridad Ciudadana (171).

A partir de este se derivan los **OBJETIVOS ESPECÍFICOS** siguientes:

- Determinar las tecnologías y herramientas de desarrollo.
- Definir la estructura lógica y física, elementos de comunicación y los mecanismos de seguridad y tratamiento de errores.
- Determinar un estilo de desarrollo y un flujo de trabajo para la construcción del sistema.

Para satisfacer los anteriores objetivos se plantean las siguientes **TAREAS DE INVESTIGACIÓN:**

- Analizar el estado del arte de las tecnologías y herramientas de desarrollo.
- Identificar estilos y patrones arquitectónicos aplicables en el sistema.
- Valorar el conjunto de frameworks a utilizar que den solidez a la arquitectura.
- Definir la infraestructura base de desarrollo.

1 *Arquitectura de software*

- Características generales
- Niveles de abstracción arquitectónicos
- Estilos y patrones arquitectónicos
- Patrones de diseño

CAPÍTULO 1.- ARQUITECTURA DE SOFTWARE

Con el desarrollo de la era digital y el creciente aumento de la información los sistemas fueron ganando en complejidad e inevitablemente se hizo necesario el desarrollo de metodologías y técnicas para satisfacer las necesidades de las nuevas aplicaciones. Los aspectos arquitectónicos del desarrollo de software están recibiendo un interés cada vez mayor, tanto desde la comunidad científica como desde la propia industria del software.

La arquitectura de software remonta sus antecedentes hasta la década de 1960 pero su historia no ha sido tan continua como la del campo más amplio en el que se inscribe, la Ingeniería de Software. (Pfleeger, 2002) (Platt, 2002)

Hacia 1968, Edsger Dijkstra, de la Universidad Tecnológica de Eindhoven en Holanda y Premio Turing 1972, propuso que *"se establezca una estructuración correcta de los sistemas de software antes de lanzarse a programar, escribiendo código de cualquier manera"* (Dijkstra, 1983). De sus ensayos arranca la tradición de hacer referencia a *"niveles de abstracción"* que ha sido tan común en la arquitectura subsiguiente.

Después de las tempranas inspiraciones del legendario Edsger Dijkstra, y posteriormente de David Parnas y de Fred Brooks, la arquitectura de software quedó en estado de vida latente durante unos cuantos años, hasta comenzar su expansión explosiva en la década de 1990 con los manifiestos de Dewayne Perry de AT&T Bell Laboratories de New Jersey y Alexander Wolf de la Universidad de Colorado. (E. Perry, et al., 1992)

Estos autores profetizaron que la década de 1990 habría de ser la de la arquitectura de software, y conceptualizan esa arquitectura en contraste con la idea de diseño. Puede decirse que Perry y Wolf fundaron la disciplina, y su llamamiento fue respondido en primera instancia por los miembros de lo que podría llamarse la escuela estructuralista de Carnegie Mellon[2]: David Garlan, Mary Shaw, Paul Clements, Robert Allen.

Existen diversas definiciones de arquitectura de software que han evolucionado conjuntamente con el desarrollo de los nuevos sistemas:

[2] Se refiere a la Universidad Carnegie Mellon (en inglés: Carnegie Mellon University, CMU) que se ubica en la ciudad de Pittsburgh (Pensilvania) y es uno de los más destacados centros de investigación superior de los Estados Unidos en el área de informática y robótica.

"… es el estudio de la estructura a gran escala y el rendimiento de los sistemas de software. La arquitectura del sistema incluye la división de funciones entre los módulos del sistema, los medios de comunicación entre los módulos, y la representación de la información compartida." (Lane, 1990)

"… un conjunto de arquitectura (o, si se quiere, de diseño), son elementos que tienen una forma particular. Distinguiendo entre elementos de proceso, elementos de datos, y elementos de conexión." (Perry, et al., 1992)

"… la especificación de un sistema abstracto que consiste principalmente en los componentes funcionales descritos en términos de sus comportamientos, las interfaces y interconexiones entre los componentes." (Hayes-Roth, 1994)

"... la arquitectura de software se encarga del diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer las principales necesidades de funcionalidad y requisitos de rendimiento como la escalabilidad y disponibilidad. Arquitectura de software se refiere a la abstracción, con descomposición y composición, con estilo y estética." (Kruchten, 1995)

"La estructura de los componentes de un programa / sistema, sus interrelaciones y los principios y directrices que rigen su diseño y evolución en el tiempo." (Garlan, et al., 1995)

"El conjunto de decisiones significativas acerca de la organización de un sistema de software; la selección de los elementos estructurales a partir de los cuales se compondrá el sistema y sus interfaces, junto con la descripción del comportamiento de dichas interfaces en las colaboraciones que se producen entre los elementos del sistema; la composición de esos elementos estructurales y de comportamiento para formar subsistemas de tamaño cada vez mayor; y el estilo o patrón arquitectónico que guía esta organización: los elementos y sus interfaces, las colaboraciones y su composición." (Jacobson, et al., 1999)

"… es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución." (ANSI/IEEE Std 1471-2000)

> *"La arquitectura de software de un programa o sistema informático es la estructura o estructuras del sistema, que incluyen elementos de software, las propiedades visibles externamente de esos componentes, y las relaciones entre ellos."* (Bass, et al., 2003)

Los integrantes de una y otra corriente comparten una visión composicional del software, centrada en la noción de componente como unidad elemental, y no sólo para las etapas iniciales del desarrollo, es decir, durante la especificación y el diseño, sino también durante la implementación (que se convierte en ensamblado de componentes reutilizables dentro de algún marco arquitectónico), y evolución del sistema (que consiste entonces en la sustitución de componentes y la reconfiguración de la arquitectura del sistema). Esta taxonomía en general persiste en la mayoría de otras definiciones y enfoques.

Por otro lado, estas aproximaciones se basan en el concepto de interfaz como medio para describir los componentes, haciendo explícita la funcionalidad que ofrecen y sus dependencias de contexto, y ocultando los detalles internos de su implementación. La composición de componentes para formar sistemas mayores se realiza por medio de la conexión de estas interfaces.

1.1 Características generales

La arquitectura de software se encuadra dentro de la Ingeniería de Software, y en particular del diseño de software, enfrentándola ante los algoritmos y las estructuras de datos, fijando su papel dentro del proceso de diseño. De este modo, el objeto de la arquitectura de software no es el diseño de algoritmos o estructuras de datos, sino la organización a alto nivel de los sistemas de software, incluyendo aspectos como la descripción y análisis de propiedades relativas a su estructura y control global, los protocolos de comunicación y sincronización utilizados, la distribución física del sistema y sus componentes, etc. Junto a estos, también se presta especial atención a otros aspectos de carácter más general, relacionados con el desarrollo del sistema y su evolución y adaptación al cambio, como son los aspectos de composición, reconfiguración, reutilización, escalabilidad, mantenibilidad, etc. (Canal Velasco, 2000)

Una arquitectura de software es ante todo una abstracción de un sistema que suprime los detalles de los elementos que no afectan al modo en que se utilizan, en que son utilizados, en que se relacionan, o que interactúan con otros elementos. En casi todos los sistemas modernos, los elementos interactúan entre sí por medio de interfaces que dividen los detalles

de un elemento en partes públicas y privadas. La arquitectura se refiere a la parte pública de esta división; los datos privados (los que tienen que ver únicamente con la implementación interna) no son de la arquitectura. (Bass, et al., 2003)

Otro elemento significativo se refiere a la naturaleza de los elementos objeto de estudio y, fundamentalmente, a las relaciones que se establecen entre ellos, y enfrenta las interacciones entre estos componentes con las relaciones de definición. De esta forma, las relaciones de definición/uso modularizan el sistema en función de su código fuente, haciendo explícitas las relaciones de importación y exportación que indican dónde se define y dónde se usa ese código fuente. Sin embargo, desde el punto de vista de la arquitectura de software, el sistema se divide en una serie de componentes, que no tienen que coincidir con los módulos de compilación. Estos componentes realizan computaciones y almacenamiento de datos, y llevan a cabo diversas interacciones unos con otros durante la ejecución del sistema. (Canal Velasco, 2000)

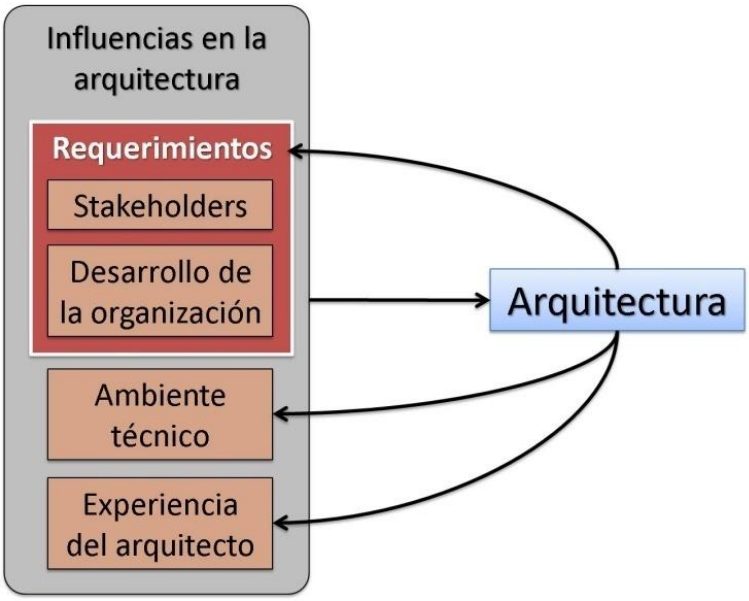
El comportamiento de cada elemento es parte de la arquitectura y es la medida en que dicho comportamiento puede ser observado o comprendido desde el punto de vista de otro elemento. Tal comportamiento es lo que permite a los elementos interactuar entre sí. No significa que el comportamiento y el rendimiento exacto de cada elemento debe ser documentado en todas las circunstancias, sin embargo, en la medida en que un elemento influye en el comportamiento de otro, debe ser escrito para interactuar entre si. (Bass, et al., 2003)

Del análisis de los distintos enfoques se destaca la relación que se establece entre los métodos arquitectónicos y los métodos de desarrollo de software (como los orientados a objetos o los estructurados). Mientras el objetivo de estos últimos es proporcionar un camino entre el espacio del problema y el de la solución, la arquitectura de software se centra únicamente en el que podríamos denominar como espacio de los diseños arquitectónicos, preocupándose de cómo guiar las decisiones a tomar en este espacio —decisiones que se basan en las propiedades de los diferentes diseños arquitectónicos y su capacidad para resolver determinados problemas—. No obstante, ambos métodos están estrechamente relacionados y se complementan: detrás de la mayoría de los métodos de desarrollo hay un estilo arquitectónico preferido, mientras que el desarrollo y uso de nuevos estilos arquitectónicos lleva normalmente a la creación de nuevos métodos de desarrollo que exploten sus características. (Canal Velasco, 2000)

La arquitectura de software establece los fundamentos para que analistas y diseñadores de sistemas, programadores y todo el capital humano necesario para el desarrollo de un software

trabajen en una línea común que permita alcanzar los objetivos y necesidades del sistema informático en cuestión. Por lo tanto la arquitectura de software está influenciada por varios elementos que a su vez reciben la influencia de la arquitectura obtenida. (Ver Figura 1)

Figura 1 Influencias en la arquitectura de software



Estas influencias simultáneas se pueden resumir en la Tabla 1. (Bass, et al., 2003)

Tabla 1 Resumen de las influencias sobre la arquitectura de software

Elementos	**Influencias**	
	Elementos sobre la arquitectura	**Arquitectura sobre los elementos**
Stakeholders	Los stakeholders tienen diferentes intereses que desean que el sistema garantice u optimice. Entre ellos el rendimiento, la fiabilidad, la disponibilidad, la compatibilidad entre plataformas, la utilización de memoria, el uso de la red, la seguridad, usabilidad, y la interoperabilidad con otros sistemas. Estas propiedades determinan el diseño global de la arquitectura.	La arquitectura puede afectar a los requisitos del cliente para el próximo sistema, dándole la oportunidad de recibir un sistema (basado en la misma arquitectura) de una forma más confiable, oportuna y económica que el que se construyó desde cero. Puede estar dispuesto a relajar algunos de los requisitos para obtener estas economías.

Estructura y metas de la organización que la desarrolla	La arquitectura es influenciada por la estructura o naturaleza de la organización que la desarrolla. Las mismas se pueden dividir en tres tipos: *Negocio inmediato:* la organización puede estar interesada en arquitecturas existentes y en productos basados en ellas. En consecuencia puede exigir que el sistema propuesto sea una secuencia de sistemas similares y las estimaciones de costo asumen un alto grado de reutilización de activos. *Negocio a largo plazo:* puede invertir en una infraestructura para perseguir objetivos estratégicos de un futuro producto y pueden ver el sistema propuesto como uno de los medios de financiación y de la ampliación de la infraestructura. *Estructura organizacional:* la organización de la empresa puede determinar o condicionar la división de la funcionalidad de la arquitectura que permite el aislamiento de las especialidades.	Una arquitectura establece una estructura para un sistema, prescribe las unidades de software que deben llevarse formarse. Estas unidades son la base para la estructura del proyecto de desarrollo. Los equipos están formados por unidades individuales de software y el desarrollo, prueba e integración de todas las actividades giran en torno a estas unidades. Asimismo con los calendarios, asignación de los recursos y los presupuestos. La construcción eficaz de un sistema puede permitir a una empresa establecer un pie en un área de mercado. La arquitectura puede proporcionar oportunidades para la producción eficiente y el despliegue de sistemas similares, y la organización podrá ajustar sus objetivos aprovechando su nueva experiencia para sondear el mercado.
Ambiente técnico	El entorno actual cuando se diseña una arquitectura tiene influencias directas sobre la misma. Se podrían incluir prácticas estándares de la industria o técnicas de la ingeniería de software predominantes en la comunidad profesional del arquitecto.	Algunos sistemas influyen y, de hecho, cambian la cultura de la ingeniería de software. Muchos de ellos llegan a marcar pautas y estándares tras los cuales los sistemas subsiguientes se ven afectados por su legado.
Experiencia del arquitecto	Si los arquitectos de un sistema han tenido buenos resultados con un enfoque arquitectónico es probable que lo proyecten en los nuevos proyectos. Por el contrario, si su experiencia previa con este enfoque fue desastrosa, los arquitectos pueden ser reacios a intentarlo de nuevo. Las opciones arquitectónicas también puede provenir de la educación y la formación de un arquitecto, la exposición a patrones de arquitectura exitosos, o la exposición a los sistemas que han funcionado especialmente bien o especialmente mal.	El proceso de construcción del sistema contribuirá con la experiencia del arquitecto con sistemas posteriores, añadiendo la base de la experiencia empresarial. Los patrones y estilos arquitectónicos aplicados es muy probable que se vuelvan a utilizar, por ser del dominio del equipo de trabajo y del arquitecto. Por otra parte, las arquitecturas que fallen tienen menos probabilidades de ser elegidas para los proyectos futuros.

1.1.1 Importancia de la arquitectura de software

Comunicación entre los stakeholders

Una arquitectura de software representa una abstracción común de un sistema que todos o casi todos los stakeholders pueden utilizar como base para el entendimiento mutuo, la negociación, el consenso y la comunicación. Cada uno de los stakeholders se preocupa por diferentes características del sistema que se ven afectadas por la arquitectura. Por ejemplo, el usuario le preocupa que el sistema sea fiable y esté disponible cuando sea necesario, al cliente le preocupa que la arquitectura se puede implementar en el plazo y con el presupuesto previsto, el líder se preocupa (así como sobre los costos y el calendario) de que la arquitectura permita a los equipos de desarrollo trabajar independiente el uno del otro, interactuando en forma disciplinada y controlada. El arquitecto se muestra preocupado por las estrategias a seguir para alcanzar todos esos objetivos. La arquitectura proporciona un lenguaje común para expresar los intereses de cada stakeholders. Sin ese lenguaje, es difícil entender suficientemente los grandes sistemas para hacer que las primeras decisiones puedan influir en la calidad y utilidad del producto. (Bass, et al., 2003)

Tempranas decisiones de diseño

La arquitectura de software se manifiesta tempranamente en las decisiones de diseño de un sistema, y estos primeros enlaces tienen un peso fundamental sobre el desarrollo restante del sistema, su funcionamiento en el despliegue y su mantenimiento. Es también el primer momento en el que las decisiones de diseño que rigen el sistema a construir pueden ser analizadas. (Bass, et al., 2003)

- Define las limitaciones de implementación: Una implementación de una arquitectura es conformada por las decisiones de diseño estructural descritas por la arquitectura en cuestión. Esto significa que la aplicación debe ser dividida en elementos prescritos, los elementos deben interactuar entre sí de la forma determinada, y cada elemento debe cumplir con su responsabilidad con los otros como establece la arquitectura definida.
- Dicta la estructura organizativa: La arquitectura no sólo prescribe la estructura del sistema que se está desarrollando, sino que esta estructura se convierte en un grabado de la estructura del proyecto de desarrollo (y veces, en la estructura de toda la organización). Debido a que la arquitectura incluye el más alto nivel de descomposición del sistema, normalmente es utilizado como base para el desglose estructural del trabajo, que a su vez determina las unidades de planificación, programación y

presupuesto; canales de comunicación entre los equipos de trabajo, control de configuración y organización del sistema de archivos, la integración y planes de pruebas.

- Permite o inhibe los atributos de calidad de un sistema: Las estrategias para obtener atributos de calidad de alto nivel (rendimiento, seguridad, usabilidad, escalabilidad, etc.) son sumamente arquitectónicas. Es importante comprender, sin embargo, que la arquitectura por sí sola no puede garantizar la funcionalidad o la calidad. Un pobre diseño o decisiones de implementación no apropiados siempre pueden socavar un adecuado diseño arquitectónico.
- Permite precisar costos y estimar cronogramas: Las estimaciones de costos y el cronograma son una importante herramienta de gestión para adquirir los recursos necesarios y entender si un proyecto está en problemas. Estimar costos sobre la base de una comprensión de las piezas del sistema es, intrínsecamente, más preciso que los basados en el conocimiento general del sistema. La definición inicial de una arquitectura significa que los requisitos para un sistema se han revisado y, en cierto sentido, validado.

Abstracción transferible de un sistema: un modelo reusable

- La arquitectura de software constituye una parte relativamente pequeña, un modelo de cómo un sistema se estructura y la forma en que sus elementos colaboran, y este modelo es transferible a través de los sistemas. En particular, se puede aplicar a otros sistemas similares y requisitos funcionales, y pueden promover en gran escala la reutilización. Cuanto antes en el ciclo de vida de un software la reutilización se aplique, mayor será el beneficio que puede lograrse. Mientras que la reutilización de código es beneficiosa, la reutilización en el nivel de arquitectura provee una tremenda influencia de los sistemas con requisitos similares. (Bass, et al., 2003)
- Las líneas de productos comparten una arquitectura común: En una línea de productos de software la arquitectura es diseñada para manejar las necesidades de toda la familia y prevé atender a todos los miembros de la línea tomando decisiones de diseño que se aplican en toda la familia tempranamente. La arquitectura define lo que es fijo para todos los miembros de la línea de productos y lo que es variable.
- Utilización de elementos desarrollados externamente: El desarrollo de una arquitectura a menudo se centra en la composición o montaje de elementos que es probable que se hayan desarrollado por separado, incluso de forma individual, los unos de los otros.

Esta composición es posible porque la arquitectura define los elementos que pueden incorporarse en el sistema y establece los protocolos de comunicación entre ellos.

- Puede ser la base para la formación profesional: La arquitectura, incluyendo una descripción de cómo interactúan los elementos para llevar a cabo una determinada funcionalidad, puede servir como introducción, al sistema, de nuevos miembros al proyecto. Esto refuerza el punto de que una de las aplicaciones importantes de la arquitectura de software es apoyar y fomentar la comunicación entre los distintos stakeholders.

1.1.2 Recomendaciones para el diseño de una arquitectura de software

Bass, Clements y Kazman (Bass, et al., 2003) dividen sus observaciones para diseñar una arquitectura de software en dos grupos: recomendaciones de proceso y recomendaciones de producto (también denominadas recomendaciones estructurales). Algunas de ellas se muestran a continuación.

Recomendaciones de proceso:

- La arquitectura debe ser el producto de un solo arquitecto o de un pequeño grupo de arquitectos con un líder identificado.
- El arquitecto (o equipo de arquitectura) debe tener los requisitos funcionales del sistema así como la lista de prioridades de los atributos de calidad (como la seguridad o extensibilidad) que la arquitectura debe satisfacer.
- La arquitectura debe estar bien documentada, con al menos una vista estática (describe qué componentes la conforman) y una dinámica (describe cómo se comportan los componentes a lo largo del tiempo y como interactúan entre sí), utilizando una notación que todos los stakeholders pueden entender con un mínimo de esfuerzo.
- La arquitectura debe ser distribuida entre los stakeholders del sistema, que deben participar activamente en su revisión.
- La arquitectura debe ser analizada en cuanto a las medidas cuantitativas aplicables (como máximo rendimiento) y evaluados formalmente los atributos de calidad antes de que sea demasiado tarde para hacer cambios a la misma.

Recomendaciones estructurales:

- La arquitectura debe caracterizarse por tener módulos bien definidos cuyas responsabilidades funcionales son asignadas por los principios de ocultamiento de la

información y la separación de incumbencias, aislando la mayor parte del software de los cambios en la infraestructura.

- Cada módulo debería tener una interfaz bien definida que encapsule el cambio de diferentes aspectos (tales como las estrategias de implementación y estructuras de datos) a otro software que utiliza sus servicios. Estas interfaces deben permitir que sus respectivos equipos de desarrollo trabajen con independencia uno de otro.
- Los atributos de calidad se lograrían conociendo con exactitud las tácticas arquitectónicas específicas para cada atributo.
- La arquitectura nunca debe depender de una versión particular de un producto o herramienta. Si depende de un determinado producto comercial, debe estar estructurada de tal manera que el cambio a otro producto sea sencillo y barato.
- Los módulos que producen datos (o que acceden a los datos) deben ser separados de los módulos que consumen datos (capas de servicios). Esto tiende a disminuir el impacto de algún cambio porque los mismos se limitan a menudo, a la obtención o al consumo de datos.
- La arquitectura debe contener un pequeño número de sencillos patrones de interacción. Es decir, el sistema debe hacer las mismas cosas de la misma manera en todos los casos. Esto ayuda en la claridad, reduce el tiempo de desarrollo, aumenta la fiabilidad y mejora la modularidad. También muestra la integridad conceptual de la arquitectura que conlleva a un buen desarrollo.

1.2 Niveles de abstracción arquitectónicos

El estudio de los aspectos arquitectónicos del desarrollo de software está estructurado en niveles jerárquicos, partiendo del más general al más particular. Cada nivel tiene una profundidad bien definida lo que permite un análisis más detallado del mismo. De este modo podemos distinguir varios niveles. (Canal Velasco, 2000)

- Estilos arquitectónicos: define una familia de sistemas en términos de un patrón de organización estructural. Más concretamente, un estilo arquitectónico determina el vocabulario de los componentes y conectores que pueden ser utilizados en instancias del propio estilo, junto con un conjunto de restricciones sobre la forma en que pueden combinarse. (Garlan, et al., 1994). En el epígrafe siguiente se abordan con más detalles los elementos que caracterizan este nivel de abstracción.

- Modelo de referencia: es una división de funcionalidad y del flujo de datos entre las piezas. Es la descomposición estándar de un problema conocido en partes que cooperen para resolver el problema. Derivadas de la experiencia, los modelos de referencia son una característica de dominios maduros. (Bass, et al., 2003)
- Arquitectura de referencia: es un modelo de referencia asignada a los elementos de software (que en cooperación implementan la funcionalidad definida en el modelo de referencia) y al flujo de datos entre ellos. Considerando que un modelo de referencia divide la funcionalidad, una arquitectura de referencia es la correspondencia de esa funcionalidad con la descomposición del sistema. Esta correspondencia puede ser, pero no necesariamente, uno a uno. Un elemento de software puede implementar parte de una función o de varias funciones. (Bass, et al., 2003)
- Marcos de trabajo: definen una arquitectura adaptada a las particularidades de un determinado dominio de aplicación, definiendo de forma abstracta una serie de componentes y sus interfaces, y estableciendo las reglas y mecanismos de interacción entre ellos. Normalmente se incluye además la implementación de algunos de los componentes o incluso varias implementaciones alternativas. La labor del usuario del marco sería, por un lado, seleccionar, instanciar, extender y reutilizar los componentes que este proporciona, y por otro, completar la arquitectura del mismo desarrollando componentes específicos, que deben ser encajados en el marco, logrando así desarrollar diferentes aplicaciones siguiendo las restricciones estructurales impuestas por el marco de trabajo. (Canal Velasco, 2000)
- Familias o líneas de productos: son un conjunto de aplicaciones similares, o bien diferentes versiones o configuraciones de la misma aplicación, de forma que todas estas aplicaciones tienen la misma arquitectura, pero cada una de ellas está adaptada o particularizada al entorno o configuración donde se va a ejecutar. Además de compartir una arquitectura, en los productos de la línea se reutilizan toda una serie de componentes que son, precisamente, los que caracterizan la línea de productos. Las ventajas de establecer estas líneas de productos tienen que ver fundamentalmente con la reutilización, ya que implican una disminución de costos y un aumento de la calidad y fiabilidad del producto. (Canal Velasco, 2000)
- Instancias arquitectónicas: representan la arquitectura de un sistema de software concreto, caracterizada por los módulos o componentes que lo forman y las relaciones que se establecen entre ellos. Como es lógico, cualquier aplicación tiene una

> arquitectura que puede ser descrita y estudiada y que presenta determinadas propiedades, ya sea o no la aplicación parte de una familia o línea de productos, haya sido o no desarrollada utilizando un marco de trabajo, siga o no un modelo o arquitectura de referencia, o incluso pueda ser catalogada dentro de un estilo arquitectónico o siga un estilo heterogéneo. (Canal Velasco, 2000)

1.3 Estilos y patrones arquitectónicos

Desde los inicios de la arquitectura de software, se observó en la práctica del diseño y la implementación que ciertas regularidades de configuración aparecían una y otra vez como respuesta a similares demandas. A estos elementos que siguen el mismo patrón estructural se les denomina estilo arquitectónico.

