VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Utilización de patrones y la arquitectura J2EE para el diseño de la interfaz de usuario de la Herramienta Integral *MoProSoft* (HIM)"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA (COMPUTACIÓN)

P R E S E N T A:

KARIN VALDIVIESO CASTILLO.

DIRECTOR DE TESIS: DRA. HANNA OKTABA.

México, D.F. **2005.**

ANTECEDENTES

MoProSoft es un modelo de procesos que incorpora una serie de actividades bien definidas para el desarrollo y la gestión de proyectos de software. Las actividades se relacionan unas con otras en base a una persona encargada de llevarlas a cabo, es decir, un rol. Se tiene una jerarquía de roles que se enfocan en alcanzar objetivos a diferente nivel en una empresa. Estos objetivos se cumplen y comunican a otros procesos a través de la generación de productos que sirven de entrada a otro proceso y se obtiene alguna realimentación importante generando documentos de salida, cada uno de los productos generados tienen un proceso de revisión y validación que lo habilita para que pueda servir de referencia.

MoProSoft es un modelo de procesos para administración de proyectos de software orientado a la pequeña y mediana empresa. Las empresas mexicanas de software se encuentran en una etapa de madurez, y muchas de ellas adoptan prácticas diversas. Cada una de ellas tiene una forma de desarrollar software que si bien ha funcionado, quizá no se han obtenido resultados óptimos. MoProSoft ya se ha convertido en una norma mexicana para desarrollo de software. Aprovechando esta situación se propone una primera versión de una Herramienta para apoyar este modelo (*MoProSoft*), se pretende una herramienta que siga este modelo de procesos de tal forma que la administración de proyectos de software no sea tan complicada. El nombre de Integral viene porque se busca que la herramienta a desarrollar contenga lo necesario para llevar a cabo la administración y desarrollo de proyectos de software, requiriendo componentes adicionales mínimos. El desarrollo de la herramienta pretende ser una referencia para seguir adecuadamente el modelo y es una vía de difusión de la norma.

Se han investigado dos plataformas para el desarrollo de la herramienta: .NET y J2EE. Los dos marcos de trabajo permiten el desarrollo de una herramienta WEB, cada uno de ellos requiere un contenedor especial.

Para alojar la aplicación .NET se requiere un contenedor propietario proporcionado por Microsoft, así que para alojar la herramienta se necesita por lo menos su servidor de aplicaciones Web, lo que implica un costo en herramientas. Con J2EE existen distribuciones gratuitas de servidores de aplicaciones WEB, que permiten el desarrollo de aplicaciones de prueba.

Se pretende que la herramienta generada sea de distribución libre y que su distribución se pueda probar sin necesidad de gastar dinero en su adopción, por éstas razones se ha elegido como plataforma de desarrollo la arquitectura J2EE.

J2EE es un conjunto de especificaciones que utiliza el lenguaje Java 2 como su principal producto y cuenta con el apoyo de fabricantes líderes entre los que encontramos a Sun e IBM.

Muchos desarrollos se han hecho en Java y se han creado soluciones a problemas de software recurrentes. Estas soluciones se han implementado a través de patrones o plantillas de software. J2EE hace uso extenso de los patrones para construir soluciones robustas. En especial, el patrón MVC (Modelo-Vista-Control)) se ha utilizado ampliamente y con gran éxito por ello se ha elegido como el patrón más general para

esta herramienta. La parte de la vista o interfaz de usuario es muy importante y debe mantener ciertas reglas para su construcción. Se han desarrollado algunos patrones para manejar la lógica para construir la interfaz. Es en esta parte donde haré mi aportación, revisando, seleccionando e implementando esos patrones para controlar el despliegue de la interfaz de usuario.

INTRODUCCIÓN

El desarrollo de proyectos de software en México ha dado lugar a la creación de muchas empresas de software. Cada empresa que presta un servicio requiere planeación y gestión de cada una de sus actividades con el objetivo de ser una organización eficiente. Lograr una buena planeación y comunicarla no es un asunto sencillo, requiere de una disciplina y un plan de trabajo bien definido. El uso de un modelo de procesos no garantiza el éxito operativo de una empresa, sin embargo, es un buen punto de partida para organizar las actividades que se realizan dentro de la organización.