Mary Shaw y Paul Clements (Shaw, et al., 1996) dieron algunas de las primeras definiciones formales de estilos arquitectónicos determinando que es un conjunto de reglas de diseño que identifican los tipos de componentes y conectores que pueden utilizarse para componer un sistema o subsistema, junto con las restricciones locales o globales de la manera en que esta composición se realiza.

Los componentes, incluyendo los subsistemas encapsulados, se pueden distinguir por la naturaleza de su computación: por ejemplo, si retienen estado entre una invocación y otra, y de ser así, si ese estado es público para otros componentes. Los tipos de componentes también se pueden distinguir conforme a su forma de empaquetado, o dicho de otro modo, de acuerdo con las formas en que interactúan con otros componentes. El empaquetado es usualmente implícito, lo que tiende a ocultar importantes propiedades de los componentes. Para clarificar las abstracciones se aísla la definición de esas interacciones bajo la forma de conectores: por ejemplo, los procesos interactúan por medio de protocolos de transferencia de mensajes, o por flujo de datos a través de tuberías (pipes). Es en gran medida la interacción entre los componentes, mediados por conectores, lo que confiere a los distintos estilos sus características distintivas. (Shaw, et al., 1996)

Otra caracterización es ofrecida por Robert Monroe, Drew Kompanek, Ralph Melton y David Garlan (Monroe, et al., 1997), también de Carnegie Mellon, que definen el estilo como una entidad consistente en cuatro elementos:

1. *Vocabulario de elementos de diseño:* componentes y tipos de conectores tales como tubos, filtros, clientes, servidores, analizadores, bases de datos y otros.

2. *Reglas de diseño o restricciones:* permiten determinar la composición de esos elementos. Por ejemplo, las normas pueden prohibir los ciclos en un estilo de tuberías y filtros, especificar que un sistema cliente-servidor la relación debe ser de n a una, o definir un patrón de composición específico como un patrón de descomposición por tuberías en un compilador.
3. *Interpretación semántica:* proporciona significados bien definidos en las composiciones de los elementos de diseño, debidamente limitada por las normas de diseño.
4. *Análisis de los sistemas construidos con bases en el estilo seleccionado.* Por ejemplo análisis de disponibilidad para estilos basados en procesamiento en tiempo real, o detección de abrazos mortales para modelos cliente-servidor.

En los últimos años, casi nadie propone nuevas definiciones de estilo, sino que comienzan a repetirse las ideas tratadas por autores anteriores. Roy Thomas Fielding (Fielding, 2000) sintetiza la definición de estilo diciendo que un estilo arquitectónico es un conjunto coordinado de restricciones arquitectónicas que restringe los roles/rasgos de los elementos arquitectónicos y las relaciones permitidas entre esos elementos dentro de la arquitectura que se conforma a ese estilo.

De esta forma, es evidente que un estilo arquitectónico define una familia de sistemas en términos de un patrón de organización estructural. En particular, de acuerdo a los autores, un estilo arquitectónico define tanto un vocabulario de tipos de componentes y conectores –como en el caso de filtros y tubos- como un conjunto de restricciones sobre cómo combinar esos componentes y conectores. Los componentes y conectores que lo forman representan unidades computacionales y de datos así como y los mecanismos de interacción entre ellos. Mientras tanto las invariantes del estilo se representan mediante restricciones de interconexión. Asociados a cada estilo hay una serie de propiedades que lo caracterizan, determinando sus ventajas e inconvenientes, condicionando la elección de uno u otro estilo.

Los estilos arquitectónicos pueden ser clasificados según la principal área en que se enfocan. La Tabla 2 muestra estas aéreas y los estilos correspondientes. (Meier, et al., 2008)

Tabla 2 Clasificación de los estilos arquitectónicos

Categoría	Estilo arquitectónico
Despliegue	Cliente-Servidor, N capas, 3 Capas
Estructura	Basado en Componentes, Orientado a Objetos, Arquitectura en Capa
Dominio	Modelo de Dominio, Gateway (Pasarela)
Comunicación	Arquitectura Orientada a Servicios (SOA), Bus de Mensaje, Tuberías y Filtros

Esta es una de las clasificaciones de las muchas existentes. Diferentes autores las agrupan en otros criterios que no son los anteriores, pero independientemente de la clasificación o el nombre que reciban, todos conservan las principales características que lo identifican.

Los estilos arquitectónicos no son excluyentes y a menudo se superponen al elegir los que se adapten a las necesidades del sistema a desarrollar. Por ejemplo, se puede utilizar un diseño orientado a objetos organizados en una arquitectura en capas, con un modelo de dominio para el acceso a los datos, y la comunicación mediante una arquitectura orientada a servicios.

El uso de estilos arquitectónicos, tiene una serie de importantes beneficios (Monroe, et al., 1997):

- *Promueve la reutilización del diseño:* las soluciones con propiedades bien entendidas se pueden volver a aplicar con confianza a nuevos problemas.
- *Posibilita la reutilización de código:* a menudo los aspectos invariantes de un estilo arquitectónico comparten las mismas implementaciones. Por ejemplo, los sistemas que describen un estilo de tuberías y filtros podrían reutilizar primitivas del sistema operativo Unix para manejar la programación de tareas, sincronización y comunicación a través de tuberías. Del mismo modo, un cliente-servidor pueden aprovechar el estilo de los actuales mecanismos RPC[3] y la capacidad de generación.
- *Facilita la comprensión*: es más fácil de entender la organización de un sistema si se utilizan las estructuras preestablecidas por el estilo o los estilos empleados. Por ejemplo, la caracterización de un sistema como un "cliente-servidor" inmediatamente transmite una fuerte imagen de los tipos de elementos y la forma en que estos encajan.

Por otra parte, los estilos arquitectónicos difieren de los patrones en varios aspectos importantes: (Buschmann, et al., 1996)

- Los estilos arquitectónicos sólo describen la estructura del conjunto de frameworks para las aplicaciones. Los patrones de arquitectura de software, sin embargo, abarcan otros elementos, comenzando con la definición de la estructura básica de una aplicación

[3] RPC (del inglés Remote Procedure Call, Llamada a Procedimiento Remoto) es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos.

(patrones de arquitectura) y terminando con los modelos que describen cómo aplicar un diseño de un determinado tema en el lenguaje de programación.

- Los estilos arquitectónicos son independientes uno de otro, pero un patrón depende de los pequeños patrones que contiene, de los patrones con los que interactúa, así como de los grandes patrones en el que figura.
- Los patrones son más orientados hacia los problemas que los estilos arquitectónicos. Los estilos arquitectónicos expresan técnicas de diseño desde un punto de vista que es independiente de una situación real de diseño. Un patrón expresa un muy específico problema recurrente de diseño y presenta una solución al mismo, todo ello desde el punto de vista del contexto en que surge el problema.

Un patrón arquitectónico es una descripción de los elementos y tipos de relación, junto con una serie de limitaciones sobre la forma en que pueden utilizarse. Un patrón puede ser pensado como un conjunto de limitaciones a una arquitectura —tipos de elementos y patrones de interacción— de estas limitaciones se definen un conjunto o una familia de arquitecturas que se ajusten a ellas. Por ejemplo, cliente-servidor es un modelo arquitectónico común. El cliente y el servidor son dos tipos de elementos, y su coordinación se describe en términos del protocolo que el servidor utiliza para comunicarse con cada uno de sus clientes. La utilización del término cliente-servidor implica sólo que existen varios clientes, los clientes no se identifican a sí mismos, y no hay discusión en su funcionalidad, que no sea la implementación de los protocolos entre ellos. Un patrón de arquitectura no es una arquitectura, pero transmite una imagen útil del sistema, que impone limitaciones en la arquitectura y a su vez, en el sistema. (Bass, et al., 2003)

Una definición formal es la siguiente:

> *"Un patrón de arquitectura de software describe un particular problema de diseño recurrente que se plantea en contextos específicos de diseño, y presenta un esquema genérico probado. El esquema de solución se especifica mediante la descripción de los elementos que lo constituyen, sus responsabilidades y relaciones, y las formas en las que colaboran."* (Buschmann, et al., 1996)

Expresan el esquema fundamental de organización para sistemas de software. Proporcionan un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye las normas y directrices para la organización de las relaciones entre ellos. Son las plantillas de las

arquitecturas de software concretas. Especifican las propiedades estructurales de todo el sistema y tienen un impacto en la arquitectura de sus subsistemas.

Los patrones de arquitectura representan el nivel más alto en el sistema de patrones propuesto en Pattern Oriented Software Architecture - Volume 1 (Buschmann, et al., 1996) (en adelante, POSA). Cada patrón de arquitectura ayuda a conseguir una propiedad específica en el sistema global; por ejemplo, la adaptabilidad de la interfaz de usuario. Los patrones que dan soporte a características similares se agrupan en una misma categoría.

Uno de los aspectos más útiles de los patrones es que muestran los atributos de calidad conocidos. Este es el motivo por el que el arquitecto escoge un patrón particular y no uno al azar. Algunos patrones conocidos representan soluciones a los problemas de rendimiento, mientras que otros se prestan muy bien a sistemas de alta seguridad, otros se han utilizado con éxito en sistemas de alta disponibilidad. Elegir un patrón arquitectónico es a menudo la primera gran decisión de diseño del arquitecto. (Bass, et al., 2003)

El término estilo arquitectónico también ha sido ampliamente utilizado para describir el mismo concepto. (Bass, et al., 2003)

A continuación se muestra la categorización utilizada en dos de los sistemas de patrones arquitectónicos más extendidos y utilizados: el de POSA y el de Pattern of Enterprise Application Architecture (en adelante, PEAA).

En el POSA, se divide a los patrones en las siguientes categorías:

- Estructura (From Mud to Structure)
- Sistemas Distribuidos (Distributed Systems)
- Sistemas Interactivos (Interactive Systems)
- Sistemas Adaptables (Adaptable Systems)

En PEAA, se describe una gran cantidad de patrones orientados a la arquitectura de aplicaciones empresariales, que se definen en las siguientes categorías:

- Patrones de Dominio Lógico (Domain Logic Patterns)
- Patrones de Arquitectura de Fuente de Datos (Data Source Architectural Patterns)
- Patrones de Comportamiento de Objetos Relacionales (Object-Relational Behavioral Patterns)
- Patrones Estructurales de Objetos Relacionales (Object-Relational Behavioral Patterns)

- Patrones de Mapeo de Metadatos de Objetos Relacionales (Object-Relational Metadata Mapping Patterns)
- Patrones de Presentación Web (Web Presentation Patterns)
- Patrones de Distribución (Distribution Patterns)
- Patrones de Concurrencia fuera de Línea (Offline Concurrency Patterns)
- Patrones de Estado de Sesión (Session State Patterns)
- Patrones de Base (Base Patterns)
- Patrones de Presentación Web (Web Presentation Patterns)

De esta manera, el estilo está asociado a formas generales de organización –sistemas orientados al objeto- mientras que los patrones estarán asociados a formas más concretas, que tienen que ver con la especialización que adoptan los objetos y clases de acuerdo al tipo de aplicación o entorno tecnológico, a técnicas conocidas por su eficiencia para resolver ciertos problemas, etc.

Por ejemplo, pensando que el entorno de ejecución de una aplicación puede estar distribuido en distintos elementos –nodos- es aconsejable y beneficioso descomponer la aplicación en capas, correspondientes a los nodos donde podría tener que ejecutarse. La descomposición más habitual –es decir el patrón usualmente empleado- es dividir la aplicación en tres capas: presentación, lógica de negocio y la lógica de fuente de datos.

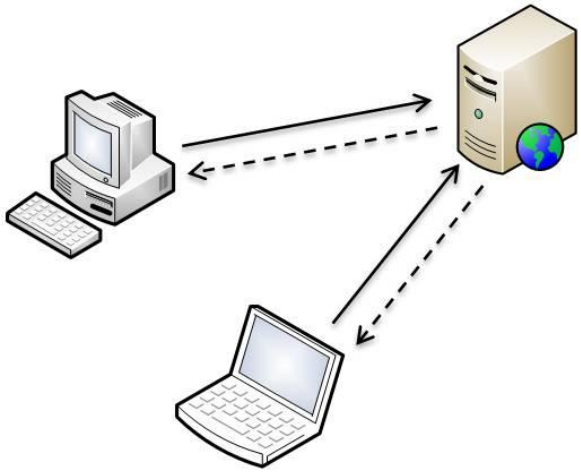
La diferencia entre los estilos y patrones arquitectónicos está en el nivel de abstracción que manifiestan. Los estilos se aplican a un nivel muy alto de abstracción, en el cual no interesa saber cuál es la semántica de los elementos que estamos utilizando, sólo hablamos de filtros, tubos u objetos sin interesarnos por lo que están representando. En el caso de los patrones, se tiene en cuenta el significado de los filtros, tubos u objetos, o en la descomposición en capas.

Un patrón arquitectónico, o una combinación de varios, no es una arquitectura de software completa. Sigue siendo un marco estructural para un sistema informático que debe ser especificado y refinado. Esto incluye la tarea de integrar la funcionalidad de la aplicación con el framework, y detallar sus componentes y las relaciones, quizás con la ayuda de los patrones de diseño y lenguajes. La selección de un patrón arquitectónico, o una combinación de varios, es sólo el primer paso en el diseño de la arquitectura de un sistema de software. (Buschmann, et al., 1996)

1.3.1 Arquitectura cliente-servidor

Esta arquitectura se encuentra dentro de la clasificación de estilo de llamada y retorno. El cliente y el servidor generalmente están localizados en diferentes sistemas, o pueden encontrarse en el mismo sistema. El cliente es la entidad que hace la petición por un servicio. El servidor es la entidad que provee el servicio correspondiente a la petición. El servicio debe procurar el resultado, el cual es retornado al cliente (Ver Figura 2).

Figura 2 Arquitectura cliente-servidor



Los clientes pueden ser clasificados como clientes “flacos” y/o “gordos”. Lo clientes gordos típicamente contienen, además de la lógica de presentación, gran parte de la lógica de negocio de la aplicación. Lo clientes flacos manejan usualmente sólo la lógica de presentación lo que adiciona grandes ventajas como permitir que futuros cambios de negocio en la aplicación no afecten al cliente.

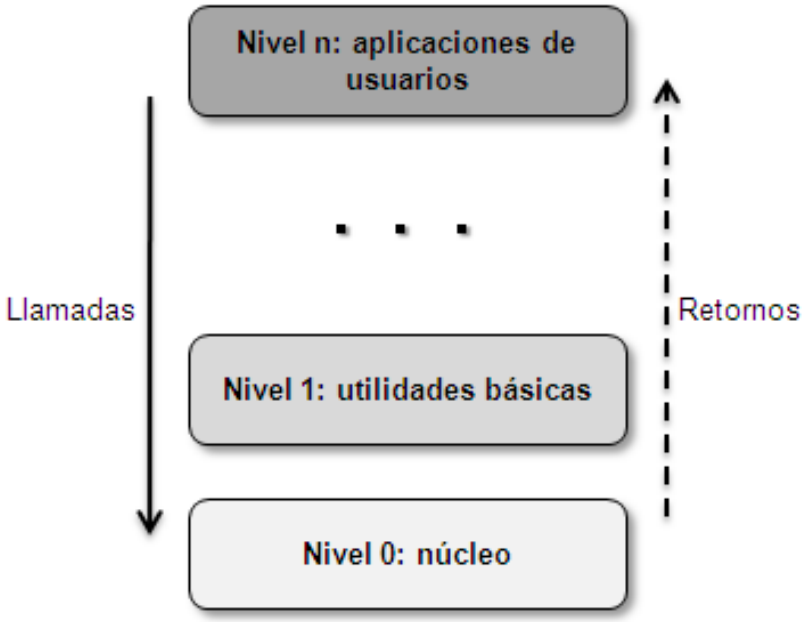
1.3.2 Arquitectura en capas

Garlan y Shaw definen el estilo en capas como una organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior.

En este estilo los componentes son las capas o niveles, que pueden estar implementadas internamente tanto por objetos como por procedimientos. Cada nivel tiene asociada una funcionalidad: los niveles bajos implementan funciones simples, ligadas al hardware o al entorno, mientras que los niveles altos implementan funciones más abstractas. El mecanismo de interacción entre componentes es el de llamadas a procedimientos (métodos), con la restricción de que son las capas superiores las que invocan funciones o métodos de las

inferiores. Una variante muy común es la que sólo permite realizar llamadas entre niveles adyacentes. (Ver Figura 3)

Figura 3 Estilo arquitectónico de organización en capas o niveles



Este diseño basado en niveles de abstracción crecientes permite a los implementadores la partición de un problema complejo en una secuencia de pasos incrementales. Permite realizar optimizaciones y refinamientos enfocando los cambios en un solo lugar. Proporciona amplia reutilización dada la división bien definida de responsabilidades. Al igual que los tipos de datos abstractos, se pueden utilizar diferentes implementaciones o versiones de una misma capa en la medida que soporten las mismas interfaces de cara a las capas adyacentes. Esto conduce a la posibilidad de definir interfaces de capa estándar, a partir de las cuales se pueden construir extensiones o prestaciones específicas.

Entre las propiedades que presenta esta organización arquitectónica se pueden citar las siguientes:

- Se facilita la migración del sistema. El acoplamiento con el entorno ésta localizado en el nivel inferior. Para transportar el sistema a un entorno diferente basta con implementar de nuevo este nivel.
- Cada nivel implementa interfaces claras y lógicas, lo que facilita la sustitución de una implementación por otra.

- Permite trabajar en varios niveles de abstracción. Para implementar los niveles superiores no necesitamos conocer el entorno subyacente, sólo las interfaces que proporcionan los niveles inferiores.

Una especialización muy usada de la arquitectura en capas es la arquitectura de tres capas donde se observan muy bien delimitadas las responsabilidades de cada funcionalidad en la aplicación.

Figura 4 Arquitectura en tres capas

En la Figura 4 se ejemplifica una arquitectura de tres capas, donde cada capa está muy bien delimitada de las demás. Una capa superior interactúa con una capa inferior mediante interfaces que definen las funcionalidades que la misma debe brindar. Las capas de la aplicación pueden residir tanto en el mismo nodo físico como en nodos diferentes.

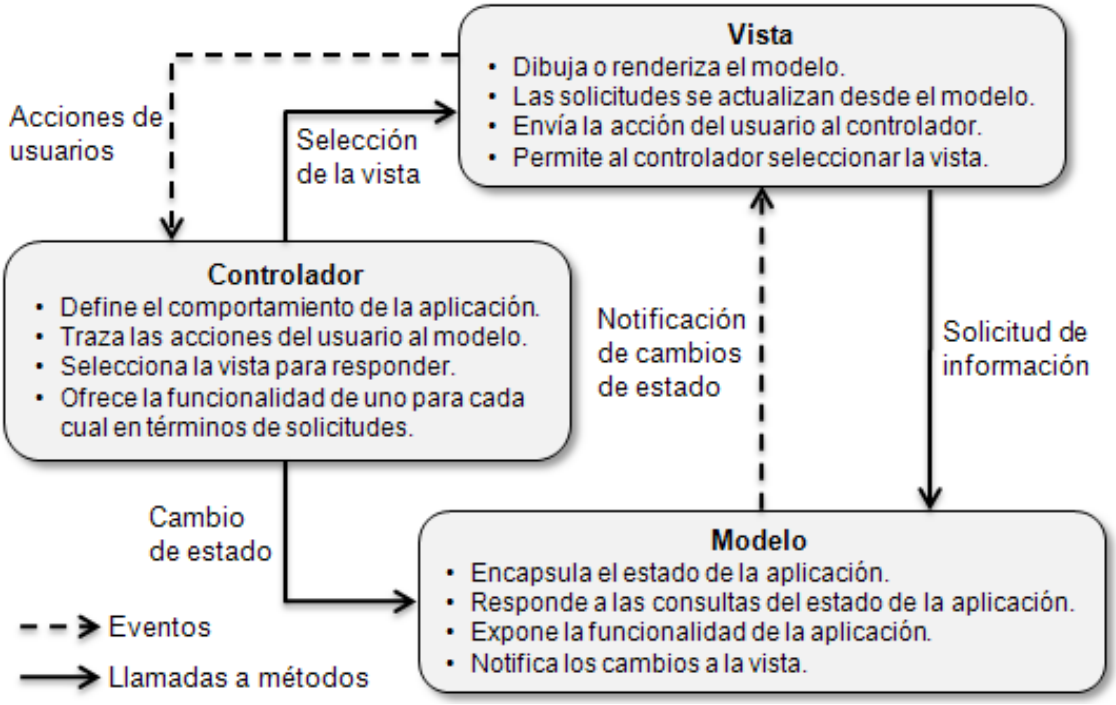
1.3.3 Modelo Vista Controlador

Modelo – Vista – Controlador (MVC) es un patrón de arquitectura de software (considerado también por algunos autores como un estilo arquitectónico) que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. Fue descrito por primera vez en 1979 por Trygve Reenskaug, realizando la primera implementación para Smalltalk-80.

MVC divide una aplicación interactiva en tres componentes. El modelo contiene la funcionalidad básica y los datos. La vista muestra la información al usuario. Los controladores manejan las entradas de los usuarios. Las vistas y los controladores integran la interfaz de usuario. Un

mecanismo de cambio-propagación asegura la coherencia entre la interfaz de usuario y el modelo. (Buschmann, et al., 1996) (Ver Figura 5)

Figura 5 Esquema del patrón Modelo-Vista-Controlador



El modelo representa los datos y las reglas de negocio que rigen su acceso y actualización; puede verse como una modelación de los procesos del mundo real.

Las vistas se encargan de presentar los datos obtenidos del modelo. Es responsabilidad de las vistas mantener la información actualizada, esto se puede logar a través de peticiones de actualización al modelo o a través de notificaciones de cambio que el modelo emite (eventos).

El controlador actúa como un traductor de las acciones que se realizan en las vistas en las operaciones que ocurren en el modelo. Las acciones realizadas por el modelo desencadenan la activación de procesos de negocio o cambian el estado del modelo. Sobre la base de las acciones del usuario y los resultados del modelo, el controlador responde mediante la selección de la vista apropiada.

1.4 Patrones de diseño

Un patrón de diseño aborda problemas recurrente del diseño orientado a objetos. En él se describe el problema, la solución, el momento de aplicar la solución, y sus consecuencias.

También da consejos de aplicación y ejemplos. La solución es un conjunto de clases y objetos y las relaciones que se establecen entre ellos para resolver el problema. La solución es personalizada y aplicada para resolver el problema en un contexto particular.

Aunque la orientación a objetos facilita la reutilización de código, la reutilización efectiva sólo se produce a partir de un buen diseño basado en el uso de soluciones que han probado su utilidad en situaciones similares; por lo que los patrones constituyen un conjunto plantillas básicas que cada diseñador adapta a las peculiaridades de su aplicación, logrando soluciones elegantes y efectivas.

No obstante, no fue hasta principios de 1990 cuando los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro Design Patterns escrito por GoF[4], en el que se recogen 23 patrones de diseño comunes.

Los patrones de diseño tienen su origen en los trabajos de Gamma y colaboradores (Gamma, et al., 1994) quienes no sólo realizan una definición de los mismos y sugieren cómo deben describirse, sino que ofrecen todo un catálogo de patrones básicos aplicables a la programación orientada a objetos en general. Posteriormente, han sido propuestos numerosos patrones, tanto de propósito general, como específicos para multitud de áreas de aplicación.

"Un patrón de diseño ofrece un sistema para el perfeccionamiento de los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. En él se describe una estructura común de comunicación de los componentes que resuelve un problema de diseño dentro de un contexto particular." (Buschmann, et al., 1996)

Los patrones de diseño son patrones de mediana escala. Son más pequeños en la escala de los patrones arquitectónicos, pero tienden a ser independientes de un lenguaje de programación o paradigma de programación. La aplicación de un patrón de diseño no tiene ningún efecto sobre la estructura fundamental de un sistema de software, pero puede tener una gran influencia en la arquitectura de un subsistema. Muchos patrones de diseño proporcionan estructuras más complejas para descomponer los servicios o componentes. Otras abordan la cooperación efectiva entre ellos. (Buschmann, et al., 1996)

[4] GoF: Gang of Four (la banda de los cuatro) se refiere a Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides los cuatro autores de Design Patterns: Elements of Reusable Object-Oriented Software.

Los patrones no son bibliotecas de clases, sino más bien un esqueleto básico que cada diseñador adapta a las peculiaridades de su aplicación. Los patrones se describen en forma textual, acompañada de diagramas y pseudocódigo.

Entre las ventajas de los patrones de diseño se pueden citar que son soluciones simples y técnicas que se aplican a problemas muy comunes que aparecen en el diseño orientado a objetos, facilitando la reutilización del diseño. Tienen carácter fundamentalmente práctico, que indica cómo resolver el problema desde el punto de vista técnico de la orientación a objetos. Entre sus inconvenientes está, en primer lugar, que son soluciones concretas a problemas concretos. Todos ellos son independientes, lo que hace difícil, dado un problema determinado, averiguar si existe un patrón que lo resuelve. En segundo lugar, el uso de un patrón no se refleja claramente en el código de la aplicación. A partir de la implementación de un sistema es difícil determinar que patrones de diseño se han utilizado en la misma. Por último, si bien permiten reutilizar el diseño, es mucho más difícil reutilizar su implementación. Este describe clases que juegan papeles genéricos, explicadas normalmente con ayuda de un ejemplo, pero su implementación, adaptándolo a las peculiaridades de una aplicación en particular, consistirá en clases concretas, dependientes del dominio de aplicación. (Canal Velasco, 2000)

2 *Plataforma y frameworks de desarrollo*

- Plataforma J2EE
- Spring Framework
- iBatis
- Acegi Security
- Direct Web Remoting

CAPÍTULO 2.- PLATAFORMA Y FRAMEWOKS DE DESARROLLO

En el presente capítulo se abordan los elementos más significativos de la plataforma de desarrollo seleccionada, así como las características fundamentales de los frameworks utilizados.

2.1 Plataforma J2EE

J2EE[5] es el acrónimo de **J**ava **2 E**nterprise **E**dition diseñado por Sun Microsystems y define un estándar para el desarrollo de aplicaciones empresariales multicapas.

Simplifica las aplicaciones empresariales basándolas en componentes modulares y estandarizados, proporcionando un completo conjunto de servicios a través de estos componentes, y manejando muchas de las funciones de la aplicación de forma automática, sin necesidad de una programación compleja. (Sun Microsystems, Inc.) En otras palabras, la idea subyacente a la plataforma J2EE es proporcionar un estándar sencillo y unificado para aplicaciones distribuidas en un modelo de aplicación basado en componentes.

Está basado en J2SE (Java 2 Standard Edition) manteniendo muchas de sus características, como su portabilidad *"write once, run anywhere"*, pero añade una serie de elementos necesarios en entornos empresariales, relativos a redes, acceso a datos y entrada/salida que requieren mayor capacidad de procesamiento, almacenamiento y memoria. La decisión de separarlos es debido a que no todas estas características son necesarias para el desarrollo de aplicaciones estándar.

J2EE ofrece muy buenas perspectivas para la implementación de software empresarial. Entre las ventajas que ofrece se pueden citar las siguientes:

- *Soporte para múltiples sistemas operativos:* es posible desarrollar arquitecturas basadas en J2EE usando cualquier sistema operativo donde pueda ejecutarse una máquina virtual de Java, teniendo la gran ventaja de una independencia total de la arquitectura de hardware.
- *Objetos gestionados:* proporciona un entorno gestionado para componentes y las aplicaciones son céntricas respecto al contenedor. Al ser gestionados, los componentes

[5] J2EE: Java 2 indica que es la versión 2 de la plataforma Java, mientras que Enterprise Edition se refiere a la edición empresarial.

utilizan la infraestructura proporcionada por los servidores de J2EE sin que el programador sea consciente de ello. Las aplicaciones de J2EE son también declarativas, un mecanismo con el que puede modificar y controlar el funcionamiento de las aplicaciones sin cambiar de código. (Beust, et al., 2002)

- *Reusabilidad:* la separación de los requisitos de una aplicación en sus partes integrantes es un modo de conseguir la reutilización; utilizar la orientación a objeto para encapsular la funcionalidad compartida es otro. Sin embargo, a diferencia de los objetos, los componentes distribuidos requieren una infraestructura más compleja para su construcción y gestión. J2EE ofrece una arquitectura notablemente rigurosa para la construcción y gestión de componentes distribuidos así como su reutilización de componentes. Además, ya que los componentes son reproducibles, lo cual quiere decir que es posible identificar ciertos metadatos sobre los componentes, las aplicaciones pueden ser creadas componiendo tales componentes. Ambas características fomentan la reutilización del código a alta granulometría. (Beust, et al., 2002)
- *Modularidad:* es siempre recomendable dividir una aplicación en módulos discretos, cada uno de ellos responsable de una tarea específica, logrando que las aplicaciones sean mucho más fáciles de mantener y comprender. Por ejemplo, los servlets de Java, las JSP[6] y EJB[7] proporcionan un modo de modularizar las aplicaciones, fragmentando las aplicaciones en diferentes niveles y tareas individuales. (Beust, et al., 2002)
- *Fácil integración con los sistemas de información existentes:* JDBC, una tecnología J2EE, es una API de Java para bases de datos SQL, lo que permite que se acceda a cualquier tipo de información recogida en tablas que pueda existir. JNDI permite a las aplicaciones que utilizan tecnología Java tener acceso a los servicios de nomenclatura y directorio de la empresa.
- *Gran variedad de herramientas, servidores y componentes*: para desarrollar las aplicaciones que necesiten el equipo de desarrollo puede seleccionar las soluciones que mejor se adapten a sus necesidades, sin tener que ajustarse a las ofertas de un solo fabricante.

[6] JSP acrónimo de Java Server Page

[7] EJB acrónimo de Enterprise Java Beans

- *Organismo de control:* J2EE está controlada por un organismo formado por más de 400 empresas. Entre esas empresas se encuentran muchas de las más importantes del mundo informático, tales como Sun Microsystems, IBM, Oracle, BEA, HP, AOL, etc.
- *Competitividad:* muchas empresas crean soluciones basadas en J2EE que ofrecen características tales como rendimiento y precios muy diferentes. De este modo, se ha desarrollado a un nivel exponencial la plataforma y los clientes tienen la posibilidad de escoger entre una gran cantidad de opciones.
- *Madurez:* creada en el año 1997, J2EE ya tiene varios años de vida y una amplia cantidad de proyectos importantes están desarrollados sobre esta plataforma.
- *Soluciones libres:* sobre la plataforma J2EE es posible crear arquitecturas basadas por completo en productos de software libre. No solo eso, sino que los arquitectos de software disponen de muchas soluciones libres para cada una de las partes de su arquitectura.

2.1.1 Modelo de desarrollo J2EE

Para finalizar esta breve descripción de la plataforma J2EE es importante tener conocimiento de algunos conceptos básicos sobre el modelo de desarrollo de aplicaciones bajo esta plataforma. La plataforma J2EE define un modelo de programación encaminado a la creación de aplicaciones basadas en n-capas. La lógica de la aplicación se divide en componentes con diferentes funciones y que están distribuidos en un ambiente multicapas.

Aunque puede variar, típicamente una aplicación suele tener cinco capas diferentes:

- Capa cliente: representa la interfaz de usuario que interactúa con el cliente.
- Capa de presentación: representa el conjunto de componentes que generan la información que se mostrará en la interfaz de usuario del cliente. Normalmente se crea a través de componentes basados en Servlets y JSP.
- Capa de lógica de negocio: contiene los componentes de negocio reutilizables.
- Capa de integración: aquí se encuentran los componentes que permiten hacer más transparente el acceso a la capa de sistemas de información. Este es el lugar idóneo para implementar la lógica de objetos de acceso a datos (DAO, data access object).
- Capa de recursos: engloba los sistemas en los cuales la información se almacena físicamente como bases de datos relacionales, sistemas legacy (heredados), bases de datos orientadas a objetos, bancos de ficheros de datos, etc.

Las ventajas de un modelo como este son muy importantes. Al tener las capas separadas existe un bajo acoplamiento entre las mismas, de modo que es mucho más simple hacer modificaciones en ellas sin que afecten a las demás. Todo esto conlleva a la obtención de mejoras en cuanto a mantenibilidad, extensibilidad y reutilización de componentes. Otra ventaja que se obtiene es la de promover la heterogeneidad de los clientes, ya que añadir nuevos tipos de clientes se reduce a añadir nuevas capas de interfaz de usuario y presentación, sin necesidad de modificar el resto de las capas.

Como ya se ha dicho, el modelo de desarrollo con J2EE está basado en componentes reutilizables, con el objetivo de aumentar la reusabilidad de las aplicaciones. Estos componentes, además, gracias a las especificaciones, son intercambiables entre servidores de aplicaciones, por lo que la portabilidad de las aplicaciones es máxima.

2.2 Spring Framework

Spring es un framework de código abierto, creado por Rod Johnson y que se describe en su libro Expert One-on-One J2EE Design and Development (Johnson, 2003). Fue creado para abordar la complejidad del desarrollo de aplicaciones empresariales y es el resultado de la amplia experiencia de trabajo de su autor como consultor independiente de software para clientes del sector financiero en el Reino Unido. Aunque inicialmente fue destinado a la plataforma Java recientemente se ha portado a la plataforma .NET y se encuentra actualmente disponible bajo la licencia de código abierto Apache 2.0 (Walls, et al., 2008).

Un eje central de Spring es permitir la reutilización de objetos del negocio y de acceso a datos que no están vinculados a servicios J2EE específicos. Tales objetos pueden ser reutilizados a través de entornos J2EE (web o EJB), aplicaciones independientes, entornos de prueba y otros sin ningún tipo de molestia.

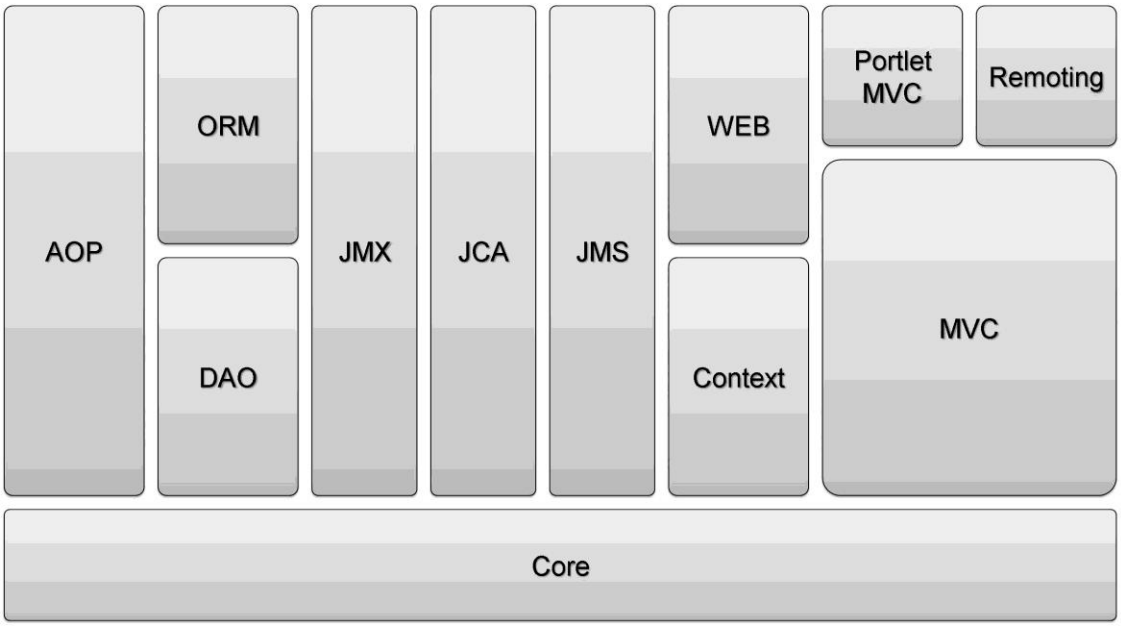
Las características más significativas de Spring Framework pueden ser resumidas en (Johnson, et al., 2005):

- No es un framework invasivo.
    - El código escrito en aplicaciones que usen Spring pueden ejecutarse sin usar el framework u otro contenedor.
    - La dependencia con Spring es mínima posibilitando la reutilización en otro contenedor de peso ligero.
    - Posibilita la migración a versiones futuras del framework de forma sencilla.

- Provee un modelo de programación consistente que desacopla el código de la aplicación de los detalles del entorno, haciendo el código menos dependiente del contexto de ejecución.
- Facilita las buenas prácticas de programación reduciendo la complejidad del código mediante el uso de interfaces, ocultando elegantemente la implementación de las clases y los requisitos de configuración.
- Promueve el uso de puntos de extensión (pluggability) a través de interfaces.
- Facilita la extracción de los valores de configuración desde el código Java hacia un XML [8] o un fichero de propiedades logrando un código configurable, de fácil mantenibilidad y reusabilidad.
- Está diseñado para que las aplicaciones sean fáciles de probar.
- Posibilita la elección de la arquitectura a emplear posibilitando el uso de diferentes frameworks en las distintas capas de las aplicaciones.
- No reinventa la rueda, utiliza los productos ya existentes y probados aportando una solución fácil de usar e integrar.

Spring framework está formado por diferentes módulos bien definidos cuyo uso está condicionado por los requisitos del sistema a implementar. (Ver Figura 6)

Figura 6 Módulos de Spring Framework



[8] Extensible Markup Language (Lenguaje de Marcado extensible)

Spring Framework permite la fácil integración con otros framework de propósitos específicos entre los que se encuentran (Johnson, et al., 2005):

- **Persistencia:** JDO, TopLink, Apache OJB, Hibernate, iBATIS
- **Web:** Struts, WebWork, Tapestry, JSF, Spring MVC
- **AOP[9]:** AspectJ, Spring AOP
- **Otros:** Jasper Reports, Quartz scheduler, Velocity, FreeMarker

En epígrafes posteriores se abordarán con mayor profundidad los framework que se emplearon en el desarrollo del sistema.

2.2.1 Inversión del Control e Inyección de Dependencias

Spring se identifica plenamente con la Inversión del Control (IoC) y, concretamente, con la Inyección de Dependencias (ID). Es a menudo considerado como un contenedor de inversión del control, aunque en realidad es mucho más.

IoC se entiende mejor a través de la expresión del "Principio de Hollywood"[10], que básicamente significa *"No me llames, yo te llamo"[11].* Con el Principio de Hollywood, el código del framework invoca el código de la aplicación, coordinando el flujo general de trabajo, en lugar de que las aplicaciones invoquen el código del framework. (Johnson, et al., 2005)

El aspecto que se invierte es el modo en que los objetos obtienen las instancias de los objetos que colaboran con él o de los cuales depende. Bajo esta premisa Martin Fowler denominó como inyección de dependencias a esta forma de invertir el control. (Fowler, 2004)

Los términos IoC e ID, son conceptos estrechamente relacionados, pero en realidad no son la misma cosa. IoC es un concepto mucho más general, y puede ser expresado de muchas maneras diferentes. ID no es más que un ejemplo concreto de IoC. (Ladd, et al., 2006)

---

[9] AOP siglas de Aspect-oriented programming (Programación orientada a aspectos). En el epígrafe 2.2.3 se aborda con más profundidad el tema.

[10] El Principio de Hollywood es una metodología de diseño de software que lleva su nombre por el cliché de la respuesta dada a los aficionados en las audiciones en Hollywood: *"Don't call me, I'll call you."* Se trata de un paradigma útil que ayuda en el desarrollo de código con una alta cohesión y bajo acoplamiento que es más fácil de depurar, mantener y poner a prueba.

[11] Original en inglés: "Don't call me, I'll call you."

ID se basa en los constructores del lenguaje Java, en lugar de la utilización de las interfaces de un framework específico. En lugar de que las aplicaciones utilicen el código de las API de los frameworks para resolver las dependencias, tales como parámetros de configuración de los objetos y la colaboración, las clases de la aplicación exponen sus dependencias a través de métodos o constructores que el framework puede llamar con los valores en tiempo de ejecución, sobre la base de la configuración. (Johnson, et al., 2005)

IoC cubre una amplia gama de técnicas que permiten a un objeto convertirse en un participante pasivo en el sistema. Cuando la técnica de IoC es aplicada, los objetos seden la responsabilidad de la creación de sus dependencias a una tercera entidad (el framework) que administra los objetos del sistema, la cual "inyecta" las dependencias en los objetos correspondientes. Algunos ejemplos incluyen la creación de objetos o la delegación a los objetos dependientes. IoC puede eliminar estos problemas de los objetos con la inyección de dependencia y la programación orientada a aspectos, respectivamente. El resultado final es la eliminación de código duplicado y clases simplificadas que se centran en su lógica de negocio. (Ladd, et al., 2006)

La Figura 7 muestra cómo IoC e ID se relacionan entre sí. (Interface21, et al., 2007)

Figura 7 Relación entre Inversión del Control e Inyección de Dependencias

Como se muestra en la Figura 7 IoC prevé dos formas de resolver las dependencias: la búsqueda de dependencias y la inyección de dependencia. En la primera de ellas la responsabilidad de resolver las dependencias está en las manos de la aplicación. Esta ha sido la forma estándar de resolución de dependencias en Java desde hace muchos años, pero siempre requieren acoplarse al código. Spring framework apoya también la búsqueda de dependencia. En cambio, con la inyección de dependencias la responsabilidad de resolver las

dependencias está en las manos del contenedor de IoC, como en Spring framework. Un archivo de configuración define la forma en que las dependencias de una aplicación van a ser resueltas. El fichero de configuración es leído por el contenedor, el cual creará los objetos que se definen en este fichero e inyectará estos objetos en otros componentes de la aplicación. (Interface21, et al., 2007)

Spring es compatible con varios tipos de inyección de dependencias: (Johnson, et al., 2005)

- *Setter Injection* (Inyección por métodos set): La inyección de dependencias a través de métodos setter de JavaBean.
- *Constructor Injection* (Inyección por constructores): La inyección de dependencias a través de parámetros del constructor.
- *Method Injection* (Inyección por métodos): Una forma que raramente se utiliza, en la que la inyección de dependencia en el contenedor se encarga de la implementación de los métodos en tiempo de ejecución. Por ejemplo, un objeto puede definir un método abstracto protegido, y el contenedor puede implementarlo en tiempo de ejecución para retornar un objeto resultante de un contenedor de búsqueda.

Spring permite que todas estas formas puedan ser mezcladas para la configuración de una clase.

Algunos de los beneficios del uso de la inyección de dependencias son los siguientes (Machacek, et al., 2008):

- Reduce la cantidad de código necesario para acoplar los diferentes componentes.
- Externaliza la configuración de las dependencias permitiendo la reconfiguración sin necesidad de recompilar la aplicación.
- La gestión de las dependencias se realiza en un solo lugar (contenedor IoC de Spring) simplificando esta tarea y disminuyendo el número de errores.
- Mejora la forma de realizar las pruebas al sistema.
- Fomenta el buen diseño de aplicaciones.

2.2.2 Spring MVC

En el desarrollo de aplicaciones web, la administración del estado, el flujo de trabajo, y la validación son características importantes que deben abordarse. También involucran una gran cantidad de tiempo y esfuerzo por parte de los programadores, sobre todo con la plataforma J2EE. Por este motivo existen frameworks web cuya meta principal es simplificar el desarrollo

de la capa web tanto como sea posible. Entre los mas reconocidos en la comunidad Java se pueden citar los siguientes: Struts, WebWork, Tapestry, JSF[12] y Spring MVC.

Spring MVC es un potente entorno para crear aplicaciones web limpias y débilmente acopladas. Está basado en el patrón Modelo-Vista-Controlador (MVC) y brinda una implementación *"out of the box"*[13] del flujo de trabajo típico para aplicaciones web. Es muy flexible, posibilitando utilizar una gran variedad de tecnologías para las vistas. Asimismo, permite la integración plena con los demás módulos del framework y emplea a su vez la inyección de dependencias. (Johnson, et al., 2005)

Una aplicación Spring MVC está claramente dividida entre vistas, clases controladoras, y el modelo. Las vistas son típicamente JSP, aunque puede usar una variedad de otras tecnologías. El conjunto de clases controladoras que brinda el framework abarcan los elementos más comunes en aplicaciones web, desde la creación de formas básicas hasta el llenado de formularios en una forma compleja. Posee un excelente apoyo para tareas como la validación de los datos y la presentación de contenido en múltiples formatos de salida. (Minter, 2008)

Una de las principales ventajas de Spring MVC es que se separa claramente todas las responsabilidades en una aplicación web. Los modelos contienen los datos que se presentarán en la respuesta y las vistas realizan la representación real de los mismos, mientras que los controladores procesan la petición del usuario, invocan el servicio de negocio correspondiente para obtener el modelo de datos a mostrar, y elige la vista adecuada para generar la respuesta para el usuario. Se debe tener en cuenta que Spring ofrece una elegante disociación de estos tres conceptos, posibilitando que sean más independientes y maximizando el potencial de reutilización. Spring Web MVC es una solución flexible para la construcción de aplicaciones web. (Johnson, et al., 2005)

Componentes principales de Spring MVC (Interface21, et al., 2007)

- *DispatcherServlet:* La implementación de Spring de la arquitectura MVC para aplicaciones web se basa en el `DispatcherServlet`. Es un controlador frontal[14] para

[12] JSF: Java Server Face.

[13] "*out of the box*": Elementos, funcionalidades o características que no requieren instalaciones adicionales, plug-ins, paquetes de expansión, o de los productos.

[14] Controlador frontal: Patrón de diseño que se basa en usar un controlador como punto inicial para la gestión de las peticiones. El controlador gestiona estas peticiones, y realiza algunas funciones como: comprobación de

una aplicación Spring MVC que procesa las solicitudes e invoca el controlador adecuado para manejarla. Todas las solicitudes pasan a través de este servlet, gestionando todos los elementos que intervienen en el procesamiento de una solicitud. Es declarado y configurado una sola vez igual que un servlet normal dentro de una aplicación.

- *Controllers:* El trabajo de manejar las peticiones individuales de cada página es responsabilidad de los controladores. Spring MVC proporciona una amplia colección de tipos de controladores. La Figura 8 ilustra la jerarquía de los mismos. Los controladores son responsables sólo de aceptar una nueva solicitud y recoger el resultado del procesamiento de la petición. El controlador entonces crea un modelo con el fin de pasárselo a la vista. El controlador no gestiona la presentación de las vistas, pero usualmente realiza la selección de las mismas.
- *HandlerMapping:* El trabajo de analizar una solicitud para determinar qué controlador debe ser invocado se delega a la interfaz `HandlerMapping`. Normalmente, el URI[15] es el factor determinante, pero puede tener otras implementaciones. También es posible declarar y configurar múltiples `HandlerMapping` para dar cabida a múltiples estrategias de resolución de controladores, siempre especificándole el orden en que las instancias de los `HandlerMapping` deben ser consultadas.
- *Model y View:* El modelo (`Model`) es una colección de objetos destinados a ser presentados en la interfaz de usuario mediante la vista (`View`). Puede contener los resultados de las operaciones realizadas por el modelo de dominio o por objetos personalizados para mostrarlos en la vista. El modelo se implementa como un Map, que como llaves tiene los identificadores de cada objeto. Spring MVC combina el modelo y la vista en una clase `ModelAndView`. Los controladores son responsables de crear y rellenar una instancia de `ModelAndView` antes de completar su trabajo. La vista y el modelo se combinan de esta manera simplemente porque el controlador tiene que devolver ambos objetos cuando haya terminado el procesamiento. No se limitan los objetos que se ponen en el modelo, siempre y cuando la vista conozca cómo mostrarlos. La presentación en la interfaz de usuario de los objetos del modelo es el trabajo de la vista. Spring MVC presenta diferentes implementaciones de vistas como

---

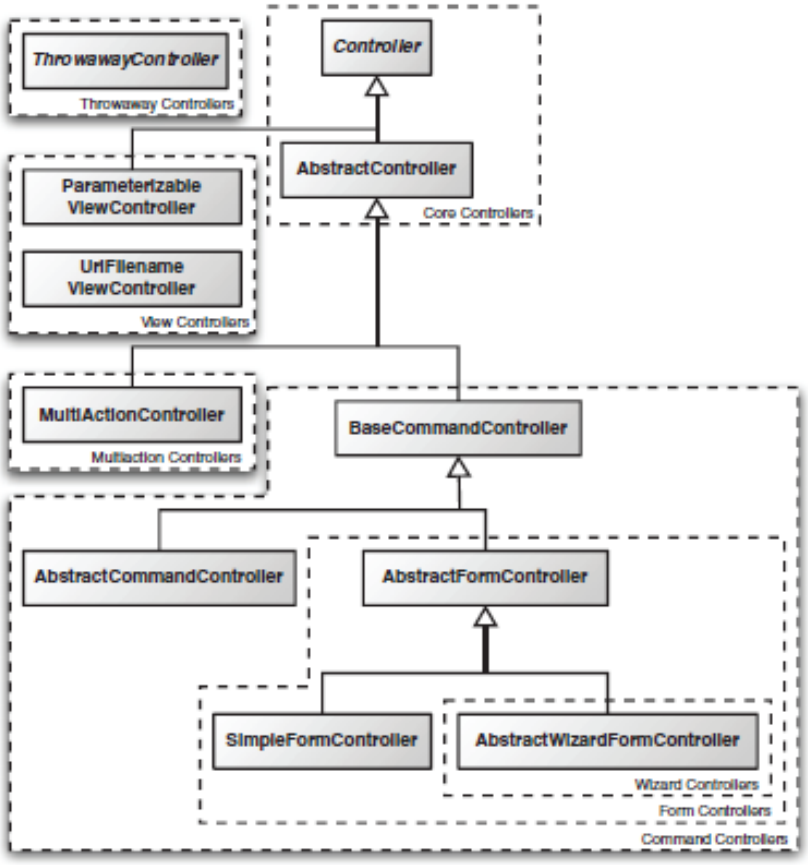
restricciones de seguridad, manejo de errores, mapear y delegación de las peticiones a otros componentes de la aplicación que se encargarán de generar la vista adecuada para el usuario.

[15] URI: Indicador de recursos uniforme (Uniform Resource Indicator).

son JSTL[16], XSLT[17], Velocity, FreeMaker, PDF, Excel y JasperReports, además de permitir vistas personalizadas para modelos propios de una aplicación.

- *ViewsResolvers:* Spring MVC hace corresponder las vistas a un nombre específico para su identificación, lo que permite desacoplar totalmente a la vista del controlador. La interfaz `ViewResolver` se encarga de resolver los nombres para una instancia de una vista. Se pueden especificar uno o más `ViewResolvers` definiendo la estrategia de resolución.