MoProSoft[DR] es un modelo de procesos que define una serie de actividades que una organización puede ejecutar para realizar mejor sus tareas. Se basa en roles que un trabajador puede tener en la organización y define una serie de productos que se obtienen como resultado de una actividad, definida en uno de los tres procesos principales: Alta dirección, Gestión y Operación. Dos profesoras de la maestría, la Dra. Hanna Oktaba y la maestra Guadalupe Ibargüengoitia proponen generar una herramienta de software que pueda servir de referencia al modelo. La herramienta propuesta se llama "Herramienta Integral MoProSoft". En una primera versión se proponen las tecnologías J2EE y RDF para cubrir los temas de investigación de cada uno de los seis participantes involucrados en su desarrollo.

Se propone una división en varias capas para dar oportunidad de que cada integrante pueda aportar algo en el desarrollo de la herramienta y el producto generado pueda servir como un trabajo de tesis.

En la Tabla 1 se muestra un resumen de los títulos que cada integrante del equipo desarrolla.

Integrante	Tema de Tesis
Karín Valdivieso Castillo	Utilización de patrones y la arquitectura J2EE para el diseño de la interfaz de usuario de la Herramienta Integral *MoProSoft* (HIM)
Jorge Cruz Vázquez	Análisis e implementación de esquemas de seguridad aplicados a la herramienta integral *MoProSoft* (HIM)
Marcos Oscar Vázquez Morales	Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para *MoProSoft* (HIM)
Araceli Eugenia Mercado Fernández	La Sincronización de los Elementos de una Base de Conocimiento para *MoProSoft* y su Aplicación en una Herramienta Integral
Ernesto Hernández Uribe	Uso de la tecnología RDF para representar y manejar los procesos MoProSoft y su aplicación en HIM
Hafiz Zurita Rendón	Arquitectura de la Herramienta Integral para *MoProSoft*

Tabla 1. Lista de participantes y tema propuesto.

Los objetivos que se persiguen con el desarrollo de esta tesis son:

- Revisar el conjunto de patrones existentes en la comunidad de software en Java, enfocados al manejo de la interfaz de usuario.
- Seleccionar un conjunto de patrones que pueden implementarse para construir la interfaz de usuario y modificarla en forma dinámica.
- Implementar esos patrones orientados a la herramienta integral MoProSoft.
- Integrar la capa de las Vistas con las otras capas de la arquitectura MVC.
- Construir una guía práctica de los patrones utilizados para distribuirlos en la comunidad de software.

Es importante recalcar que este trabajo se centra en una recopilación de patrones de la capa de presentación por lo que los patrones descritos e implementados en la capa de control y del modelo de datos se tratan con menor profundidad. El control y el modelo de datos se tratan con mayor profundidad en las tesis de otros compañeros de maestría[1].

Se tiene la intención de mostrar la importancia y la aplicación de los patrones de la capa de presentación de J2EE en el desarrollo de una herramienta que utiliza a MoProSoft como modelo de procesos. J2EE ha tenido gran aceptación como un estándar para el desarrollo de aplicaciones empresariales, sin embargo, la especificación completa presenta cierta complejidad que dificulta su aplicación para los desarrolladores de aplicaciones.

J2EE distribuye una aplicación de referencia llamada Java *PetStore* RI. Analizarla completamente hace difícil la ubicación de las partes que interesan revisar y aplicar.

1 El control se trata de manera más profunda en la tesis "Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para MoProSoft (HIM)" y el modelo de datos se trata en la tesis "Uso de la tecnología RDF para representar y manejar los procesos MoProSoft y su aplicación en HIM".

La pregunta que se pretende responder es si es posible aplicar de manera sencilla los patrones de la capa de presentación de la especificación J2EE, para generar entonces una pequeña aplicación que muestre exclusivamente la aplicación de los patrones de la capa de presentación.