Figura 8 Jerarquía de clases de los controladores de Spring Framework



Flujo básico de una petición en Spring MVC (Ver Figura 9)

El flujo comienza cuando un cliente (por lo general un navegador web) realiza una petición al servidor. El primer componente que interviene es el DispatcherServlet (**A**) que actúa como un

16 JSTL: JavaServer Pages Standard Tag Library.

17 XSLT: Extensible Stylesheet Language Transforms.

controlador frontal delegando la responsabilidad de procesar la petición a otros componentes de la aplicación, específicamente a los controladores de Spring MVC.

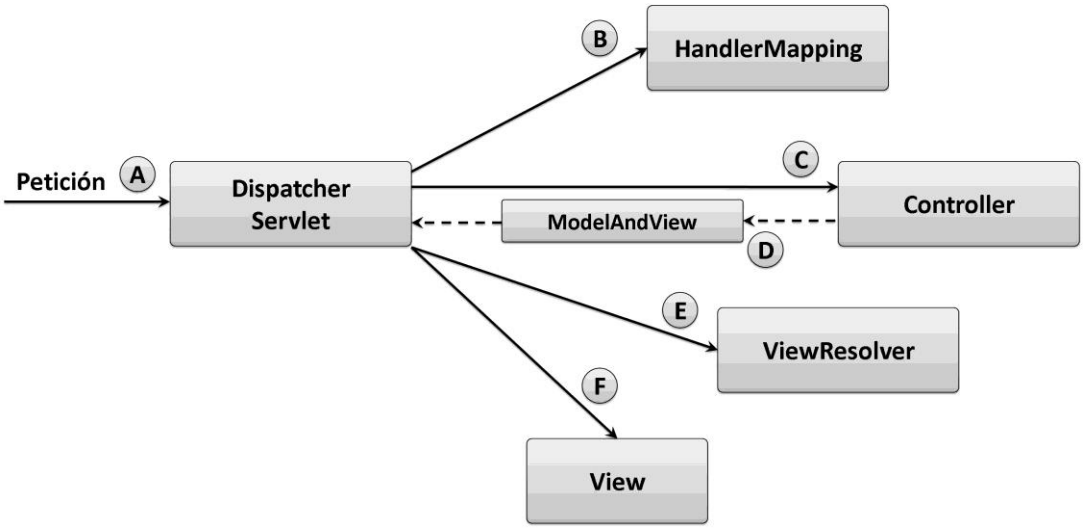
Usualmente una aplicación tiene un conjunto de controladores y el DispatcherServlet necesita decidir a cual controlador enviar la petición. Para ello consulta al HandlerMapping (**B**) (puede ser uno o varios) y determina cual será el próximo punto en el procesamiento de la solicitud. El HandlerMapping tiene en cuenta la URL contenida en la petición para hacer la correspondencia con los diferentes controladores.

La solicitud es enviada al Controller (**C**) para que procese la información y ejecute toda la lógica de negocio que tiene asociada. En un buen diseño esta lógica asociada a la petición no se realiza en el controlador (o muy poca de ella) sino que esta tarea es delegada a los objetos de servicio diseñados al efecto. Como resultado final el controlador devuelve un objeto ModelAndView (**D**) que es retornado al DispatcherServlet y que contiene la información que es necesaria retornar, además del nombre que identifica a la vista que se encargará de la transformación de estos datos para mostrarlos al usuario.

Mediante el ViewResolver (**E**) el DispatcherServlet selecciona la vista correspondiente al nombre lógico contenido en el ModelAndView y determina la View (**F**) que será la encargada de hacer la transformación de los datos del Model en un formato de presentación para el usuario.

Por último esta respuesta es enviada al DispatcherServlet el cual es el encargado de retornarla al cliente (navegador) que realizó la petición.

Figura 9 Flujo básico de una petición en Spring MVC



2.2.3 Spring AOP

En los últimos años, la Programación Orientada a Aspectos (AOP) se ha convertido en un tema de amplio debate, que aunque no es particularmente nuevo, sólo con la introducción de herramientas como AspectJ y Spring AOP es que se ha ganado en este sentido en el mundo Java.

AOP ofrece una forma diferente de pensar sobre la estructura del código, en comparación con la Programación Orientada a Objetos (POO) o a la Programación Procedimental aportando una técnica que permite la implementación de comportamientos genéricos para un conjunto de problemas que su solución no encajan de forma natural en estos paradigmas. (Minter, 2008)

Mientras que la POO modela objetos y conceptos del mundo real, tales como cuentas bancarias, y los organiza en jerarquías, AOP nos permite reflexionar sobre problemas que afectan a diversas partes del modelo de objetos y que no forman parte de su funcionalidad básica (incumbencias transversales), como por ejemplo la gestión de transacciones, logging, o monitoreo de fallas. Estas responsabilidades del sistema pueden ser difíciles de lograr con un enfoque tradicional orientado a objetos, dejando como opción la duplicación de código, o la utilización de un framework intrusivo con fines especiales. (Johnson, et al., 2005)

Estos servicios que no comprenden en si la lógica básica de los distintos componentes pueden introducir dos niveles de complejidad en el código: (Walls, et al., 2008)

- El código de estos servicios se duplica a través de múltiples componentes. Esto significa que si se necesita cambiar la forma en que esos trabajos se hacen, se tendrá que modificar múltiples componentes.
- Los componentes están plagados de código que no está alineado con la esencia de su funcionalidad.

Hay dos tipos principales de implementación de AOP. Los frameworks AOP, como AspectJ, proporcionan una solución en tiempo de compilación para la construcción de lógica basada en la AOP y añadirlo a la aplicación. La AOP dinámica, como en Spring, permite que la lógica transversal se aplique arbitrariamente a cualquier código en tiempo de ejecución. (Machacek, et al., 2008)

Spring AOP es un importante módulo que proporciona servicios críticos a nivel de sistema. Promueve el bajo acoplamiento y permite cuestiones transversales (como los servicios de

negocios y transacciones) que se separan en una forma más elegante. Permite que estos servicios sean aplicados con transparencia con relación al código. Con Spring AOP, es posible escribir aspectos personalizados y configurarlos de forma declarativa. Soporta la creación de aspectos a través de AOP Alliance y de AspectJ. (Kayal, 2008)

Spring AOP se utiliza en el propio framework para una variedad de propósitos: (Johnson, et al., 2005)

- *Gestión de transacciones de forma declarativa:* Tiene las siguientes grandes ventajas con respecto al las transacciones con EJB:
  - Se puede aplicar a cualquier POJO.
  - No está vinculado a JTA[18], pero puede trabajar con una variedad de APIs de transacciones (incluidos JTA). Por lo tanto, pueden trabajar en un contenedor web o aplicación independiente usando una sola base de datos, y no requiere un servidor de aplicaciones J2EE completo.
  - Soporta una semántica adicional que reduce al mínimo la necesidad de depender de un API de transacción propietario para forzar el rollback.
- *Intercambio en caliente:* un proxy AOP se pueden utilizar para proporcionar una capa de indirección. Este mecanismo es seguro en escenarios multihilos, proporcionando poderosas capacidades de intercambio.
- *Soporte para "objetos dinámicos":* Al igual que con intercambio en caliente, el uso de un proxy AOP puede permitir objetos "dinámico" tales como los objetos provenientes de lenguajes scripts como Groovy o Beanshell soportando a la recarga e implementando interfaces adicionales que permiten la manipulación de los mismo.
- *Acceso remoto:* Para acceso remoto del lado del cliente, con AOP se hace conexión con el servidor y se devuelve el resultado de la ejecución del método recibido desde el servidor. Para el acceso remoto desde el servidor, un objeto proxy se asegura que sólo los métodos de las interfaces seleccionados se exporten.
- *Seguridad:* Acegi Security es un framework asociado que utiliza AOP para ofrecer un sofisticado modelo de seguridad declarativo.

Los conceptos fundamentales que maneja Spring AOP son los siguientes (Johnson, et al.):

- *Aspect:* es la funcionalidad transversal que se quiere separar del resto del sistema.

[18] JPA acrónimo de Java Transaction API.

- *Joint point*: es un punto durante la ejecución de un programa, tales como la ejecución de un método o el tratamiento de una excepción.
- *Advice:* es la acción que realiza el aspecto en un joint point en particular. Pueden ser de diferentes tipos en dependencia del momento en que se ejecutarán (antes, después y durante el lanzamiento de una excepción).
- *Pointcut:* es un predicado que se asocia a los join points y se ejecuta si coincide con alguno de ellos.
- *Target:* es el objeto al que será aplicado el aspecto.
- *AOP proxy*: es un objeto creado por el framework AOP para implementar los contratos de los aspectos. Spring AOP utiliza un proxy dinámico del propio JDK o un proxy CGLIB.

2.3 iBatis

El proyecto iBatis fue creado en el 2001 por Clinton Begin e inicialmente se enfocó al desarrollo de software criptográfico escrito en Java y bajo una licencia open source. Un año más tarde efectuó un giro radical en su producción. En la actualidad el proyecto está centrado en un framework de la capa de persistencia perteneciente al proyecto Apache y bajo la licencia de Apache 2.0.

iBatis es conocido como un data mapper . Martin Fowler describe el patrón Data Mapper de la forma siguiente:

> *"Una capa de Mappers*[19] *que mueve los datos entre los objetos y la base de datos manteniendo independientes el uno del otro y del propio mapper."* (Fowler, 2002)

La ventaja de la estrategia de SQL Maps es que el desarrollador tiene el completo control sobre el SQL, lo que permite una completa personalización para una base de datos específica (tanto por el desarrollador o por un administrador de bases de datos). La desventaja es que no hay abstracción de la base de datos específica como el incremento automático de las columnas, las secuencias, selecciones de actualización, y así sucesivamente. El mapeado de sentencias debe ser definido para cada gestor de base de datos, teniendo en cuenta sus características particulares. (Johnson, et al., 2005)

[19] Mappers: Objeto que establece una comunicación entre dos objetos independientes. (Fowler, 2002)

Este framework de persistencia permite mapear las consultas SQL con objetos. Las consultas SQL son desacopladas de la aplicación poniéndolas en archivos XML. iBatis relaciona objetos con los resultados de ejecutar procedimientos almacenados o sentencias SQL usando un descriptor XML. El mapeo de los objetos recuperados es automático o semiautomático.

La idea básica del framework es reducir significativamente la cantidad de código que un desarrollador de Java normalmente necesita para acceder a una base de datos relacional con el uso de archivos XML que contienen las consultas SQL.

Su objetivo no es lograr una abstracción de la base de datos específica a utilizar, sino más bien dar control completo sobre el SQL empleado, permitiendo que el desarrollador aproveche toda la potencia de la base de datos, lo que incluye la sintaxis SQL. (Johnson, et al., 2005)

2.3.1 Características

Cualquier implementación de iBatis incluye los siguientes componentes: (Apache Software Foundation)

- Data Mapper: proporciona una forma sencilla de interacción de los datos Java y. NET con las bases de datos relacionales.
- Data Access Object: abstracción que oculta la persistencia de los objetos y proporciona una API de acceso a los datos al resto de la aplicación.

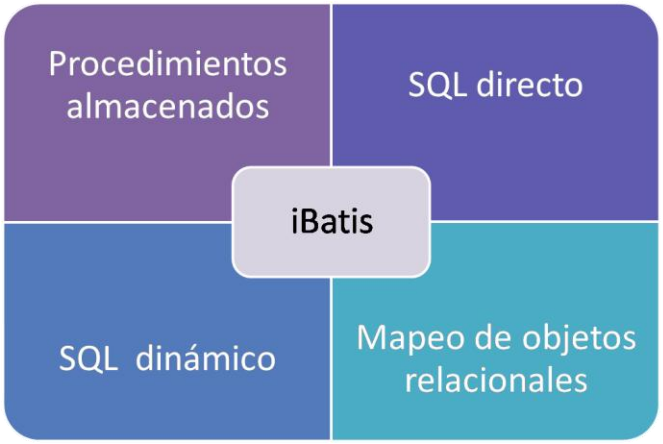
Dos de las cualidades más importantes de iBatis son la externalización y la encapsulación de las sentencias SQL. Estos conceptos proporcionan gran parte de su valor y permiten muchas de las características avanzadas que el framework consigue.

- *Externalización de SQL:* La externalización SQL que provee iBatis separa el SQL del código fuente de la aplicación, por lo tanto, mantiene limpios a ambos códigos. Esto asegura que el SQL es relativamente independiente de cualquier lenguaje o de la plataforma. iBatis hace más portátil y legible el código de las aplicaciones.
- *Encapsulación de SQL:* iBatis encapsula el SQL mediante la definición de sus entradas y salidas (es decir, su interfaz), pero a su vez oculta los detalles de la implementación al resto de la aplicación.

iBatis fue diseñado como una solución híbrida que toma las mejores ideas y atributos de los distintos métodos de acceso a una base de datos relacional y las combina entre si encontrando

un equilibrio entre estos elementos. De manera generar estos conceptos se pueden resumir como se muestra en la Figura 10.

Figura 10 Elementos que soporta iBatis SQL Maps



Cada uno de estos aspectos hace su aporte al framework:

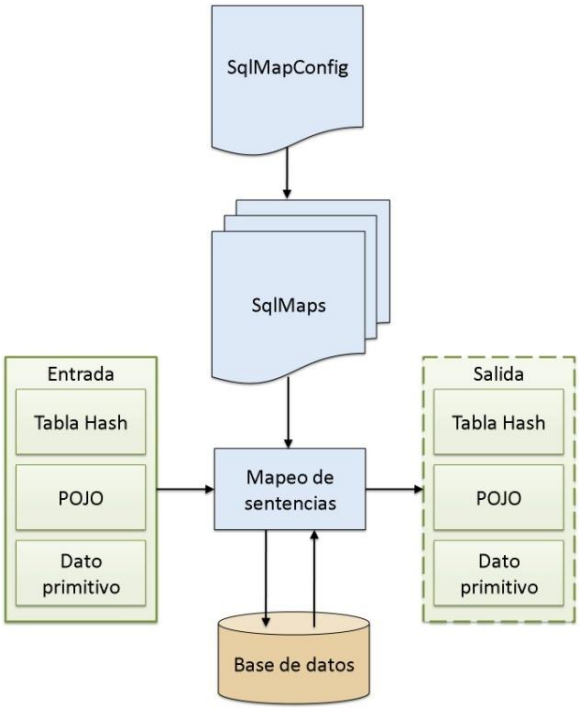
- *Procedimientos almacenados:* iBatis soporta plenamente el llamado directo de procedimientos almacenados, pero como buena práctica de diseño se recomienda mantener la lógica del negocio fuera de la base de datos, así la aplicación es más fácil de desplegar y poner a prueba, además de ser más portable.
- *SQL directo:* En lugar de pasar la lógica de negocio hacia la base de datos, el SQL se desplaza desde la base de datos para el código de la aplicación. iBatis permite escribir SQL a plenitud sin preocuparse por la concatenación de cadenas de JDBC, "ajuste" de parámetros, o "conseguir" los resultados.
- *SQL dinámico:* Resuelve algunos de los problemas del SQL directo, evitando la pre compilación. El SQL se puede manipular en tiempo de ejecución sobre la base de diferentes parámetros o funciones dinámicas de la aplicación. El SQL dinámico es actualmente el medio más popular de acceder a bases de datos relacionales. iBatis proporciona características para la construcción de consultas de forma dinámica sobre la base de parámetros sin requerir un API para ello.
- *Mapeo de objetos relacionales:* fue diseñado para simplificar la persistencia de objetos mediante la eliminación de la responsabilidad del desarrollador de manejar el código SQL. iBatis soporta muchas de las características de una herramienta ORM, tales como lazy loading, múltiples estrategias de almacenamiento en caché, la generación de código en tiempo de ejecución, la herencia y la gestión de transacciones tanto locales como distribuidas.

De manera general se pueden resumir las características que toma iBatis de los elementos analizados anteriormente:

Tabla 3 Resumen de los enfoques de iBatis SQL Maps

Enfoque	Elemento que incorpora iBatis
Procedimientos almacenados	Cada sentencia tiene una firma que le da un nombre y define sus entradas y salidas (encapsulado).
SQL directo	Permite que el código SQL sea escrito a plenitud y posibilita utilizar las variables del lenguaje directamente en los parámetros y en los resultados.
SQL dinámico	Proporciona un medio para modificar las sentencias SQL en tiempo de ejecución. Tales consultas pueden ser construidas dinámicamente para dar un resultado.
ORM	Incluye varios de los conceptos de un ORM como son: la administración avanzada de transacciones, almacenamiento en caché y carga retardada (lazy loading).

Figura 11 Flujo de iBatis SQL Maps



Es importante destacar que externalizar sentencias SQL, los resultados y las asignaciones de los parámetros en un archivo XML (Ver

Figura 11) es un simple pero potente concepto. En contraste con Hibernate, en iBatis no hay un lenguaje de consulta especial para aprender lo que significa que hay menos complejidad en su uso. (Johnson, et al., 2005)

Sin embargo, un complejo modelo de dominio con opciones de lectura y escritura no es un buen escenario para SQL Maps, ya que esto requeriría muchas definiciones para las sentencias de inserción y actualización. Para este tipo de aplicación, un ORM más sofisticado como Hibernate es mucho más recomendado siempre y cuando se tenga el completo control sobre la base de datos y sobre la aplicación en cuestión.

2.3.2 iBatis para aplicaciones empresariales

En los sistemas empresariales suelen participar no una, sino muchas bases de datos, sin tener el control sobre ellas. iBatis fue escrito en gran parte en respuesta a la necesidad de hacer frente a esta diversidad.

- No hace ninguna suposición sobre el diseño de la base de datos o el modelo de objetos. Independientemente de cómo estos dos diseños coincidan, iBatis interactúa con ambos simultáneamente. Además, no hace suposiciones sobre la arquitectura del sistema empresarial que lo utilice. Si se ha dividido la base de datos verticalmente por las reglas del negocio, u horizontalmente por la tecnología, permite trabajar de forma efectiva con los datos e integrarlos en la aplicación orientada a objetos.
- Tiene características que le permiten trabajar de forma efectiva con conjuntos de datos muy grandes. Soporta características como manipuladores de la fila que permiten el procesamiento por lotes de grandes conjuntos de registro, uno a la vez. También apoya una serie de mecanismos de obtención de los resultados, lo que le permite buscar sólo los datos absolutamente necesarios para sus necesidades inmediatas.
- Permite mapear los objetos a la base de datos de múltiples maneras. Permite tener la misma clase mapeada de múltiples formas para asegurar que cada operación está soportada de la manera más eficiente posible.

iBatis trabaja con bases de datos de cualquier tamaño o propósito. Funciona bien para pequeñas aplicaciones de bases de datos, ya que es fácil de aprender y de utilizar. Es excelente para las grandes aplicaciones empresariales, ya que no se hace ninguna hipótesis sobre el diseño de la base de datos, comportamientos, o las dependencias que podrían afectar la forma en que una aplicación utiliza la base de datos. Incluso las bases de datos que tienen

complejos diseños pueden trabajar fácilmente con iBatis. Por encima de todo, iBatis ha sido diseñado para ser lo suficientemente flexible como para adaptarse a casi cualquier situación, mientras que le ahorra tiempo al eliminar código redundante. (Begin, et al., 2007)

2.3.3 Beneficios de iBatis

Existen muchas razones para usar iBatis en casi cualquier sistema, ya que ofrece la posibilidad de inyectar beneficios arquitectónicos en las aplicaciones. Las prestaciones y características que lo hacen posible son las siguientes:

- *Simplicidad:* iBatis es ampliamente considerado como uno de los más simples frameworks de persistencia disponible en la actualidad. Esta sencillez se consigue mediante el mantenimiento de una muy sólida base sobre la cual se construye iBatis: JDBC y SQL. Es fácil de usar para los desarrolladores de Java, ya que funciona como JDBC, sólo que con mucho menos código. Es también fácil de entender para los administradores de bases de datos SQL y programadores. Sus archivos de configuración pueden ser fácilmente entendidos por casi cualquier persona con alguna experiencia en programación SQL.
- *Productividad:* La finalidad primordial de cualquier buen framework es hacer al desarrollador más productivo. El código consistente y la fácil configuración de iBatis reduce la cantidad de código en la capa de persistencia. Este ahorro se debe principalmente al hecho de que ningún código JDBC debe ser escrito.
- *Rendimiento:* En general, si se compara el código iBatis y JDBC para iterar un millón de veces en un bucle *for*, lo más probable es que el rendimiento sea a favor de JDBC. Afortunadamente, este no es el tipo de rendimiento en las cuestiones modernas de desarrollo de aplicaciones. Mucho más significativo es la forma de obtener la información de la base de datos, cuándo obtenerlos, y con qué frecuencia. Por ejemplo, utilizando una lista de datos paginados que obtiene dinámicamente los registros de la base de datos, puede aumentar significativamente el rendimiento de la solicitud debido a que no sean innecesariamente cargados miles de registros a la vez. Del mismo modo, utilizando características como la carga retardada (lazy loading) se evita la carga de datos que no necesariamente serán utilizados en el momento que se realiza la solicitud.
- *Separación de incumbencias:* iBatis gestiona todos los recursos relacionados con la persistencia como las conexiones a la base de datos, los PreparedStaments y los ResultSet. Proporciona una interfaz independiente de la base de datos y APIs que

ayudan al resto de la aplicación a permanecer independiente de cualquier recurso de persistencia. Con iBatis, siempre se trabaja con los objetos específicos del dominio de la aplicación y nunca directamente con los ResultSet. Mejora el diseño de la aplicación para garantizar el futuro mantenimiento.

- *División del trabajo:* Debido a que el código SQL es separado en gran parte del código fuente de la aplicación, los programadores de SQL pueden escribir la sentencia SQL correspondiente, sin tener que preocuparse por el código Java involucrado. iBatis posibilita separar el trabajo del programador de SQL y el programador Java manteniendo la consistencia que se requiere.
- *Portabilidad:* Debido a su diseño relativamente simple, puede ser aplicado para cualquier tipo de lenguaje o de plataforma. Hasta el momento, iBatis soporta tres plataformas de desarrollo: Java, Ruby y C#.Net. Los archivos de configuración no son totalmente compatibles a través de las plataformas en este momento, pero sus desarrolladores trabajan en lograr una homogeneidad. Esto le permite ser coherentes en el diseño de todas las aplicaciones.
- *Open source:* iBatis es libre, se encuentra bajo la licencia Apache 2.0.

2.4 Acegi Security

Acegi Security es un framework Java que proporciona mecanismos de seguridad de forma declarativa para aplicaciones empresariales basadas en Spring Framework utilizando características de AOP y de Inyección de Dependencias, de forma transparente para el desarrollador, sin necesidad de agregar código. Ofrece una solución completa de seguridad, manipulación de autenticación y autorización, tanto al nivel de peticiones en la web como al nivel de invocación de métodos. También es posible utilizarlo en aplicaciones no desarrolladas con Spring. (Minter, 2008)

El proyecto se inició a finales de 2003 como "Acegi Security" por Ben Alex, y se publicó bajo la licencia Apache en marzo de 2004. Posteriormente, Acegi se incorporó oficialmente como un sub-proyecto de Spring bajo el nombre de Spring Security y la primera versión fue liberada en abril de 2008.

2.4.1 Características principales

Acegi Security en su conjunto se pueden agrupar en tres conceptos fundamentales:

- *Autenticación:* determina si un usuario es quien dice ser.

- *Autorización*: determina lo que el usuario está autorizado a hacer si él es realmente esa persona.
- *Canal de seguridad:* se asegura de que nadie pueda escuchar la conversación.

Acegi Security soporta la mayoría de los procesos de autenticación existentes, además de permitir adicionar nuevos mecanismos. Es de código abierto y ofrece una seguridad no invasiva debido a que su implementación puede ser declarada sin hacer cambios dentro del código fuente de la aplicación mediante AOP. (Walls, et al., 2008)

Ofrece su propio enfoque de seguridad, pero también posibilita implementaciones alternativas. Proporciona un mecanismo sencillo que le permite integrarse con códigos no estándares existentes, así como una serie de implementaciones para conectarse a mecanismos de autenticación más comunes, como LDAP[20]. (Minter, 2008)

Es capaz de garantizar la seguridad de las aplicaciones sin tener que escribir ningún código directamente en la lógica de la aplicación. Gran parte de la configuración referente a la seguridad de una aplicación es totalmente independiente de los elementos que requieren asegurarse. El único componente de Acegi Security que realmente necesita saber detalles acerca del funcionamiento de la aplicación es la definición del objeto donde se asocian los recursos asegurados con las autoridades que requieren acceso a dichos recursos. (Walls, et al., 2008)

Entre las características más significativas de este framework se encuentran las siguientes:

- *Seguridad en instancias de objetos del dominio:* en muchas aplicaciones es deseable definir listas de control de acceso (Access Control List, ACL[21] por sus siglas en inglés) para instancias de un objeto de dominio. Ofrece un amplio paquete ACL con características que incluyen permisos heredados (incluido el bloqueo), respaldados por un repositorio ACL basado en JDBC, caché, puntos de extensión, y un diseño basado en interfaces.
- *Protege peticiones HTTP:* a través de filtros web garantiza el acceso a las distintas URLs y mediante rutas de Apache Ant o de expresiones regulares realiza la autorización a los recursos solicitados.

---

[20] LDAP acrónimo de Lightweight Directory Access Protocol, (Protocolo Ligero de Acceso a Directorios).

[21] ACL acrónimo de Access Control List (Lista de Control de Acceso)

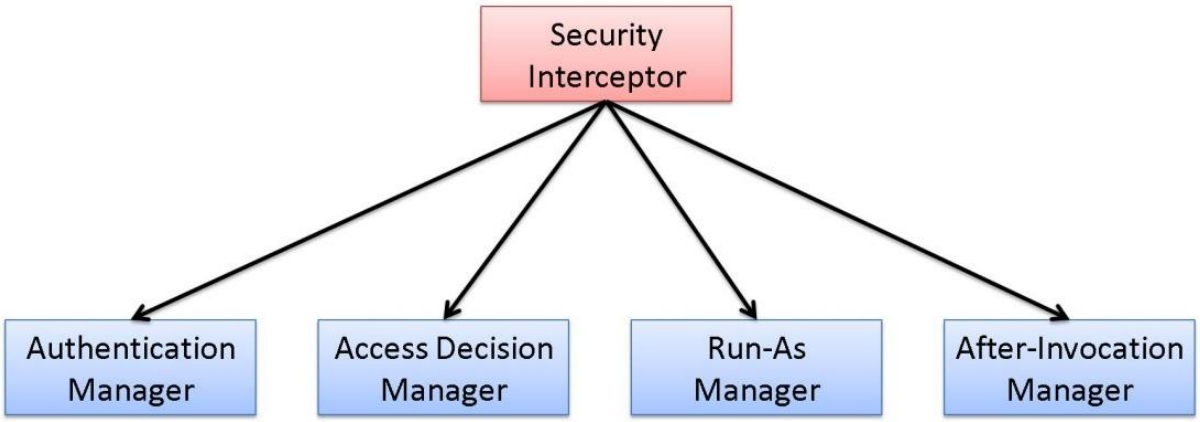
- *Canal de seguridad:* puede redirigir automáticamente las peticiones a través de un canal de transporte apropiado. Una característica común de un canal de seguridad es garantizar que las páginas seguras sólo estén disponibles a través de HTTPS, y las públicas mediante HTTP.
- *Varias formas de autenticación:* incluye la capacidad de recuperar el nombre de usuario y la definición de los privilegios de un archivo XML, fuente de datos JDBC o desde un fichero de propiedades.
- *Almacenamiento en caché:* se integra con la fábrica de EHCache de Spring. Esta flexibilidad significa que la base de datos (u otro tipo de repositorio de autenticación) no se encuesta varias veces solicitando la información de autenticación.
- *Sesiones concurrentes:* posibilita configurar el número de inicios de sesión simultáneos que puede realizar un usuario.
- *Soporte para remoting:* se integra con los protocolos remotos estándares de Spring, procesando automáticamente la autenticación básica HTTP que presentan las cabeceras de los mensajes.
- *Pruebas de unidad:* brinda mecanismos de alto nivel para realizar pruebas de unidad a los objetos de negocio asegurados. Se puede cambiar la identidad autenticada y sus autoridades asociadas directamente dentro del método de prueba.

2.4.2 Componentes de Acegi Security

La arquitectura de Acegi Security está fuertemente basada en interfaces y en patrones de diseño, proporcionando las implementaciones más comúnmente utilizadas y numerosos puntos de extensión donde nuevas funcionalidades pueden ser añadidas.

Acegi Security emplea cinco componentes fundamentales para mantener la seguridad, como se muestra en la Figura 12. (Minter, 2008)

Figura 12 Componentes generales de Acegi Security



Security interceptor (Interceptor de seguridad)

La implementación de un interceptor de seguridad dependerá de los recursos que se están asegurando. Si es una dirección URL en una aplicación web, el interceptor de seguridad será implementado como un filtro de un servlet. Si se está asegurando la invocación de un método, los aspectos se utilizarán para hacer cumplir la seguridad.

Un interceptor de seguridad solo intercepta el acceso a los recursos para hacer cumplir la seguridad sin aplicar ninguna regla específica. Esta responsabilidad es delegada a los distintos administradores.

Authentication manager (Administrador de autenticación)

Es el primer punto durante la autenticación y autorización y es el responsable de determinar quién es la entidad que está haciendo la petición. Para ello utiliza el objeto principal (normalmente un nombre de usuario), y sus credenciales (normalmente una contraseña).

El objeto principal define quién es el que realiza la petición y las credenciales son las pruebas que corroboran su identidad. Si las credenciales son lo suficientemente buenas como para convencer al gestor de autenticación que el objeto principal es quién dice ser, entonces Acegi Security sabrá con quién está tratando.

El administrador de autenticación es un componente extensible mediante interfaces. Esto hace posible su utilización con prácticamente cualquier mecanismo de autenticación. Acegi Security proporciona un conjunto de flexibles administradores de autenticación que cubren las estrategias más comunes de autenticación.

Access decision manager (Administrador de decisión de acceso)

Un administrador de decisión de acceso es el segundo punto durante la ejecución de la seguridad. Es el elemento que realiza la autorización, decide si permite el acceso considerando la información de autenticación y los atributos de seguridad que se han asociado con los recursos asegurados. El administrador de decisión de acceso también es extensible mediante interfaces.

Run-as manager (Administrador de ejecución)

Después de la autenticación y la autorización, puede haber más restricciones de seguridad determinadas por las reglas del negocio. Por ejemplo, puede que se otorgara el derecho a ver una página web, pero los objetos que se utilizan para crear estas páginas pueden tener requisitos de seguridad diferentes a los de la página web. Un administrador de ejecución puede ser usado para sustituir a la autenticación efectuada por un certificado de autenticidad que le permita tener acceso a otros objetos en la aplicación (una fábrica de objetos, un pool de conexiones, una fuente de datos, etc.).

No todas las aplicaciones tienen la necesidad de esta sustitución de identidad. Por lo tanto, los administradores de ejecución son un componente de seguridad opcional y no es necesario en muchas aplicaciones aseguradas por Acegi Security.

After-invocation manager (Administrador de post-invocación)

El administrador de post-invocación es un poco diferente de los otros componentes. Mientras que los anteriores realizan algún tipo de ejecución de seguridad antes de que un recurso asegurado sea accedido, este administrador interviene luego de que se efectúa dicho acceso. El administrador de post-invocación se asegura de que se cuente con la autorización para ver los datos que retorna un recurso asegurado.

Si un administrador de post-invocación supervisa a un bean de la capa de servicio, este dará la oportunidad de revisar el valor retornado por el método supervisado. Se puede entonces tomar una decisión en cuanto a si el usuario está autorizado a ver los objetos devueltos. Este administrador también tiene la opción de modificar el valor devuelto para asegurarse de que el usuario sólo puede acceder a ciertas propiedades de los objetos retornados.

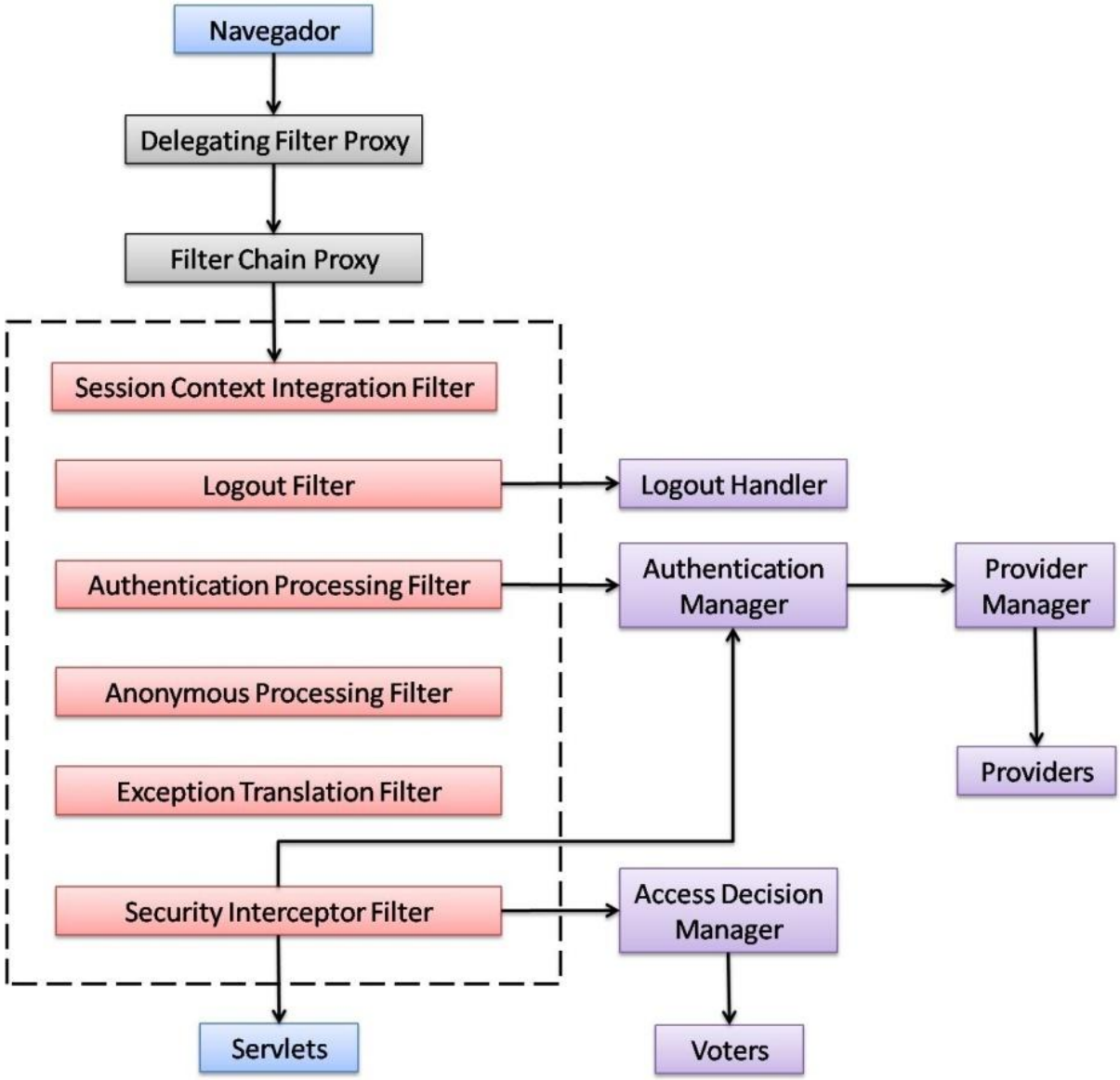
Al igual que el administrador de ejecución, no todas las aplicaciones necesitan de un administrador de post-invocación. Sólo se empleará si el esquema de seguridad de la aplicación exige que se limite el acceso en el nivel de dominio.

2.4.3 Seguridad de aplicaciones web

Al asegurar las aplicaciones web, Acegi Security utiliza filtros que interceptan las peticiones para realizar la autenticación y hacer valer la seguridad. Emplea un mecanismo único para declarar filtros que le permiten inyectar sus dependencias utilizando la inyección de dependencias de Spring. (Walls, et al., 2008)

La Figura 13 muestra los componentes típicos de una aplicación web asegurada con Acegi Security. (Minter, 2008)

Figura 13 Elementos de Acegi Security para asegurar una petición web



Estas clases se dividen claramente en los filtros (necesario para la gestión de la seguridad web) y en los componentes que proporcionan el acceso a los datos de autenticación y autorización.

2.4.4 Asegurando la invocación de métodos

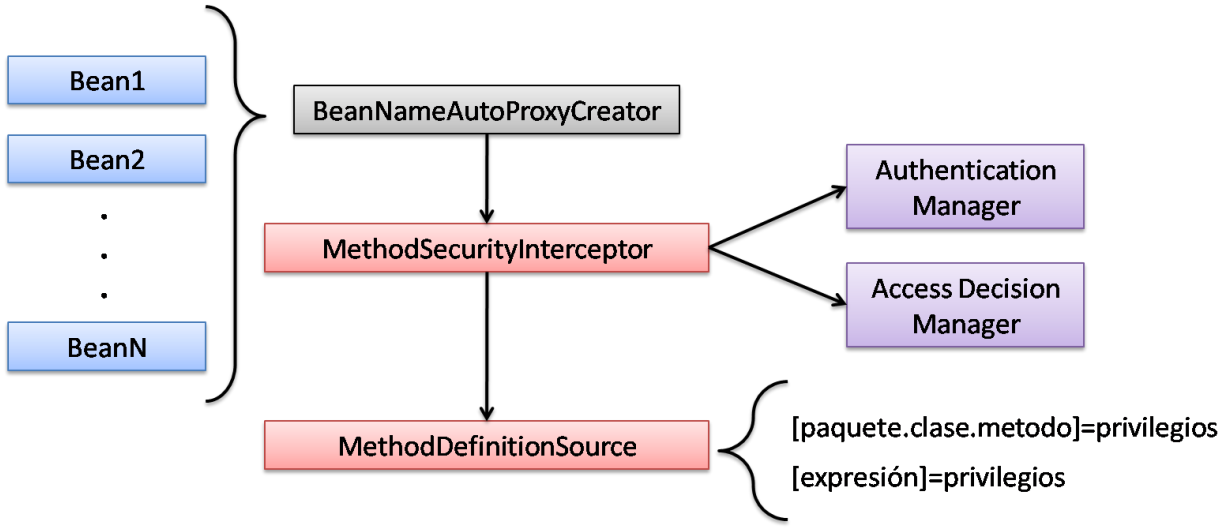
Acegi Security también puede aplicar la seguridad a un nivel inferior al asegurar la invocación de métodos.

Cuando se aseguran métodos, Spring Security utiliza proxy de objetos (representantes de objetos) mediante Spring AOP, la aplicación de los aspectos garantizan que el usuario tenga autoridad para invocar los métodos asegurados. (Walls, et al., 2008)

Uno de los mecanismos de Acegi Security que posibilita este nivel de seguridad (Ver Figura 14) es configurando un proxy AOP el cual intercepta las llamadas a métodos y le pasa el control a un interceptor, utilizando la clase `BeanNameAutoProxyCreator`.

A una instancia de esta clase se le asignan los nombres de los beans (o un patrón de nombrado) que se quieren asegurar auxiliándose de un `MethodSecurityInterceptor` que a su vez emplea un objeto de la clase `MethodDefinitionSource`, que permite asociar patrones de métodos con los privilegios necesarios para su invocación.

Figura 14 Elementos de Acegi Security para asegurar la invocación de métodos



También se configuran el `AuthenticationManager` y el `AccessDecisionManager` presentes durante el flujo de seguridad de una petición web, mediante los cuales se determina si el usuario tiene privilegios suficientes para realizar la operación solicitada.

2.4.5 Beneficios de Acegi Security

Acegi Security se ha convertido en la herramienta para asegurar los sistemas basados en Spring Framework. Entre los beneficios que proporciona se pueden mencionar los siguientes:

- *Reutiliza la experiencia de Spring:* utiliza los contextos de aplicación para todas las configuraciones logrando un desarrollo rápido y agradable.
- *Mantiene los objetos libres de código de seguridad:* muchas aplicaciones necesitan para proteger datos basarse en cualquier combinación de parámetros (usuario, hora del día, roles, el método invocado, etc.). Este paquete ofrece esta flexibilidad sin añadir código de seguridad a los objetos de negocio de la aplicación.
- *Arquitectura extensible:* Cada aspecto crítico del paquete está modelado usando alta cohesión, bajo acoplamiento, y principios de diseño orientado a interfaces. Es fácilmente reemplazable, adaptable, y con puntos de extensión que posibilitan la ampliación del diseño original.
- *Fácil integración con las bases de datos existentes:* no se restringe el esquema de autenticación a utilizar así como los datos que intervienen en el proceso de autenticación. Proporcionar facilidades para utilizar un DAO personalizado si se desea.

2.5 Direct Web Remoting (DWR)

DWR es un API RPC que permite introducir, en las aplicaciones J2EE, la filosofía AJAX de una forma sencilla. Es un producto open source bajo la licencia de Apache 2.0 que pretende reducir la cantidad de tiempo que se tarda en aplicar Ajax en las aplicaciones Java automatizando tareas comunes de Ajax y la reducción de la cantidad de código escrito por el desarrollador. (Schutta, et al., 2006)

2.5.1 Características

Esta biblioteca permite realizar llamadas remotas desde código Javascript (ejecutándose en un navegador), a objetos Java (POJOs) del servidor y viceversa.

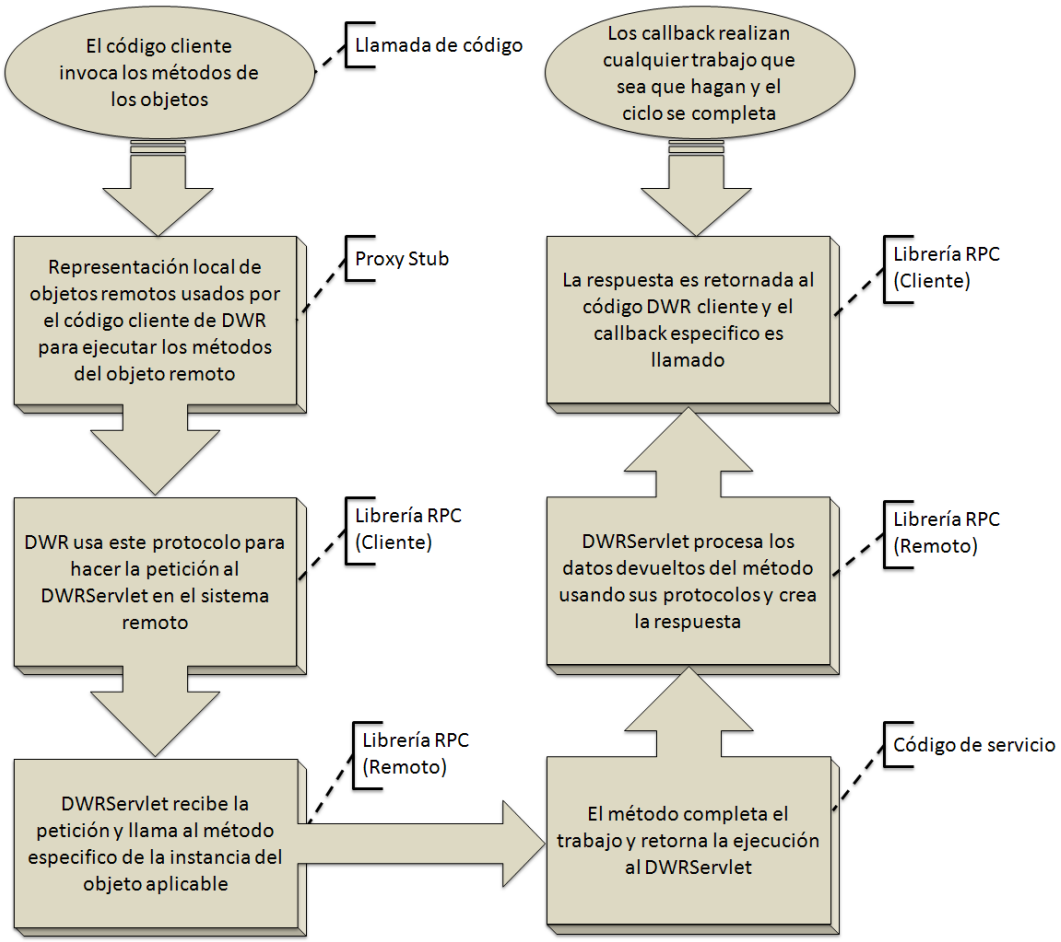
En el modelo DWR, el *"sistema"* llamado es la ejecución de Javascript en el navegador del usuario y el *"sistema"* que se llama es la clase Java que se ejecuta en el servidor.

Al igual que otras formas de RPC, el código que se escribe en Javascript es muy similar al código que se escribe en el servidor para ejecutar la misma operación. Existen, por supuesto,

algunas diferencias como era de esperar dadas las diferencias intrínsecas entre Javascript y Java, pero en general es muy similar. (Zammetti, 2008)

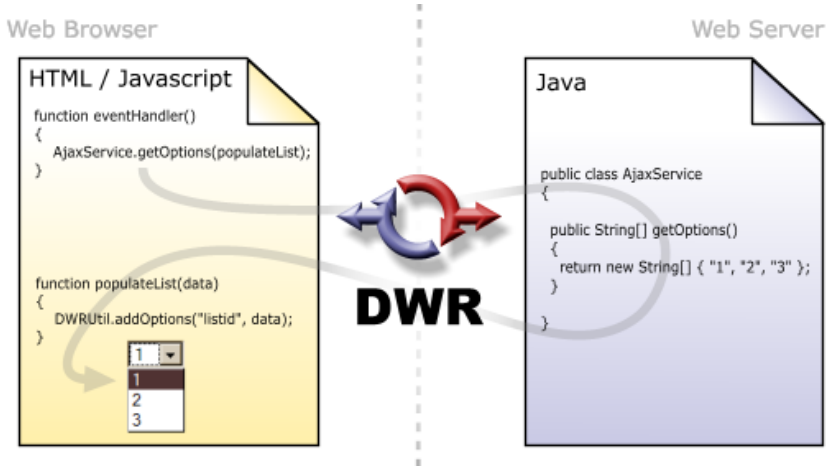
La Figura 15 muestra el flujo básico por el que transita una petición DWR desde su invocación en el cliente hasta la respuesta en el lado del servidor.

Figura 15 Flujo básico de una petición DWR



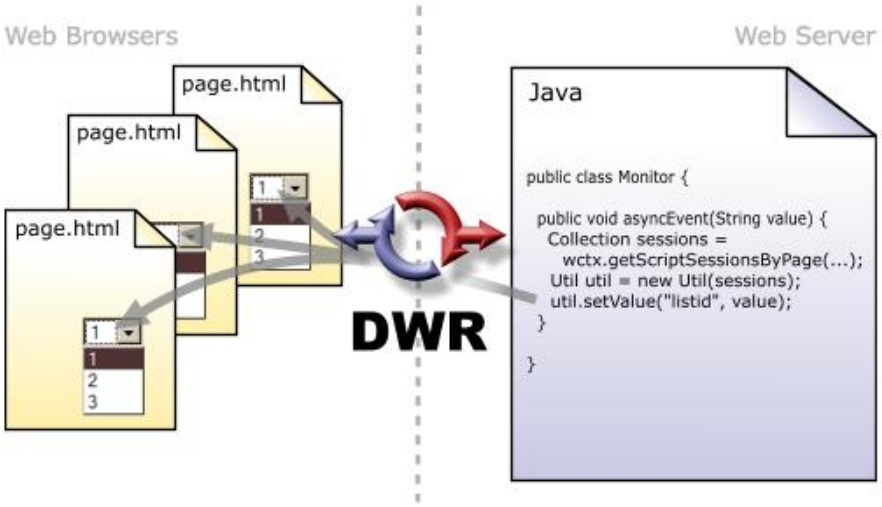
En la Figura 16 se muestra cómo DWR puede alterar el contenido de una lista de selección como resultado de algún evento Javascript.

Figura 16 Flujo DWR para una petición del cliente



Reverse Ajax[22] permite a través de código Java corriendo en el servidor enviar Javascript hacia ciertas páginas activas en los clientes, generadas ya sea manualmente o utilizando a una API de Java (Ver Figura 17).

Figura 17 Flujo DWR para una notificación del servidor a páginas activas



DWR trabaja generando dinámicamente Javascript basado en clases Java utilizando Ajax para dar la impresión de que la ejecución del código ocurre en el navegador, pero en realidad el servidor ejecuta el código y DWR coloca los datos en un sentido y otro.

Este método de funciones remotas de Java para Javascript les da a los usuarios de DWR una mejor percepción que los mecanismos convencionales RPC, como RMI o SOAP, con el

[22] Reverse Ajax (Ajax en reversa) se le denomina en DWR al proceso de actualizar el contenido de la página desde el servidor, sin ser consecuencia de la llamada de un método remoto desde el Javascript cliente.

beneficio que corre sobre la web sin requerir pluggings para los navegadores. (Apache Foundation)

Otro de los puntos clave acerca de DWR es que es una herramienta no intrusiva. No obliga a tener requerimientos en la arquitectura en términos de cómo organizar el código en el lado del servidor o del cliente. En términos del código del servidor, DWR da la libertad de usar Spring Beans, EJB, POJOs, servlets, o prácticamente cualquier otro tipo de clase Java que se pueda imaginar. (Zammetti, 2008)

DWR se compone de dos partes principales: una parte que se ejecuta en el lado cliente (en este caso, un navegador web) y otra parte que se ejecuta en el servidor (en este caso, un contenedor de Servlets).

2.5.2 DWR en el servidor

En el lado del servidor, DWR proporciona servicios para la invocación de métodos remotos y la traducción de tipos primitivos, de forma muy similar a otras tecnologías RPC.

Invocación Remota (Directa) de Métodos

DWR proporciona un envoltorio (wrapper) para permitir la invocación de métodos de objetos Java (POJOs) desde el cliente. El sistema de RPC implementado por DWR se basa en un Servlet[23], el cual, utilizando la información que le llega a través de la petición HTTP (HTTP Request), se encarga de instanciar los objetos necesarios y de realizar la invocación del método solicitado, pasándole los parámetros enviados desde el cliente. Todas estas operaciones se realizan utilizando el API de reflexión de Java.

Traducción de Tipos

DWR traduce automáticamente los parámetros y valores de retorno entre Javascript y Java. Puede realizar, traducciones de varios tipos: tipos básicos (primitivos como `int` y `boolean`, sus correspondientes clases `Integer` y `Boolean`, asi como `String` y `Date`), colecciones, arrays y beans. DWR siempre intenta traducir tipos Java al tipo Javascript más parecido. Por ejemplo, las colecciones se pueden traducir en arrays y los beans en arrays asociativos, siendo los nombres de las propiedades del bean los índices del array. DWR proporciona una gran flexibilidad, ya que no se está obligado a pasar únicamente tipos de datos simples entre el navegador y el servidor. (Schutta, et al., 2006)

---

[23] El Servlet que se refiere es la clase uk.ltd.getahead.dwr.DWRServlet que se encuentra en el API de DWR.