CONTENIDO

La presente documentación se divide en cinco capítulos que muestran gradualmente el marco en que se aplican los patrones J2EE: *Front Controller, Composite View, View Helper, Intercepting Filter* y los patrones *Validator* y *Model-View-Controller*.

1. El primer capítulo muestra el concepto de patrón de software y la necesidad de su utilización como una solución a un problema que se presenta de manera recurrente.

2. El segundo capítulo muestra un arreglo de patrones (*framework*)[2] que representan la arquitectura del sistema de tal forma que se pueda implementar el sistema de manera organizada.

3. El tercer capítulo muestra la especificación J2EE y una descripción de los patrones de la capa de presentación que se pretenden aplicar.

4. El capítulo cuatro muestra el análisis realizado en la creación de la herramienta HIM, se muestra un modelado de clases elemental donde intervienen entidades simples para representar el problema desde el punto de vista de los programadores. Se detalla mucho más el modelo obtenido en el análisis de tal forma que se obtenga una referencia en la implementación, es en el diseño cuando las clases que solucionan el problema se organizan en base a los patrones seleccionados en el *framework*.

5. El capítulo cinco muestra la implementación de la herramienta con base en el modelo de diseño presentado.

Es importante señalar que todos los elementos que se presentan en este documento se refieren y dan una importancia a la parte de presentación de la herramienta que es la que se pretende modelar. La capa de reglas de negocio **[Vazquez]** y modelo de datos **[Uribe]** se mencionan aunque no se hace referencia a ellos directamente.

2 El término framework tiene varios significados dependiendo del contexto en que se aplica. Varios autores definen a un framework como una aplicación semicompleta, reusable que puede especializarse y extenderse para producir aplicaciones específicas. **[Yacoub].**

I. INTRODUCCION A LOS PATRONES Y J2EE.

1.0 Introducción

El presente capítulo ofrece un panorama general de los patrones de software, desde sus orígenes hasta su aplicación en proyectos actuales. Se describe su significado y la ventaja de adoptarlos como una herramienta para el desarrollo de proyectos de software.

En los últimos años se han observado cambios importantes con respecto al panorama de desarrollo de software empresarial. Como una parte importante de éste ubicamos a la edición empresarial Java 2 (J2EE), que provee un esquema unificado para desarrollar aplicaciones distribuidas basadas en servidor. La amplia adopción de tecnologías de J2EE ha proveído a la comunidad de desarrollo de estándares abiertos sobre los cuales construir arquitecturas basadas en servicios para la empresa.

El aprendizaje de las tecnologías J2EE puede confundirse con aprender a diseñar con tecnologías J2EE. Muchos libros sobre Java explican aspectos específicos de la tecnología, pero no indican con claridad la forma de aplicarlos.
Para desarrollar un sistema aceptable es conveniente responder a las siguientes preguntas:

- ¿Cuáles son las mejores prácticas?
- ¿Cuáles no son prácticas recomendables?
- ¿Cuáles son los problemas recurrentes y soluciones probadas a estos problemas?
- ¿Cómo se reestructura el software desde una versión menos óptima o mala práctica a una mejor práctica descrita en un patrón?

Los buenos diseños se descubren con base en la experiencia. Cuando estos diseños se comunican como patrones usando una plantilla estándar se convierten en un mecanismo poderoso. Al difundirlo pueden influenciar la forma en que se diseña y construye software. [Deepak]

1.1 Necesidad de utilizar patrones.

Para diseñar buen software, contando con poca experiencia, es difícil tomar las decisiones correctas, decisiones que lleven a conseguir productos de calidad. Podemos aprovechar la experiencia de otros a través de los patrones.