2.5.3 DWR en el cliente

La parte de DWR que se ejecuta en el navegador, tiene dos funciones: por un lado, sirve como apoyo para la realización de llamadas a los objetos del servidor, y por otro, proporciona un conjunto de funciones que facilitan la operación sobre el código DHTML de la página web.

Stub[24] del lado cliente

DWR proporciona para cada uno de los métodos exportados por el servidor, una función Javascript que actúa como stub del cliente. Los stubs de cliente ocultan al desarrollador las complejidades de la comunicación cliente-servidor (especialmente, el manejo del objeto XMLHttpRequest). Las funciones Javascript, que hacen referencia a los métodos de una misma clase, se agrupan en un mismo fichero .js. Cada uno de estos ficheros se puede obtener siempre de la URL: <servletdwrpath>/interface/<nombreclase>.js. No es necesario escribir los ficheros stub (.js), el servlet DWRServlet se encarga de generarlos *"al vuelo"* utilizando la configuración que obtiene del fichero dwr.xml.

Javascript asíncrono

La filosofía AJAX supone que la actualización de la vista se realiza de forma asíncrona, por lo tanto las llamadas a métodos remotos deben cumplir este principio, es decir, no se puede llamar directamente a un método y obtener un resultado. DWR proporciona al usuario este mecanismo estableciendo que la invocación de cada método remoto desde Javascript reciba como parámetro una función *callback* que será la función que se ejecute cuando haya finalizado la llamada asincrónica.

Utilidades para actualizar dinámicamente la página HTML

Además de facilitar la comunicación con el servidor desde clientes ligeros, DWR incluye una biblioteca de funciones que permiten manipular código DHTML. Este conjunto de funciones facilitarán la actualización del contenido de la página web activa (que está viendo el usuario), de forma dinámica, utilizando Javascript.

Estas funciones se agrupan en un fichero de scripting, que se puede obtener de la URL: <servletdwr>/util.js. Al igual que ocurre con otros ficheros JS servidos por el Servlet DWR, el

[24] Stub se refiere a las implementaciones que ejecutan los métodos remotos haciendo transparente esta lógica para el programador.

fichero util.js se genera de forma dinámica al ser solicitado (no es necesario incluir este fichero en la aplicación web).

Las funciones de utilidad van a permitir la actualización del árbol DOM[25] de la página web activa, pero ocultando, en la medida de lo posible, las complejidades del modelo de objetos DOM.

Algunas de las operaciones más comunes van a permitir:

- Establecer u obtener un valor (o conjunto de valores) para un componente. Este conjunto de operaciones es aplicable a: campos de texto, radiobuttons, divs, etc.
- Añadir / Eliminar filas de una tabla y opciones de una lista.

[25] DOM acrónimo de Document Object Model, una forma de referirse a elementos XML o HTML como objetos.

3

Organización arquitectónica

- Aspectos generales
- Separación lógica en capas
- Seguridad
- Tratamiento de excepciones
- Modelo de despliegue
- Estilo de desarrollo

CAPÍTULO 3.- ORGANIZACIÓN ARQUITECTÓNICA

En este capítulo se analiza la estructura del Módulo de Administración y Control de Recursos, en términos de principios de diseño e implementación, distribución física y lógica, separación en capas y la comunicación entre ellas, la seguridad y el tratamiento de errores.

Se define además un flujo de trabajo, el cual permite guiar, paso a paso, el desarrollo de la aplicación y a su vez se establece un estilo de desarrollo que estandariza las pautas de codificación y la organización de los ficheros en toda la aplicación.

3.1 Aspectos generales

En el desarrollo de aplicaciones empresariales, donde los sistemas poseen un dominio complejo, con lógica de negocio compleja y muchas reglas de negocio, las cuales varían con el tiempo, y van modificando a las actuales, y nutriéndose con otras nuevas, la idea central es modelar una arquitectura que sea capaz de soportar todas estas características con el mínimo de impacto en el diseño ante posibles modificaciones en los requerimientos.

Según (Johnson, 2003) un buen diseño de aplicaciones J2EE debe cumplir una conjunto de requisitos, que aunque el autor los especifica para soluciones basadas en esta plataforma también son válidos para otros sistemas empresariales.

Estas características son las siguientes:

- Ser robusto.
- Ser eficiente y escalable.
- Tomar ventaja de los principios de diseño orientado a objetos.
- Evitar la complejidad innecesaria.
- Ser mantenible y extensible.
- Ser fácil de probar.
- Promover la reutilización.

El Módulo de Administración y Control de Recursos es un sistema que necesita estar disponible 24 horas los 7 días de la semana y que aspira a estandarizar este proceso en los Centros de Gestión de Emergencia 171 permitiendo agilizar la atención a las emergencias.

A su vez es necesario definir un sistema genérico que sea capaz de manejar la diversidad de recursos que abarcan estos centros, permitiendo definir nuevos recursos que no existían al implantarse el sistema.

Por otra parte las decisiones de diseño de la arquitectura deben estar enfocadas en permitirle a la aplicación adaptarse a los cambios de los requerimientos, con el mínimo impacto en el equipo de desarrollo y en el sistema en general.

3.1.1 Criterios de diseño

El Módulo de Administración y Control de Recursos es una aplicación web por lo que implica utilizar una arquitectura cliente-servidor (Ver epígrafe 1.3.1) mientras que la utilización de Spring MVC (Ver epígrafe 2.2.2) involucra aplicar este patrón arquitectónico (Ver epígrafe 1.3.3).

La separación de responsabilidades o incumbencias entre los distintos elementos del sistema es un aspecto fundamental en el diseño. Esta premisa supone la base de una arquitectura en capas (Ver epígrafe 1.3.2) que divide la responsabilidad en distintos niveles y que interaccionan unos con otros a través de sus interfaces. Aplicados a las aplicaciones web, el modelo más común es el de aplicaciones 3 capas. En el epígrafe 3.2 se detalla la separación lógica de las capas.

La propia utilización de la plataforma J2EE (Ver epígrafe 2.1) involucra un diseño orientado a objetos y a su vez los principios que guían este paradigma de programación.

El empleo y aplicación de patrones de diseño (Gamma, et al., 1994) facilita el entendimiento del código y, por tanto, reduce considerablemente el coste de mantenimiento, dado que además de aportar soluciones eficientes para problemas comunes, son muy interesantes como medio de entendimiento entre diseñadores e implementadores.

3.1.2 Infraestructura base de desarrollo

Es esencial para un sistema informático tener una infraestructura sólida que permita al código de las aplicaciones concentrarse en abordar los problemas de la lógica del negocio sin tener que preocuparse por otros procesos que no constituyen su línea central. Procesos como la seguridad, la gestión de objetos, transacciones, tratamiento de excepciones son elementos que si bien son de vital importancia se debería separar del desarrollo principal de las aplicaciones.

El uso de una infraestructura sólida puede ofrecer mejores aplicaciones con un tiempo de desarrollo mucho más rápido. (Johnson, 2003). Entre las ventajas se pueden citar las siguientes:

- Permite centrarse en la implementación de la lógica de negocio y otras funcionalidades de la aplicación con un mínimo de distracción. Reduce el tiempo de producción mediante la reducción de esfuerzo de desarrollo, y reduce los costos a lo largo del ciclo de vida del proyecto haciendo el código de las aplicaciones más mantenible.
- Posibilita la separación de la configuración del código fuente.
- Facilita el uso del diseño orientado a objetos, eliminando la necesidad de encargarse de tareas transversales comunes.
- Elimina la duplicación de código, solucionando cada problema una sola vez.
- Garantizar la correcta gestión de errores.
- Facilita la internacionalización, si es necesario su utilización.
- Aumenta la productividad sin comprometer principios de la arquitectura.
- Logra la coherencia entre las aplicaciones dentro de una organización.
- Garantiza que las aplicaciones sean fáciles de probar.

Para el desarrollo del sistema se escogió la plataforma J2EE que si bien está respaldada por un sin número de prestaciones (Ver epígrafe 2.1) se suma el hecho de que desde el punto de vista tecnológico se cuenta con limitaciones legales por parte del cliente fundamentadas en el Artículo 1 del Decreto Nº 3.390 de fecha 23 de diciembre de 2004, que dispone que la Administración Pública Nacional empleará prioritariamente Software Libre desarrollado con Estándares Abiertos, en sus sistemas, proyectos y servicios.

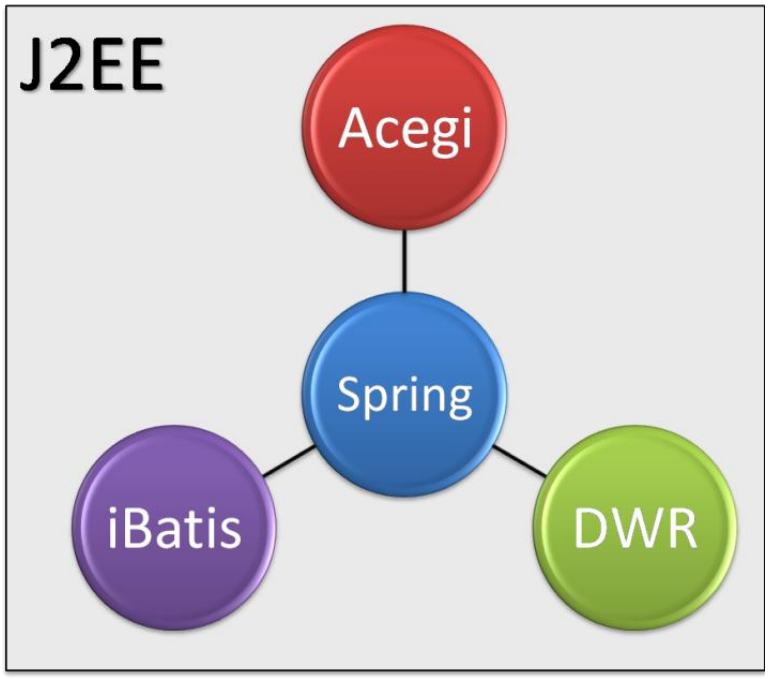
Teniendo en cuenta las facilidades que brinda el uso de frameworks para resolver problemas generales y específicos dentro de una aplicación, se realizó la selección que se refleja en la Figura 18.

Las diferentes tecnologías y herramientas que son usadas en el desarrollo de la arquitectura son las siguientes:

- JDK 1.6
- Spring Framework 2.0.4
- iBatis 2.3.0.677
- Acegi Security 1.0.4

- DWR 2.0.1
- Oracle 10g
- Apache Tomcat 5.5.17
- Eclipse 3.3
- WTP 2.0
- SpringIDE 2.0
- Subclipse 1.4.2

Figura 18 Plataforma y frameworks de desarrollo



3.1.3 Estructura del módulo

La aplicación está estructurada en 3 capas lógicas: Presentación, Negocio y Acceso a Datos. Cada una de estas capas con sus funcionalidades y componentes bien definidos. (Analizar Figura 19).

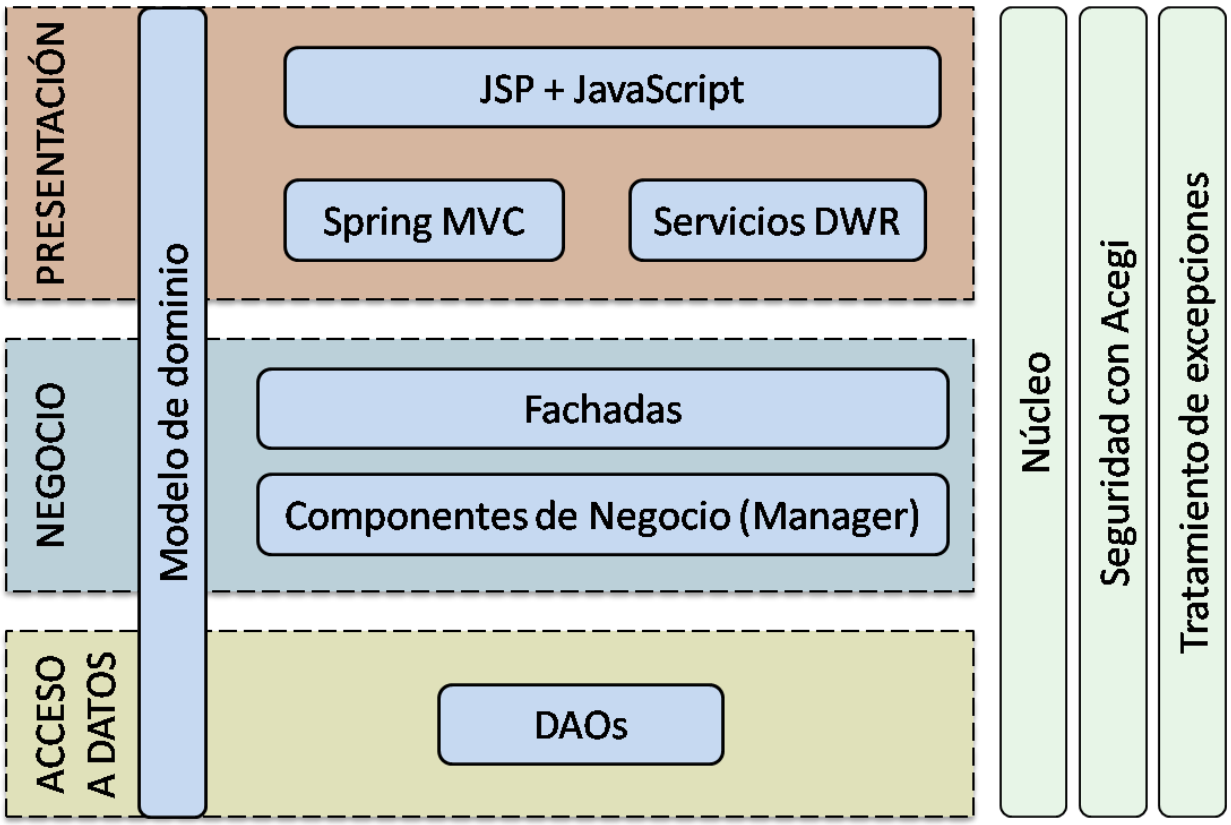
- *Presentación:* Como su nombre indica, se limita a gestionar todos aquellos aspectos relacionados con la interacción del usuario y el sistema, es decir, la lógica de presentación. Entre los componentes que están en esta capa se encuentran fundamentalmente los pertenecientes al modelo Spring MVC descrito en el epígrafe 2.2.2.

- *Negocio:* Es la capa encargada de implementar el conjunto de reglas del negocio encapsulando la lógica de las mismas. Se auxilia de la capa de acceso a datos para obtener la información necesaria que será mostrada por la capa de presentación.
- *Acceso a datos:* Esta capa es la encargada de ofrecer acceso a los datos que se encuentran dentro de los límites del sistema, posibilitando la persistencia de las entidades que se manejan en el negocio mediante interfaces genéricas.

Existen además un conjunto de funcionalidades que son utilizadas por más de una capa que se pueden definir como incumbencias transversales y que se resumen en:

- Componentes para la seguridad. (Detallado en el epígrafe 3.3)
- Tratamiento de excepciones. (Ver epígrafe 3.4)
- Núcleo. Se encuentras las clases que dan soporte a varios elementos comunes como la comunicación entre capas, configuración y otras utilidades necesarias.

Figura 19 Estructura lógica en capas



Está separación lógica no tiene que corresponder precisamente con la distribución física de la aplicación. Este diseño en tres capas puede ser desplegado en varios nodos, teniendo en cuenta el costo de desplegar el sistema en un ambiente distribuido.

3.2 Separación lógica en capas

A la hora de plantear el diseño de una aplicación web, el primer paso es conseguir separar conceptualmente las tareas que el sistema debe desempeñar entre las distintas capas lógicas y sobre la base de la naturaleza de tales tareas. Se parte de la separación inicial en tres capas, diferenciando qué proceso responde a tareas de presentación, cuál a negocio y cuál a acceso a datos. En caso de identificar algún proceso lógico que abarque responsabilidades adjudicadas a dos o más capas distintas, es probable que dicho proceso deba ser explotado en subprocesos de forma iterativa, hasta alcanzar el punto en el que no exista ninguno que abarque más de una capa lógica.

3.2.1 Capa de Presentación

La capa de presentación es donde los usuarios y el sistema interactúan. Es responsable de generar la interfaz de usuario (UI por sus siglas en inglés) del sistema. La UI es la parte más visible del sistema, que termina siendo la más frecuentemente cambiada. Esta es quizás la principal razón para separar la presentación de las demás capas de código en una arquitectura de software multicapas. (McGovern, et al., 2003)

La presentación necesita soportar algunas de las características siguientes:

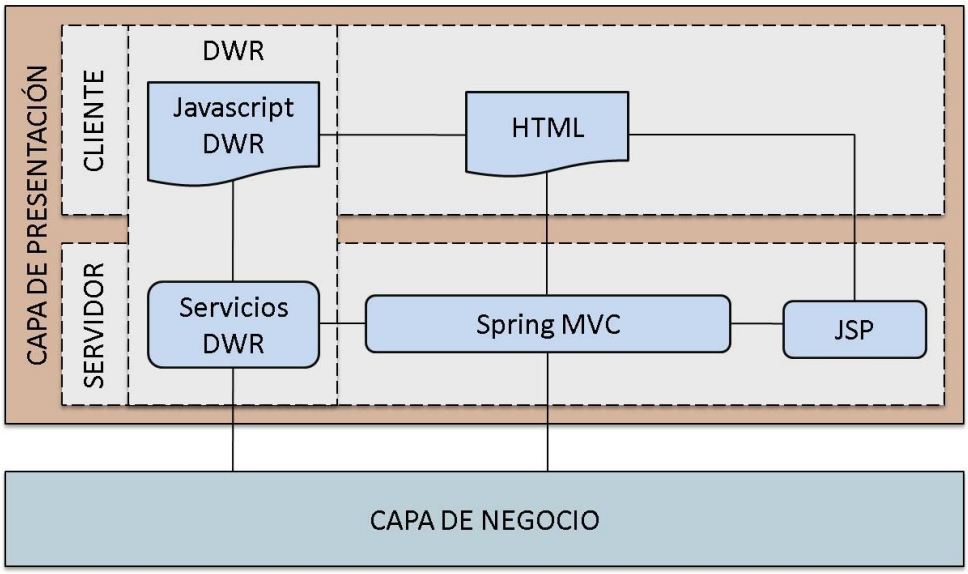
- Internacionalización y localización.
- Formateo de los datos de salida.
- Validación de los datos de entrada, en cuanto a formatos, longitudes máximas, etc.
- Personalización.
- Soporte de múltiples dispositivos y puntos de acceso.
- Soporte para procesos de negocio locales y presentación de los datos.
- Soporte de múltiples tecnologías.
- La exposición potencial de la aplicación a otros sistemas como servicios.

Esta es la capa que interactúa con el usuario final y a su vez le proporciona las opciones para interactuar con el sistema. Es la encargada de presentar la información y del empaquetado de

los datos usando los servicios expuestos por la capa de negocio. Esta capa posee sólo la lógica necesaria para juntar los datos e invocar un servicio y mostrar su resultado.

En esta arquitectura, la presentación ha sido separada en dos capas la primaria y la secundaria, correspondientes a las actividades relacionadas con el cliente y el servidor. (Ver Figura 20)

Figura 20 Estructura de la Capa de Presentación



Del lado del cliente la capa de presentación se compone por los elementos que presentan una interfaz a los usuarios para interactuar con el sistema. Para mostrar la información se utilizaron los componentes de UI de HTML proporcionados por las páginas JSP y pequeñas validaciones con Javascript asociadas a la validación de los datos proporcionados por el usuario. A esto se unen los códigos Javascript necesarios para la invocación de los métodos con DWR, así como el procesamiento de la respuesta de los mismos y los cambios que impliquen en la presentación. Se utiliza además todas las funcionalidades que brinda este framework para el trabajo con el DOM.

Del lado del servidor, la capa de presentación consiste en los componentes que preparan la UI que se muestra a los usuarios. La misma será gestionada usando Spring MVC, que a su vez posibilita la validación de los datos en el servidor, independientemente de la validación hecha en el cliente. También se encuentran los beans correspondientes a los objetos publicados remotamente utilizando DWR y que son los que dan respuesta a las peticiones realizadas por el Javascript cliente.

3.2.2 Capa de Negocio

Esta capa contiene todas las clases que modelan el dominio de la aplicación, sus entidades, sus relaciones y las reglas de negocios que se deben cumplir. Los componentes de esta capa implementan la funcionalidad básica del sistema, y encapsulan toda la lógica relevante del negocio. Es donde se implementan todas aquellas reglas obtenidas a partir del análisis funcional del proyecto.

Así mismo, debe ser completamente independiente de cualquiera de los aspectos relacionados con la presentación de la misma.

Por otro lado, la capa de negocio ha de ser también completamente independiente de los mecanismos de persistencia empleados en la capa de acceso a datos. Cuando la capa de negocio requiera recuperar o persistir cualquier conjunto de información, lo hará siempre apoyándose en los servicios que ofrezca la capa de acceso a datos para ello. De esta forma, la sustitución del motor de persistencia no afecta en lo más mínimo a esta parte del sistema. Debería poder reemplazarse el gestor de bases de datos por un conjunto de ficheros de texto sin necesitar tomar ni una línea de código de presentación o negocio.

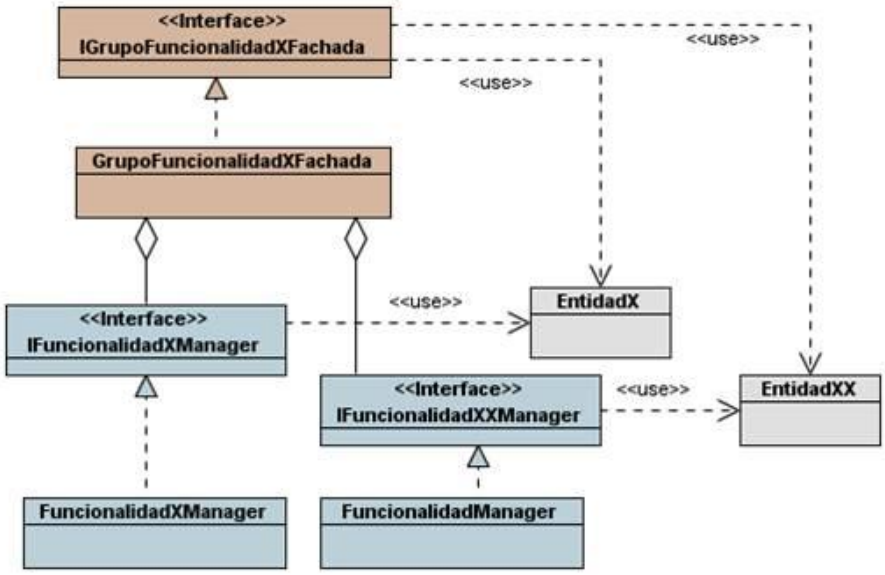
La capa de negocio se compone por tres partes fundamentales (Ver Figura 21):

- Fachada de la aplicación. Es un componente que se utiliza para combinar múltiples operaciones del negocio en una operación basada en un mensaje simple. Es útil cuando se ubican los componentes de la capa de presentación en un nivel físico separado de los componentes de la capa de negocio, permitiendo optimizar los métodos de comunicación que conectan a ambas capas. Para lograr un bajo acoplamiento entre la capa de negocio y la capa de presentación, cada fachada esta diseñada como una interfaz que define todas las funcionalidades correspondiente a un caso de uso y una clase concreta que implementa dichas funcionalidades, haciendo uso de la lógica de negocio contenida en los Managers.
- Componentes del negocio (Manager). Estos componentes implementan la lógica de negocio de la aplicación. Independientemente de si un proceso de negocio consiste en un solo paso o un flujo de trabajo organizado, la aplicación requiere componentes que implementen las reglas del negocio y que realicen las tareas del negocio. Las reglas describen cómo los datos deben ser manipulados y transformados, pudiendo ser simples o complejos, dependiendo del propio negocio. Los Managers incluyen las

operaciones CRUD[26] y otras operaciones más específicas según sea el caso, además de interactuar con la capa de acceso a datos.

- Entidades del negocio. Estas entidades se utilizan para pasar los datos entre los componentes y encapsulas los objetos que contiene la información necesaria para hacer cumplir las reglas del negocio.

Figura 21 Estructura de la Capa de Negocio



3.2.3 Capa de Acceso a Datos

El acceso a los datos es vital para las aplicaciones empresariales. A menudo, el rendimiento de la fuente de datos y las estrategias utilizadas para acceder a ella dictarán el rendimiento y la escalabilidad de una aplicación J2EE. Una de las principales tareas de un arquitecto J2EE es lograr una interfaz limpia y eficiente entre los objetos de negocio y la fuente de datos.

La capa de acceso a datos es una abstracción que facilita y estandariza el acceso a los datos de la aplicación, encapsulando la lógica inherente a la tecnología usada para la gestión de los datos.

En (Johnson, 2003) se define que las características principales que la capa de acceso a datos debe garantizar son las siguientes:

[26] CRUD: Create, Retrieve, Update, Delete. Por sus nombres en inglés para referirse a las operaciones de creación, actualización, eliminación y obtención de datos.

- Garantizar la integridad de los datos. Principalmente desde la propia base de datos y no a través del código de la capa de datos.
- La estrategia de acceso a datos no debería determinar cómo implementar la lógica de negocio.
- Debe ser posible cambiar una estrategia de persistencia en una aplicación sin reescribir su lógica de negocio, de la misma manera que se debe ser capaz de cambiar la interfaz de usuario de la aplicación sin que ello afecte a la lógica de negocio. Generalmente, esto significa utilizar una capa de abstracción como interfaces ordinarias de Java o entidades EJB entre los objetos del negocio y la persistencia.
- El código del acceso a datos debe ser mantenible. Sin la debida atención, el acceso a los datos puede representar una gran parte de la complejidad de la aplicación.

En el diseño de componentes de acceso a datos, se deben tener en cuenta las siguientes directrices:

- No acoplar el modelo de la aplicación al esquema de la base de datos.
- Abrir las conexiones lo más tarde posible y liberarlas tan pronto como sea posible.
- Evitar el acceso a la base de datos directamente desde diferentes capas de la aplicación. En lugar de ello, toda la interacción con la bases de datos debe hacerse a través de una capa de acceso a datos.
- Tomar ventaja del uso de una piscina de conexiones para reducir al mínimo el número de conexiones abiertas.
- Diseñar una estrategia de manejo de excepciones para manejar los errores de acceso a datos, y para propagar las excepciones a capas superiores.

El patrón DAO (Data Access Object por sus siglas en inglés) utiliza una capa de abstracción de interfaces entre la lógica de negocio y los componentes de la lógica de persistencia. La implementación de estas interfaces maneja la lógica de persistencia. El patrón DAO se describe ampliamente en *Core J2EE Patterns: Best Practices and Design Strategies* (Alur, et al., 2003).

Los Objetos de Acceso a Datos (también conocido simplemente como DAO), implementan el mecanismo de acceso que se requiere para trabajar con la fuente de datos de una aplicación. Independientemente del tipo de fuente de datos empleado, el DAO siempre proporciona una API uniforme a sus clientes y oculta completamente los detalles de la implementación de acceso a la fuente de datos de sus clientes. Debido a que la interfaz expuesta por el DAO a los

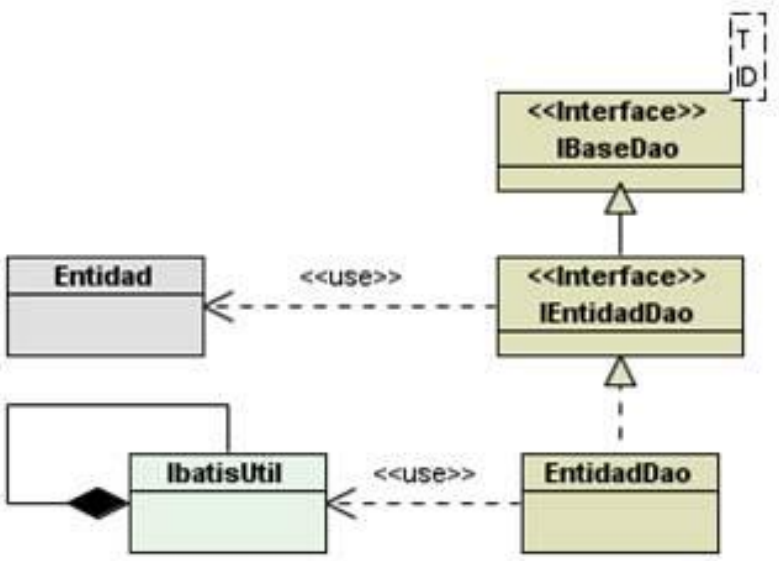
clientes no cambia cuando ocurren cambios en la fuente de datos, permite cambiar la implementación de los DAO sin alterar los componentes que están usando esta interfaz. Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos. (Alur, et al., 2003)

De manera simplificada la utilización del patrón DAO aporta los siguientes beneficios (Alur, et al., 2003):

- Expone interfaces uniformes para el acceso a los datos.
- Reduce el Acoplamiento, aumenta la manejabilidad.
- Centraliza el control de transacciones.
- Permite la transparencia.
- Permite la fácil migración.
- Reduce la complejidad del código en los clientes.
- Organiza todo el código de acceso a datos en una capa separada.
- Centraliza el control con manipuladores con un bajo acoplamiento.
- Proporciona una vista orientada a objetos.

Esta capa está compuesta por Objetos de Acceso a Datos que utilizan las entidades persistentes. La Figura 22 muestra un diagrama general de las relaciones existentes en esta capa.

Figura 22 Estructura de la Capa de Acceso a Datos



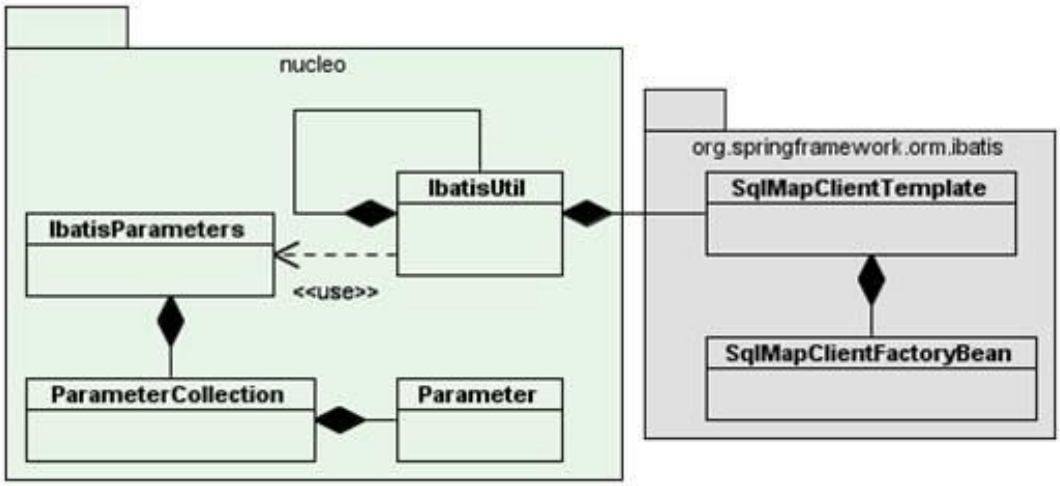
Por cada entidad persistente se creará un DAO correspondiente para la manipulación de la misma, a su vez existe una interfaz genérica (IBaseDAO) que expone los métodos CRUD tradicionales en aras de lograr una mayor reutilización de código y de estandarizar el trabajo con los diferentes DAOs.

Cada DAO se auxilia de la clase `IbatisUtil`, la cual utiliza el API que proporciona Spring para el trabajo con iBatis en el paquete `org.springframework.orm.ibatis`, para exponer las funcionalidades necesarias para trabajar con la base de datos. (Ver Figura 23)

La clase `iBatisUtil`, que implementa el patrón Singleton (Gamma, et al., 1994), utiliza la clase `IbatisParameters` que está compuesta por la clase `ParameterCollection`, que como su nombre lo indica tiene una colección de `Parameter`.

Todo este diseño simplifica en gran medida el trabajo con Spring e iBatis durante la ejecución de procedimientos almacenados, tratando a los mismos de la misma forma que se realiza con JDBC, registrando parámetros de entrada y de salida entre otros aspectos. Este elemento brinda al programador de acceso a datos muchas facilidades en cuanto a configuración y manipulación.

Figura 23 Diagrama de clases de IBatisUtil



3.2.4 Integración entre capas

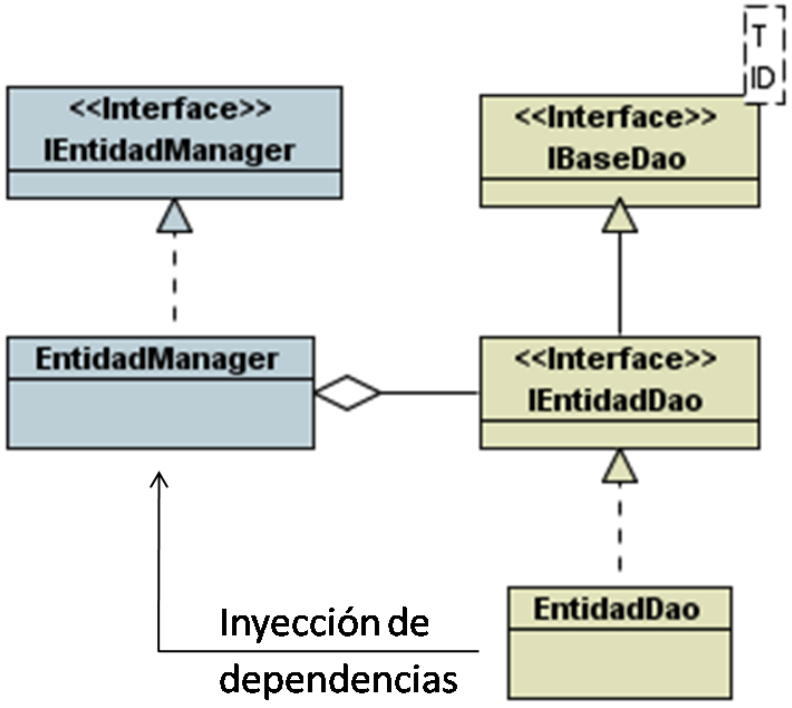
Para lograr la integración entre las diferentes capas de la aplicación se emplean dos mecanismos que tienen como objetivo principal resolver de una forma u otra problemáticas tales como:

- Lograr un mayor desacople entre las capas mejorando la capacidad de mantenimiento de la aplicación.
- Facilitar el desarrollo paralelo del sistema.
- Facilidad de uso e implementación.

El primero de los mecanismos radica en el propio concepto de Spring de Inyección de Dependencias (Ve epígrafe 2.2.1) y los principios de diseño que guían su utilización. La misma consiste en que las capas superiores conozcan una interfaz de la capa inferior y que obtenga la instancia concreta de esta interfaz a través del contenedor de objetos de Spring que a su vez es configurado mediante la definición de beans en ficheros XML.

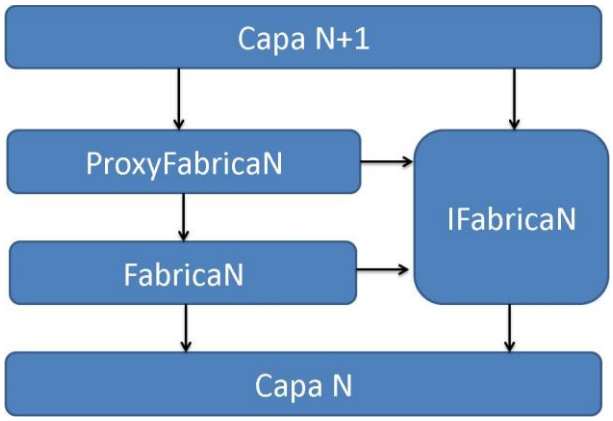
La Figura 24 muestra la integración de la capa de negocio y la de acceso a datos utilizando la inyección de dependencias. En este ejemplo la clase `EntidadManager` tiene una interfaz `IEntidadDao` y recibe su implementación concreta `EntidadDao` mediante el contenedor de Spring y beneficiándose de toda la potencialidad que este brinda.

Figura 24 Integración entre capas usando inyección de dependencias



El otro mecanismo de integración entre capas está guiado esencialmente por un conjunto de patrones de diseño que posibilitan resolver los problemas planteados al inicio de este epígrafe. En la Figura 25 se esquematiza la interacción entre los elementos de acople entre las capas.

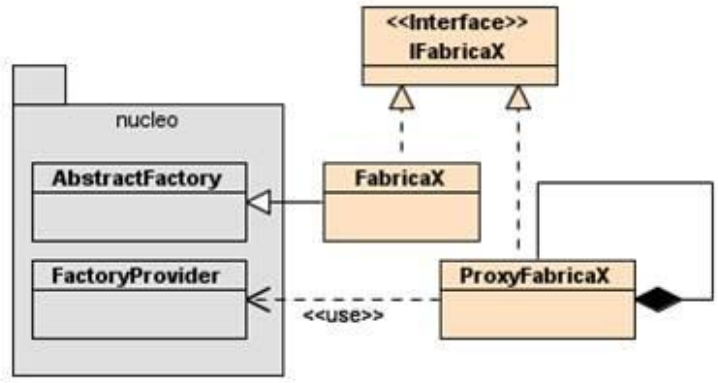
Figura 25 Integración entre capas usando patrones de diseño



- *Fachada:* Cada capa expondrá una fachada (IFabricaN) con el conjunto de funcionalidades que necesita la capa inmediata superior.
- *AbstractFactory:* Se definirán fábricas abstractas, por familias de objetos, con el objetivo de definir como se van a agrupar los objetos de dicha capa para ser creados por una fábrica concreta que forma parte de la capa en cuestión.
- *Puente:* El puente brinda un mecanismo dinámico para obtener objetos utilizando la configuración de un bean de Spring.
- *Proxy:* El proxy (proxy remoto) es el encargado de abstraer la forma en que se obtiene la fábrica concreta utilizando un "Puente".

En un escenario más específico el diagrama de clases para una capa X quedaría como se muestra en la Figura 26. En este caso `FactoryProvider` es el puente utilizado por el `ProxyFabricaX` para obtener la fábrica concreta (`FabricaX`) que construirá los objetos que fueron configurados a través de un fichero XML.

Figura 26 Diagrama de clases para la integración entre capas usando patrones de diseño

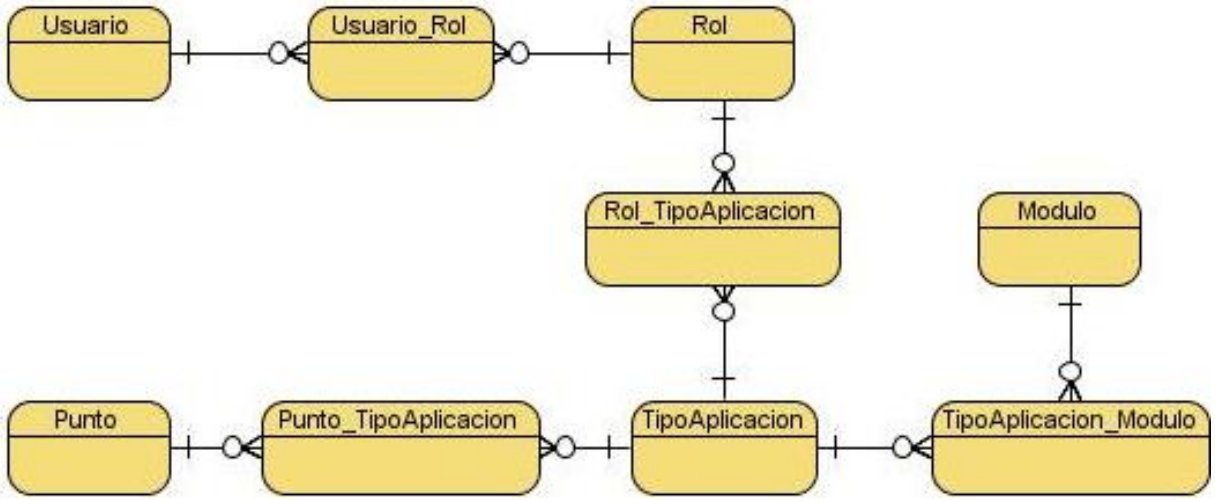


3.3 Seguridad

La seguridad de la aplicación se definió sobre un modelo basado en roles (Ver Figura 27). En este modelo a los usuarios le son asignados uno o varios roles, mientras que estos son asociados con uno o varios tipos de aplicación que pertenecen a un determinado módulo. En este caso específicamente, los involucrados son el Módulo de Administración y Control de Recursos y los tipos de aplicación Administración de Recursos Cliente y Administración de Recursos Servidor.

Cada tipo de aplicación solo puede ser ejecutada en un determinado punto que corresponde a una estación de trabajo, más específicamente a una dirección IP. En el caso de la aplicación Administración de Recursos Servidor, el sistema comprueba que la dirección IP donde se intenta desplegar la aplicación tenga los permisos necesarios para hacerlo, de no tener los privilegios el sistema no inicia.

Figura 27 Modelo Entidad-Relación para la seguridad

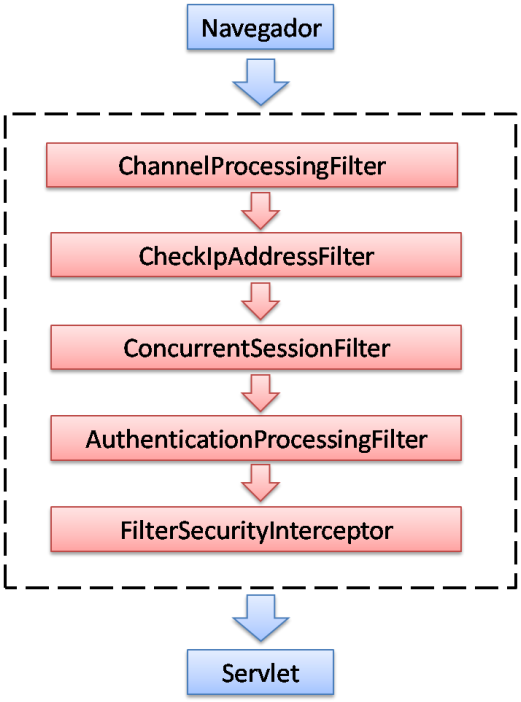


Por otro lado, la seguridad de la aplicación web además de estar basada en usuario y contraseña, verifica que la solicitud del cliente proviene de un IP con privilegios de acceso a la aplicación.

Esta verificación se realiza a través de la implementación de un filtro web que ejecuta esta política de seguridad y luego se sigue todo el flujo que Acegi Security implementa para asegurar peticiones HTTP (Ver epígrafe 2.4.3).

La Figura 28 muestra los filtros aplicados para asegurar las peticiones web.

Figura 28 Filtros web para la seguridad



Las responsabilidades de cada uno de estos filtros son las siguientes:

- *ChannelProcessingFilter:* garantiza que la petición realizada se encuentre sobre el canal requerido, redireccionando de forma correcta al protocolo de transporte correspondiente. Se utilizan dos canales uno seguro (HTTPS) y uno inseguro (HTTP). El primero de estos canales para el envío de usuario y contraseña durante el proceso de autenticación y el otro para el resto de las peticiones.
- *CheckIpAddressFilter:* comprueba que cada petición que se realice provenga de la misma dirección IP desde donde se realizó la autenticación, y al mismo tiempo garantiza que el acceso se origine desde una estación de trabajo (identificada por un número IP) con privilegios para acceder a la aplicación.
- *ConcurrentSessionFilter:* realiza dos funciones, una mantener actualizada la información del usuario en la sección y otra, determinar cuando ha expirado la misma. En caso de que expire la sección se notifica al usuario y se debe repetir el proceso de autenticación.
- *AuthenticationProcessingFilter:* es el encargado de realizar la autenticación, procesando el formulario de login para obtener las credenciales del usuario (nombre de usuario y contraseña), auxiliándose del `AuthenticationManager` para obtener la información requerida durante este proceso.

- *FilterSecurityInterceptor:* es el responsable de manipular la seguridad de los recursos HTTP realizando el proceso de autorización. Mediante el `AuthenticationManager` obtiene los permisos del usuario y utilizando el `AccessDecisionManager` determina si se tiene los privilegios de acceso al recurso solicitado.

Otro elemento significativo en la seguridad del sistema lo constituye la invocación de los métodos remotos utilizando DWR. Los mismos también cuentan con su mecanismo de seguridad que se apoya en la funcionalidad de Acegi Security para la protección durante la invocación de métodos remotos (Ver epígrafe 2.4.4).

De manera general se configuran los elementos necesarios para la utilización de AOP en Spring y su integración con Acegi Security, definiéndose los beans y los métodos de los mismos que se asegurarán. Este mecanismo utiliza la autorización definida en el contexto permitiendo la ejecución de las funcionalidades a los usuarios autenticados con los privilegios para hacerlo.

3.4 Tratamiento de excepciones

Una excepción es cualquier situación de error o comportamiento inesperado que encuentra un programa en ejecución. Las excepciones se pueden producir a causa de un error en el código, que no estén disponibles recursos del sistema operativo, que se violen determinas reglas de negocio, premisos insuficientes, etc.

La validación del lado del servidor previene de violaciones en las reglas del negocio que en determinados cosas pueden ocasionar excepciones, pero también es necesario tener en cuenta los errores inusuales e imprevistos que durante la ejecución de la aplicación puedan producirse.

Generalmente no se desea que el usuario vea las poco amigables trazas de la pila de la máquina virtual de Java, ni los mensajes de error del propio servidor de aplicación, que en al mayoría de los casos revelan información sensible relacionados con la implementación de la aplicación.

Cada implementación en las diferentes capas realiza el tratamiento de las excepciones mediante bloques `try/catch/finally` dándole el tratamiento específico en cada caso. Además se personalizan las excepciones lanzadas desde la base de datos mostrando un mensaje más detallado del error ocurrido.

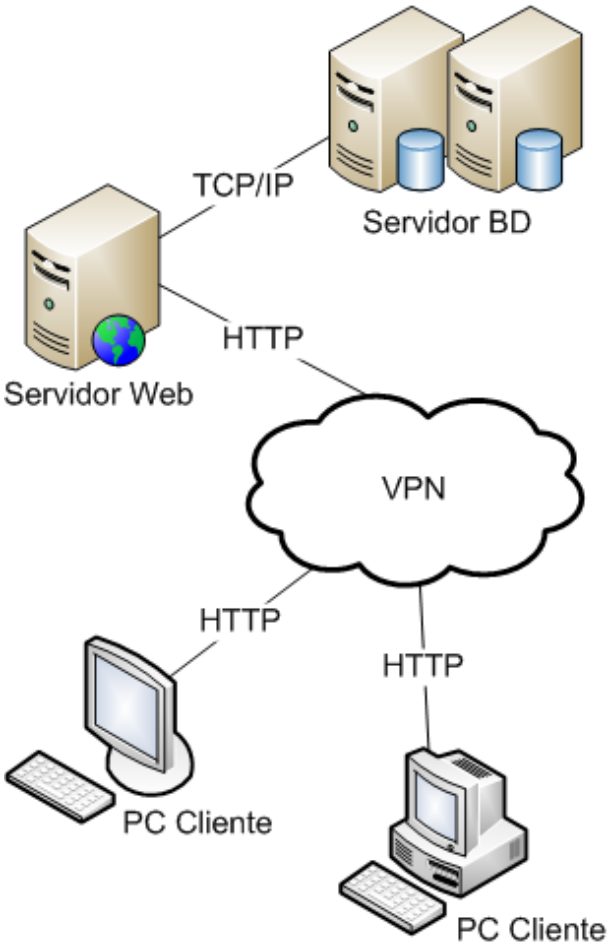
Para el tratamiento de errores, y su visualización al usuario, Spring MVC permite especificar las vistas (Views) que se deben mostrar cuando se producen determinados tipos de excepciones, controlándolas de forma centralizado.

Esta configuración se realiza con la clase `SimpleMappingExceptionResolver`. Su uso posibilita mapear las excepciones que se produzcan con una vista que informe al usuario del error ocurrido, mediante un mensaje adecuado.

3.5 Modelo de Despliegue

Un diagrama de despliegue modela la topología del hardware sobre el que se ejecuta un sistema. Muestra la configuración de los nodos que participan en la ejecución y de los componentes que residen en ellos. Un nodo es un elemento físico que existe en tiempo de ejecución. Representa un recurso computacional que, por lo general, tiene memoria y capacidad de almacenamiento. Representa a un procesador o un dispositivo sobre el cual se despliegan los componentes.

Figura 29 Diagrama de despliegue



El modelo de despliegue del módulo (Ver Figura 29) está formado por servidores y estaciones de trabajo (PC Cliente). Tanto el servidor de base de datos como el servidor web se encuentran ubicados en los Centros de Gestión de Emergencias, mientras que los nodos de las PC clientes se encuentran ubicados en cada uno de los diferentes organismos que proporcionan los recursos a estos centros.

Las estaciones de trabajo y el servidor web se encuentran conectados a través de una Red Privada Virtual (VPN por sus siglas en inglés) permitiendo una extensión de la red local sobre la red pública (Internet), garantizando la integridad, confidencialidad y seguridad de los datos que se transmiten.

3.5.1 Descripción de los nodos

Servidor de BD

Se recomienda la utilización de dos servidores en clúster para garantizar mayor vitalidad en el sistema de bases de datos.

Requerimientos de Hardware

- Tecnología Intel
- 2 procesadores duales a 2.6 Ghz.
- 4 GB de memoria RAM.
- 2 discos de 72 GB en RAID1.
- 500 GB en RAID1 para el almacenamiento de los datos. Debe ser almacenamiento compartido si se utilizan dos servidores.

Requerimientos de Software

- Sistema operativo Red Hat Linux Enterprise Advanced Server 4.0.
- Gestor de bases de datos Oracle Database 10g R2.
- Oracle Real Application Clusters. Solo necesario si se utilizan dos servidores.
- Oracle Spatial.

Otros Requerimientos

- Conexión mediante el protocolo TCP/IP a la red local.
- Dirección IP fija.
- Conexión de red a 1 Gigabit.

Cada servidor debe cumplir todos los requerimientos antes mencionados.

Servidor Web

Requerimientos de Hardware

- Tecnología Intel.
- 2 procesadores de 3.0 Ghz.
- 2 GB de memoria RAM.
- 2 discos de 72 GB en RAID1.

Requerimientos de Software

- Sistema operativo Red Hat Linux Enterprise Advanced Server 4.0.
- Entorno de ejecución de Java, Java(TM) SE Runtime Environment (JRE) 1.6.
- Servidor Web Apache Tomcat 5.5.17.

Otros Requerimientos

- Conexión mediante el protocolo TCP/IP a la red local.
- Dirección IP fija.
- Conexión de red a 1 Gigabit.

PC cliente

Requerimientos de Hardware

- Para el Navegador Web Mozilla Firefox 2.0.
  - Procesador mínimo 233 MHz, recomendado 500 MHz.
  - RAM mínima 64 MB, recomendada 256 MB.
  - Espacio mínimo en disco duro 50 MB disponibles, recomendado: 100 MB disponibles.
- Para el Navegador Web Internet Explorer 7.
  - Procesador mínimo: 233 MHz.
  - RAM mínima: 64 MB, recomendada: 128 MB.