Al usarlos obtenemos algunos beneficios:

- Se documentan formalmente los compromisos adquiridos en el diseño de software: sobre los pros y los contras de las elecciones en el desarrollo. La

estandarización de los patrones facilita a los profesionales la comprensión de los efectos que pueden conllevar las decisiones.
- Se puede construir un catálogo de patrones. Esto permite que los nuevos desarrolladores se beneficien de la experiencia obtenida a lo largo de los años.
- Se pueden interrelacionar los patrones, por lo que un desarrollador puede ver qué patrón o que grupo de patrones se pueden utilizar en un proyecto. [Stelting]

1.2 Referencias históricas.

En 1970s, Christopher Alexander publicó varios libros documentando patrones en la ingeniería civil y la arquitectura[3]. Posteriormente la comunidad de software adoptó la idea de los patrones basados en su trabajo, aunque anteriormente había un interés de la comunidad de software por estas ideas. **[Alexander]**.

El trabajo de Alexander captó el interés de la comunidad de la programación orientada a objetos, y a lo largo de esa década algunas personas desarrollaron los patrones de software. Kent Beck y Ward Cunningham estuvieron entre los primeros, discutiendo un conjunto de patrones de diseño de *Smalltalk* en una presentación en la conferencia de OOPSLA en 1987. James Coplien fue otro de los que promocionó activamente los principios de los patrones, escribiendo un libro sobre expresiones comunes en C++ a principios de los noventa.**[Coplien].**
Otro foro importante para la evolución del movimiento sobre los patrones fue el grupo *Hillside*, establecido por Kent Beck y Grady Booch.

Los patrones de software se popularizaron por el libro **Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizables. (*Design Patterns: Elements of Reusable Object-Oriented Software)*** publicado por Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides (también conocidos como la pandilla de los cuatro "Gang of Four", ó GoF). **[Gamma]**. El trabajo de GoF resultó en patrones que se convirtieron en un tema de discusión común en los equipos de desarrollo alrededor del mundo El punto importante a recordar es que los patrones, que ellos describieron, no fueron su invento. Ellos identificaron diseños recurrentes en muchos proyectos y los documentaron.

Otro trabajo importante que impulsó el desarrollo de los patrones fue el libro ***Pattern-Oriented Software Arquitecture, A system of Patterns*** de Buschmann, Meunier, Rohnert, Sommerlad y Stal.

Desde estas publicaciones se han generado muchos libros sobre patrones para varios dominios y propósitos.

3 Dos referencias importantes sobre el trabajo de Cristopher Alexander son: "*The Timeless Way of Building*".[Alexander] y "*A Pattern Language*".[Alex]

1.3 Definición de patrón.

Los patrones tratan sobre comunicar problemas y soluciones. Nos permiten documentar un problema recurrente conocido, su solución en un contexto particular y comunicar este conocimiento a otros. Uno de los elementos claves en la definición previa es la palabra ***recurrente***, ya que el objetivo del patrón es fomentar el reuso conceptual a lo largo del tiempo.

Algunas definiciones conocidas de patrones son las siguientes:

Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un problema y una solución.
—Christopher Alexander [Alex].

Cada patrón es una regla de tres partes la cual expresa una relación entre un contexto, un sistema de fuerzas que ocurren repetidamente en ese contexto y una cierta configuración de software la cual permite a estas fuerzas resolverse a sí mismas. –Richard P. Gabriel [Gabriel].

Un patrón es una idea que ha sido útil en un contexto práctico y probablemente será útil en otros.
—Martin Fowler [Fowler].

Existen muchas definiciones para un patrón, pero todas esas definiciones tienen un tema común relacionados con una recurrencia del tipo problema/solución aplicados a cierto contexto.

Algunas de las características de los patrones son:

- Se observan a través de la experiencia.
- Se escriben típicamente en un formato estructurado (Plantilla).
- Evitan invertir mucho tiempo a problemas parecidos.
- Existen a diferentes niveles de abstracción.
- Son sujetos a mejoras continuas.
- Son artefactos que se pueden utilizar muchas veces.
- Comunican diseños y mejores prácticas.
- Pueden utilizarse en conjunto para resolver problemas grandes.