Requerimientos de Software

- Navegador Web Internet Explorer 7 o Mozilla Firefox 2.0.

Otros Requerimientos

- Conexión mediante el protocolo TCP/IP al servidor de administración y control de recursos.
- Dirección IP fija.

- Conexión de red a 128 Kbps.
- Resolución gráfica de 800x600 pixeles.

3.6 Estilo de desarrollo

Definir un estilo de desarrollo que comprenda aspectos como pautas de codificación, convenciones de nombres, organización del sistema de archivos y buenas prácticas de codificación, así como la definición de los roles y sus responsabilidades apoyará grandemente a la organización del equipo de trabajo y una mayor productividad.

Estas restricciones van encaminadas esencialmente a preservar la legibilidad y estructuración del código con vista a facilitar revisiones, mantenimientos, reutilización o simplemente cambios en el equipo de desarrollo. Pueden estar sujetas a cambios durante el proceso de construcción de la aplicación.

3.6.1 Pautas de codificación

Las pautas de codificación o convenciones de código son de vital importancia para el desarrollo de una aplicación, ya que dictaminan estándares a seguir por todo el equipo de trabajo en la escritura del código fuente. Las convenciones de código son importantes para los desarrolladores por un buen número de razones: (Gosling, et al., 2000)

- El 80% del coste del código de un programa va a su mantenimiento.
- Casi ningún software lo mantiene toda su vida el autor original.
- Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo más rápidamente y a fondo.
- Si distribuyes tu código fuente como un producto, necesitas asegurarte de que está bien hecho y presentado como cualquier otro producto.

Las pautas adoptadas son las propuestas por Sun Microsystems en Java Language Specification (Gosling, et al., 2000) para la escritura de códigos fuentes en Java. Es importante destacar la siguiente convención de nombres:

Tabla 4 Pautas de nombrado

Elemento	Regla de nombrado	Ejemplo
Paquete	Todas las letras en minúsculas y solo letras.	`controladores`
Clases	Los nombres de las clases deben ser sustantivos simples y descriptivos usando el estilo camello.	`Recurso` `RecursoUtil`

Interfaces	Similar a las clases anteponiendo el prefijo "I".	`IRecursoManager`
Métodos	Utilizar verbos de acción que expresen su propósito y emplear la primera letra en minúsculas y luego el estilo camello.	`listar` `seleccionarPorId`
Variables	Se usa el mismo estilo de los métodos. Los nombres de las variables deben ser cortos y con un significado.	`recurso` `recursoDao`
Constantes	Todas las letras mayúsculas y si son compuestas separadas por "_".	`NUMERO_MAX`

3.6.2 Organización de los ficheros

Para la organización de los ficheros se sigue la organización tradicional para aplicaciones web utilizando Eclipse como IDE[27]. Se definen dos directorios principales, `src` para incluir todo el código fuente y `WebContent` para todos los ficheros de recursos correspondientes a la parte web en particular. El Anexo 1 ilustra la organización de los directorios del módulo en general.

La Tabla 5 resume la descripción de cada paquete del directorio `src`, detallando que elementos incluye en cada caso.

Tabla 5 Descripción de los paquetes del directorio src

Paquete	Descripción
`accesodatos`	Lógica asociada con el acceso a datos.
`accesodatos.dao`	Definiciones de las interfaces de los objetos de acceso a datos (DAO), siguiendo el patrón I<NombreEntidad>DAO.
`accesodatos.dao.impl`	Implementaciones concretas de los DAO nombrándolas según el patrón <NombreEntidad>DAOImpl.
`accesodatos.fabrica`	Definición de la interfaz de la fábrica de objetos DAO.
`accesodatos.fabrica.impl`	Implementación de la fábrica concreta y del proxy.
`accesodatos.map`	Ficheros de mapeo de iBatis.
`configuracion`	Ficheros XML de definiciones de objetos (beans) que serán incluidos en contexto de Spring así como ficheros de propiedades (.properties) con el resto de las configuraciones.
`dominio`	Definición de todas las entidades del negocio.

[27] IDE: Integrated Development Environment. Entorno Integrado de Desarrollo

`dominio.validador`	Implementación de los `Validator` de Spring para cada entidad según el patrón <NombreEntidad>Validator.
`negocio`	Incluye toda la lógica de negocio de la aplicación.
`negocio.fachada`	Interfaces de las fachadas de la capa de negocio nombradas con el patrón I<Nombre>Fachada.
`negocio.fachada.impl`	Implementaciones de las fachadas según el patrón <Nombre>FachadaImpl
`negocio.manager`	Definición de las interfaces de los manager (entidades del negocio) nombrados con el patrón I<Nombre>Manager.
`negocio.manager.impl`	Implementaciones concretas de los manager con el nombre siguiendo el patrón <Nombre>ManagerImpl.
`servicio`	Clases que serán expuestas con DWR para la invocación de métodos remotos con Javascript.
`util`	Incluye todas aquellas clases de apoyo que se utilizan en cualquier capa de la aplicación.
`web`	Lógica asociada a la parte de la capa de presentación incluida en el servidor de aplicaciones.
`web.controlador`	Clases controladoras (`Controller`), nombrados con el patrón <Nombre>Controller.
`web.util`	Clases de apoyo utilizadas solamente por los controladores durante el procesamiento de las peticiones.

La organización de los elementos dentro del directorio `WebContent` se define en la Tabla 6.

Tabla 6 Estructura del directorio WebContent

Directorio	Descripción
`css`	Hojas de estilos de la aplicación.
`imagen`	Imágenes de la aplicación.
`js`	Estarán todos los archivos Javascript de las validaciones así como el código necesarios para invocar los métodos con DWR.
`reporte`	Recursos necesarios para los reportes.
`WEB-INF`	Directorio especial para el contenedor web (Apache Tomcat).
`WEB-INF/dtd`	Ficheros dtd para la validación de los XML de la aplicación.
`WEB-INF/jsp`	Todos los ficheros JSP de la aplicación, organizados por funcionalidades.

`WEB-INF/lib`	Directorio para las librerías que se utilicen.
`WEB-INF/log`	Directorio para almacenar los logs de la aplicación.
`WEB-INF/tld`	Ficheros tld con la definición de los taglib utilizados en los ficheros JSP.

3.6.3 Flujo de trabajo

El flujo de trabajo es un conjunto de actividades que guían a los desarrollares en la construcción de los distintos componentes de la aplicación, permitiendo desarrollar las diferentes capas a la vez. Se define un rol por cada capa lógica de la aplicación: Desarrollador de Presentación, de Negocio y de Acceso a Datos (Ver Anexo 2).

Las diferentes responsabilidades de cada rol definido se resumen a continuación:

Desarrollador de presentación

1. *Crear vistas:* Se crearán las vistas que van a interactuar con el usuario, principalmente las páginas JSP.
2. *Crear los controladores:* Se declararán e implementan los controladores que van a interactuar con las vistas. En este paso son se incluirá lo indispensable para que una vista pueda ser mostrada.
3. *Crear validadores de los controladores:* Se implementan los validadores con la lógica de validación del lado del servidor.
4. *Crear los servicios de DWR:* Se implementan las clases que serán los servicios de DWR y que atenderán las peticiones remotas del lado del servidor.
5. *Implementar el código Javascript para DWR:* Se implementa todo el código Javascript necesario del lado del cliente para el trabajo con DWR, para hacer las llamadas asincrónicas así como la manipulación de los mensajes de retorno.
6. *Realizar pruebas a los servicios DWR:* Se comprueba el correcto funcionamiento de estos servicios a través de pruebas unitarias.
7. *Definir los servicios DWR en el contexto de la aplicación:* Se definen los servicios DWR en el contexto de la aplicación y se configura el servlet de DWR para atender las peticiones remotas.
8. *Implementar la lógica de los controladores:* Se implementa la lógica que va a ejecutar para atender la petición proveniente del cliente.

9. *Definir controladores en el contexto de la aplicación:* Se definen los controladores en el contexto de la aplicación, mapeando las peticiones que atenderá y con cuales vistas va a interactuar.

Desarrollador de negocio

1. *Crear entidades del negocio:* Se crea el conjunto de entidades del negocio que se van a utilizar en la aplicación.
2. *Definir interfaces de los Managers:* Se definen las distintas funcionalidades que están asociadas a una entidad de negocio específica.
3. *Definir interfaces de las Fachadas:* Se define una interfaz con las funcionalidades del negocio pertinentes para dar solución un grupo de funcionalidades en específico (por lo general un paquete de casos de usos).
4. *Implementar lógica de los Managers:* Se implementa las funcionalidades definidas en las interfaces de los managers, conteniendo toda la lógica de negocio asociada a la entidad.
5. *Realizar pruebas a los Managers:* Comprobar el correcto funcionamiento de los Managers mediante las pruebas de unidad.
6. *Implementar lógica de las Fachadas:* Se implementan todas las funcionalidades que se van a exponer a la capa de Presentación.
7. *Realizar pruebas a las Fachadas:* Se realizan pruebas de unidad a la fachada para comprobar su funcionamiento.
8. *Definir los Managers en el contexto de la aplicación:* Definir en el contexto de la aplicación los managers que fueron debidamente probados.
9. *Definir la fachada en el contexto de la aplicación:* Se define en contexto de la aplicación las fachadas que su funcionamiento se haya probado.

Desarrollador de acceso a datos

1. *Crear procedimientos almacenados:* Se implementaran los procedimientos almacenados necesarios para la interacción con la base de datos.
2. *Crear ficheros de mapeo:* Se mapean, en los archivos XML de iBatis, los procedimientos almacenados con las clases de entidad.
3. *Definir las interfaces de los DAO:* Se define las funcionalidades de los objetos de acceso a dato, inicialmente los métodos CRUD y luego en demanda los que se necesiten.
4. *Implementar los DAOs:* Implementar la lógica del acceso a datos.

5. *Realizar pruebas a los DAOs:* Realizar pruebas de unidad a los DAOs para verificar su correcto funcionamiento.
6. *Definir los DAOs en el contexto de la aplicación:* Definir en los ficheros de contexto donde se configura cada bean correspondiente a cada DAO.

CONCLUSIONES

La definición de un modelo de arquitectura sólido sobre el cual desarrollar provee numerosos beneficios que se reflejan directamente en la comprensión del sistema y que logra a su vez una mejor comunicación entre el equipo de trabajo. Por otro lado permite organizar el proceso de desarrollo, fomentando la reutilización de componentes, lo que trae consigo ahorro de tiempo y el aumento de la calidad de los sistemas desarrollados.

Se identificaron los elementos estructurales a partir de los cuales se compone el sistema, su comportamiento, los estilos arquitectónicos aplicables y los patrones de diseño que guían su implementación. De igual forma se definen como elementos sustanciales de una arquitectura de software la extensibilidad, reusabilidad, y la facilidad de uso de los sistemas informáticos.

Se determinó la infraestructura base de desarrollo, seleccionando la plataforma y los frameworks a emplear en la construcción del sistema. Además se definió la estructura lógica y física del módulo, la comunicación entre las diferentes capas, así como los mecanismos de seguridad y tratamiento de errores. También se estableció un estilo de desarrollo y un flujo de trabajo para guiar al equipo de trabajo durante el ciclo de vida del proyecto.

La implantación del Módulo de Administración y Control de Recursos en los Centros de Gestión de Emergencias 171 como parte fundamental del Sistema de Gestión de Emergencia de Seguridad Ciudadana 171 aspira a repercutir significativamente en el incremento de la seguridad ciudadana en la República Bolivariana de Venezuela.

Con la arquitectura propuesta se desarrolló el Módulo de Administración y Control de Recursos del SIGESC 171 contribuyendo favorablemente a que la aplicación cumpliera con los requisitos establecido por el cliente.

RECOMENDACIONES

Teniendo en cuenta el alcance del presente trabajo así como el análisis de los resultados se recomienda:

- Incluir la utilización de algún framework Ajax en la capa de presentación que permita agregar más dinamismo y funcionalidad a la aplicación.
- Valorar la migración del sistema a versiones superiores de Spring Framework para aprovechar las nuevas características que reducen el número de configuraciones.

BIBLIOGRAFÍA

Alur, Deepak, Crupi, John y Malks, Dan. 2003. *Core J2EE Patterns: Best Practices and Design Strategies.* Second Edition. s.l. : Prentice Hall PTR, 2003.

ANSI/IEEE Std 1471-2000. *Recommended Practice for Architectural Description of Software-Intensive Systems.*

Apache Foundation. Direct Web Remoting. [En línea] [Citado el: 3 de Febrero de 2009.] http://directwebremoting.org/dwr/overview/dwr.

Apache Software Foundation. iBATIS Overview. [En línea] [Citado el: 10 de Marzo de 2009.] http://ibatis.apache.org/overview.html.

Bass, Len, Clements, Paul y Kazman, Rick. 2003. *Software Architecture in Practice, Second Edition.* s.l. : Addison Wesley, 2003.

Begin, Clinton, Goodin, Brandon y Meadors, Larry. 2007. *iBATIS in Action.* s.l. : Manning Publications, 2007.

Beust, Cedric, Davies, John y Allamaraju, Subrahmanyam. 2002. *Programación Java Server con J2EE Edición 1.3.* s.l. : Anaya Multimedia, 2002.

Buschmann, Frank, y otros. 1996. *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns.* s.l. : John Wiley & Sons, 1996.

Canal Velasco, José Carlos. 2000. *Un Lenguaje para la especificación y validación de arquitecturas de software.* Escuela Técnica Superior de Ingeniería Informática Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga. 2000. Tesis de doctorado (Doctor en Informática).

Dijkstra, Edsger. 1983. The Structure of the The Multiprogramming system. *Communications of the ACM, 26(1).* 1983, págs. 49-52.

E. Perry, Dewayne y L. Wolf, Alexander. 1992. *Foundations for the study of software architecture.* s.l. : ACM SIGSOFT Software Engineering Notes, 17(4), 1992. págs. 40–52.

Fielding, Roy Thomas. 2000. *Architectural styles and the design of network-based software architectures.* University of California. Irvine : s.n., 2000. Tesis doctoral.

Fowler, Martin. 2004. Inversion of Control Containers and the Dependency Injection pattern. [En línea] Enero de 2004. [Citado el: 2009 de Febrero de 2.] http://martinfowler.com/articles/injection.html.

—. 2002. *Patterns of Enterprise Application Architecture.* s.l. : Addison Wesley, 2002.

Gamma, Erich, y otros. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1994.

Garlan, David y Perry, Dewayne E. 1995. *Introduction to the Special Issue on Software Architecture.* s.l. : IEEE Transactions on Software Engineering, 21:4, 1995.

Garlan, David y Shaw, Mary. 1994. *An Introduction to Software Architecture.* s.l. : School of Computer Science Carnegie Mellon University, 1994.

Gosling, James, y otros. 2000. *The Java Language Specification.* Second Edition. s.l. : Sun Microsystems, Inc., 2000.

Hayes-Roth, Frederick. 1994. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program.* s.l. : Teknowledge Federal Systems. Version 1.01, 1994.

Interface21, Smeets, Bram y Ladd, Seth. 2007. *Building Spring 2 Enterprise Applications.* s.l. : Apress Inc., 2007.

Jacobson, Ivar, Grady, Booch y Rumbaugh, James. 1999. *The UML Modeling Language User Guide.* s.l. : Addison-Wesley, 1999.

Johnson, Rod. 2003. *Expert One-on-One J2EE Design and Development.* s.l. : Wrox Press, 2003.

Johnson, Rod, y otros. Aspect Oriented Programming with Spring. *The Spring Framework - Reference Documentation.* [En línea] [Citado el: 15 de Enero de 2009.] http://static.springframework.org/spring/docs/2.0.x/reference/aop.html.

Johnson, Rod, y otros. 2005. *Professional Java Development with the Spring Framework.* s.l. : John Wiley & Sons, 2005.

Kayal, Dhrubojyoti. 2008. *Pro Java™ EE Spring Patterns: Best Practices and Design Strategies Implementing Java™ EE Patterns with the Spring Framework.* s.l. : Apress Inc., 2008.

Kruchten, Philippe. 1995. *The 4+1 View Model of Architecture.* s.l. : IEEE Software, 1995.

Ladd, Seth, y otros. 2006. *Expert Spring MVC and Web Flow.* s.l. : Apress Inc., 2006.

Lane, Thomas G. 1990. Studying Software Architecture Through Design Spaces and Rules. *Technical Report CMU/SEI-90-TR-18.* s.l. : Carnegie Mellon University, 1990.

Machacek, Jan, y otros. 2008. *Pro Spring 2.5.* s.l. : Apress Inc., 2008.

McGovern, James, y otros. 2003. *A Practical Guide to Enterprise Architecture.* s.l. : Prentice Hall PTR, 2003.

Meier, J.D., y otros. 2008. *Application Architecture Guide 2.0 Project - Patterns & Practices.* s.l. : Microsoft Corporation, 2008.

Minter, Dave. 2008. *Beginning Spring 2: From Novice to Professional.* s.l. : Apress, 2008.

Monroe, Robert, y otros. 1997. *Stylized Architecture, Design Patterns, and Objects.* s.l. : IEEE Software, 1997. págs. 43-52.

Perry, Dewayne E. y Wolf, Alexander L. 1992. *Foundations for the Study of Software Architecture.* s.l. : ACM SIGSOFT Software Engineering Notes, 17:4, 1992.

Pfleeger, Shari Lawrence. 2002. *Ingeniería de Software: Teoría y Práctica.* s.l. : Prentice-Hall, 2002.

Platt, Michael. 2002. *Microsoft Architecture Overview: Executive summary.* 2002. ", http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnea/html/eaarchover.asp.

Schutta, Nathaniel y Asleson, Ryan. 2006. *Pro Ajax and Java.* s.l. : Apress, 2006.

Shaw, Mary y Clements, Paul. 1996. *A field guide to Boxology: Preliminary classification of architectural styles for software systems.* Computer Science Department and Software Engineering Institute, Carnegie Mellon University. s.l. : Proceedings of the 21st International Computer Software and Applications Conference, 1996.

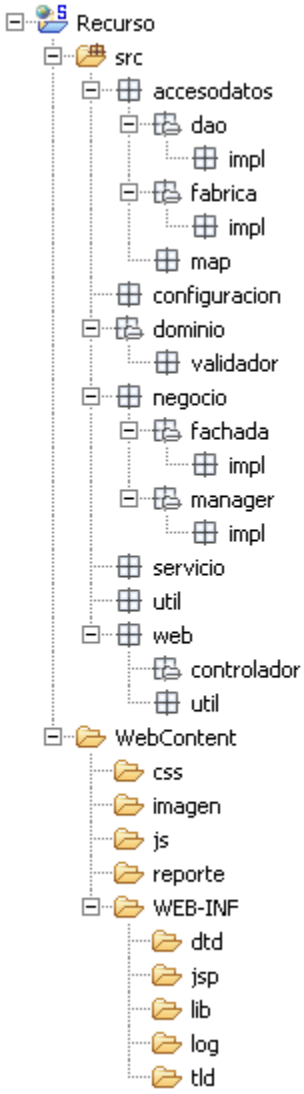
Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition (J2EE) Overview. [En línea] [Citado el: 8 de Abril de 2009.] http://java.sun.com/j2ee/overview.html.

Walls, Craig y Breidenbach, Ryan. 2008. *Spring in Action Second Edition.* s.l. : Manning Publications, 2008.

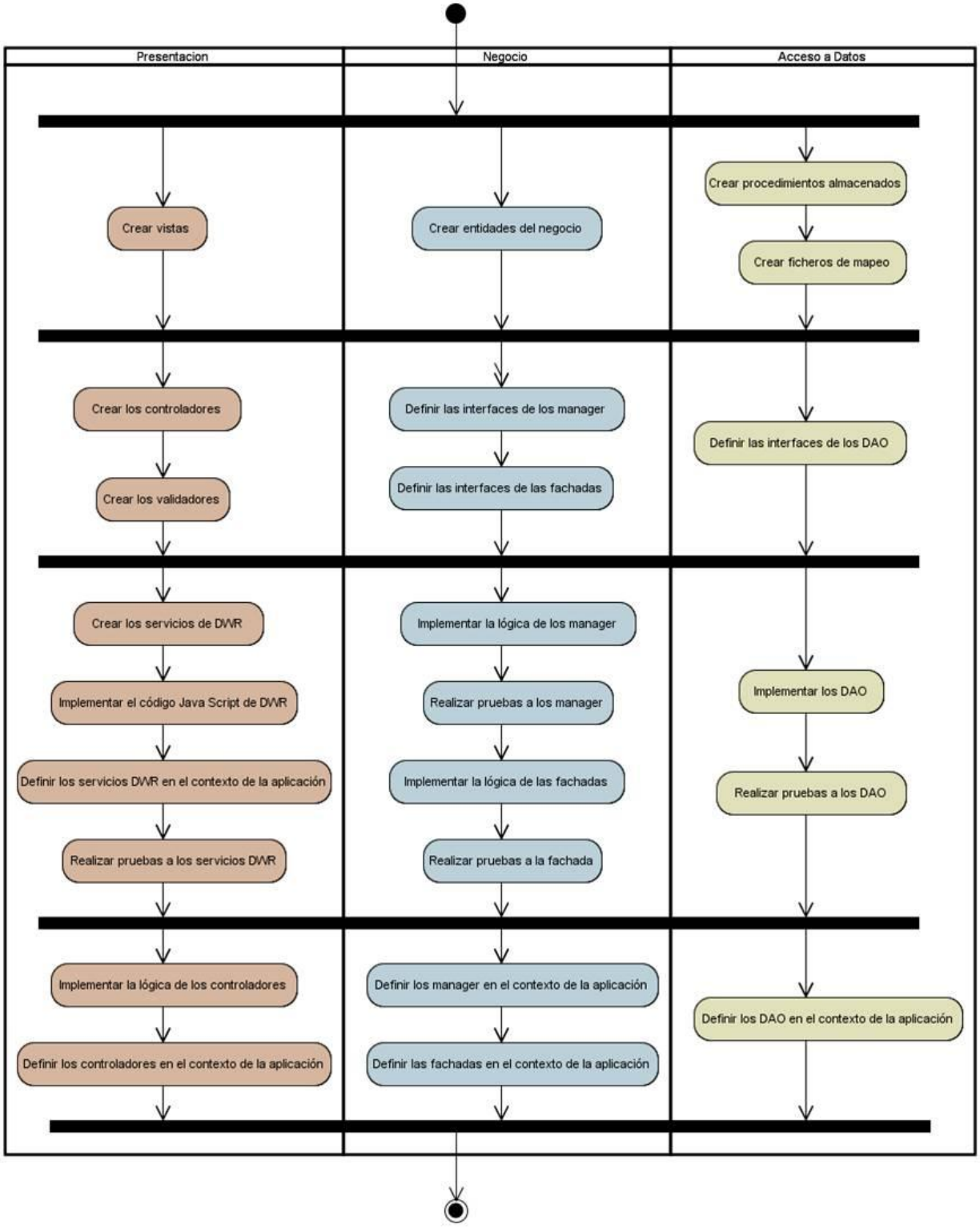
Zammetti, Frank W. 2008. *Practical DWR 2 Projects.* s.l. : Apress, 2008.

ANEXOS

Anexo 1 Organización física de los directorios



Anexo 2 Flujo de Trabajo



GLOSARIO DE TÉRMINOS

Ajax: acrónimo de Asynchronous Javascript And XML (Javascript asíncrono y XML). Es una tecnología asíncrona, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página.

API: Interfaz de Programación de Aplicaciones (Application Programming Interface) es el conjunto de funciones y procedimientos (o métodos, si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

BLOB: Binary Large OBjects, (grandes objetos binarios) son elementos utilizados en las bases de datos para almacenar datos de gran tamaño que cambian de forma dinámica.

CGLIB: Code Generation Library es una librería dinámica desarrollada para extender clases Java e implementar interfaces en tiempo de ejecución.

CLOB: Character Large OBject, (grandes objetos de caracteres) es una colección de datos de caracteres en un sistema de gestión de bases de datos, generalmente almacenados en lugares separados que hacen referencia dentro de la misma tabla.

Data Mapper: Mapeador de datos.

Framework: marco de trabajo.

iBatis: combinación de internet(i) y abatis (batis). Esta última es un término usado en defensas terrestres para designar a un obstáculo formado por ramas de árboles puestas en hileras, con las puntas en dirección al enemigo. Según sus creadores iBatis significa defensa en internet.

JDBC: Java Database Connectivity, más conocida por sus siglas JDBC, es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.

JDK: Java Development Kit es un conjunto de aplicaciones desarrolladas por Sun Microsystem que se utiliza para crear programas en lenguaje Java.

JNDI: Interfaz de Nombrado y Directorio Java (Java Naming and Directory Interface). Es una API J2EE para servicios de directorio.

Lazy loading: es un patrón de diseño comúnmente utilizado en la programación para aplazar el inicio de un objeto hasta el punto en el que se necesita. Puede contribuir a la eficiencia en el funcionamiento del programa y si se utiliza adecuadamente.

Middleware: es un software de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas.

Object-relational mapping: Mapeo objeto relacional. Es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional.

Open source: código abierto por su traducción en español, es el término con el que se conoce al software distribuido y desarrollado libremente.

ORM: ver Object-relational mapping.

Servlet: Es un objeto que se ejecuta en un servidor o contenedor JEE, especialmente diseñado para ofrecer contenido dinámico desde un servidor web, generalmente HTML.

SQL Maps: Mapeos SQL o correspondencias de SQL.

Stakeholder: Son todas aquellas personas u organizaciones que afectan o son afectadas directa e indirectamente por el proyecto. Ejemplos: accionistas, proveedores, clientes, desarrolladores, etc.