1.4 Clasificación de los patrones.

Los patrones representan soluciones expertas a problemas recurrentes en un contexto y así se han capturados a muchos niveles de abstracción y en muchos dominios.
Se han sugerido muchas categorías para clasificar patrones de software.

La clasificación más común es la siguiente:

- Patrones de diseño.
- Patrones de arquitectura.
- Patrones de análisis.
- Patrones de creación.
- Patrones de estructura.
- Patrones de comportamiento. [Stelting].

Aún en esta breve lista de categorías, se observan varios niveles de abstracción y esquemas de clasificación.

En la arquitectura J2EE se maneja una categoría más simple de patrones. Cada patrón flota entre un patrón de diseño y un patrón de arquitectura, porciones de cada patrón podrían quedar a un nivel más bajo de abstracción.

Se clasifica cada patrón dentro de una de las siguientes capas lógicas:

- Capa de Presentación
- Capa de Negocio
- Capa de Integración

En algún punto en la evolución del catálogo de patrones, posiblemente crecerá a un tamaño que garantizará su clasificación usando una estructura más sofisticada.

1.5 Plantillas de patrones.

Uno de los resultados más útiles del trabajo de Alexander fue el desarrollar una plantilla para representar los patrones, que se suele denominar formulario o formato. El formulario de Alexander usa cinco áreas para formalizar la explicación de un patrón y su solución.
Es importante que un patrón proporcione un nombre descriptivo para el patrón y la respuesta a la pregunta "¿Qué hará este patrón por nosotros?".
Además debería incluir una exposición del problema, una explicación de cómo el patrón soluciona el problema y una lista de ventajas, inconvenientes y compromisos asociados con el uso de los patrones.
Se han desarrollado algunas variaciones del formulario de Alexander para adaptarlos a las necesidades de los desarrolladores. Un ejemplo puede ser el siguiente:

- **Nombre**: Un nombre descriptivo del patrón

- **También conocido como**: nombres alternativos en caso de que los haya.
- **Propiedades**: La clasificación del patrón. Definimos un patrón de acuerdo con dos aspectos principales:
  * Tipo: Si es de creación, comportamiento, estructura, etc.
  * Nivel: Se aplica a una clase, a un grupo de clases o a la arquitectura.
- **Propósito**: Una breve explicación de las implicaciones del patrón
- **Presentación**: Una breve descripción de un problema que puede ser afrontado con este patrón, usando un ejemplo para mostrarlo.
- **Aplicabilidad**: Cuándo y por qué sería deseable usar este patrón de diseño.
- **Descripción**: Una explicación más detallada del patrón, indicando qué hace y cómo se comporta.
- **Implementación**: Una explicación de qué debe hacerse para implementar este patrón. Si sabe que quiere usar este patrón, esta sección ayudará a implementarlo
- **Ventajas e inconvenientes**: Las consecuencias de usar el patrón y los compromisos asociados al uso del mismo.
- **Variantes**: posibles implementaciones alternativas y variaciones del patrón.
- **Patrones relacionados**: otros patrones que están asociados o con los que tiene una relación estrecha.
- **Ejemplo**: Un ejemplo con código. [Stelting].

1.6 Beneficios de usar patrones.

Los patrones son fáciles de usar, aunque no siempre son fáciles de entender. Entender un patrón requiere un análisis cuidadoso. Una vez que los patrones son comprendidos podemos obtener muchas ventajas. En los siguientes puntos se explican con detalle algunas de las ventajas obtenidas al utilizar patrones. [Deepak].

1.6.1 Influencia de una solución probada.

Un patrón está basado en un documento que es evidencia de que la solución que se ofrece ha sido utilizada una y otra vez para resolver problemas similares en diferentes etapas y en proyectos diferentes. Los patrones proveen un mecanismo de reuso, ayudando a que los desarrolladores y arquitectos no trabajen desde cero. [Deepak].

1.6.2 Vocabulario común.

Los patrones proveen a los diseñadores con un vocabulario y formato comunes.
Un diseñador que no cree en la necesidad de los patrones puede gastar mayor esfuerzo para comunicar sus diseños a otros diseñadores o desarrolladores. Los diseñadores usan el vocabulario de patrones para comunicarse efectivamente. Por ejemplo, supongamos que una solución involucra al patrón *Value Object*. Primero, se

puede describir el problema sin ponerle una etiqueta. Por ejemplo, se puede describir la necesidad de la aplicación de intercambiar datos con *Enterprise Beans,* la necesidad de maximizar el rendimiento debido a la sobrecarga de la red. Más tarde, una vez que se ha aprendido como aplicar el patrón *Value Object* al problema, podemos referirnos a una situación similar en términos de una solución *Value Object* y construirla desde ahí.

Afirmaciones como las siguientes podrían agregarse al vocabulario:
- *Deberíamos usar Data Access Objects en nuestros servlets o beans.*
- *Usemos Front Controller y Service to Worker. Podríamos usar Composite Views para algunas páginas complejas.* [Deepak].

1.6.3 Restringir el espacio de la solución.

La aplicación de patrones introduce un componente de diseño principal que son los límites. Al usar un patrón se restringe o se crean límites dentro de un espacio de solución al cual el diseño y la implementación deben apegarse.

Ir más allá del límite establecido por el patrón rompe la adherencia al patrón y al diseño, y puede llevar a una situación fuera de control.
Los patrones no limitan la creatividad sino que describen una estructura o modelo a un nivel de abstracción.
Los diseñadores y desarrolladores aún tienen muchas opciones para implementar otros patrones dentro de estos límites. [Deepak].

1.6.4 Abstracción de software y reutilización.

Abstracción y reutilización de software son dos conceptos importantes en la programación.
La abstracción representa la forma en que los desarrolladores resuelven problemas complejos. La reutilización de software se busca continuamente, se prefiere utilizar código probado con anterioridad que escribir fragmentos de código nuevos cada vez que se crea una aplicación.

La forma más primitiva de reutilizar el código es la de usar fragmentos de código (CaP-Cut and Paste), sin embargo no contiene mucho grado de abstracción.
La reutilización de algoritmos proporciona una forma más general de gestionar la reutilización, abstrae un problema particular.
La reutilización funcional y estructuras de datos permiten una abstracción más directa ya que una función puede copiarse en los nuevos proyectos.
En la tecnología orientada a objetos se tiene un mayor grado de abstracción pues se empaquetan las estructuras (datos) con las funciones (comportamiento) y se obtiene una forma más efectiva de reutilizar un elemento de software.
La posibilidad más atractiva de los patrones es que permiten que los desarrolladores apliquen con mayor efectividad las técnicas de reusabilidad.

En la Tabla 1 se resumen los diferentes grados de reusabilidad y abstracción que pueden aplicarse al código generado en el desarrollo. [Stelting]

Tipo de reutilización	Reusabilidad	Abstracción	Genericidad
Fragmento de código	Muy pobre	Nada	Muy pobre
Estructura de datos	Buena	Tipo de datos	Moderada-Buena
Funcional	Buena	Método	Moderada-Buena
Plantilla	Buena	Operación para tipo	Buena
Algoritmo	Buena	Fórmula	Buena
Clase Interfaz Polimorfismo Clase abstracta	Buena	Datos+métodos	Buena
Biblioteca de código	Buena	Funciones	Buena-Muy buena
API	Buena	Clases útiles	Buena-Muy buena
Componente	Buena	Grupo de clases	Buena-Muy buena
Patrón de diseño	Excelente	Solución a problema	Muy buena

Tabla 1. Grado de reusabilidad del software. [Stelting].

1.7 Aplicación de los patrones a la arquitectura J2EE

Desde su surgimiento el lenguaje Java ha experimentado muchos cambios. Se han incluido nuevas bibliotecas y estándares para cubrir varias necesidades. La empresa Sun[4] en compañía de líderes de la industria de software unificó todos los estándares y bibliotecas bajo una plataforma llamada J2EE. J2EE es una plataforma para desarrollar aplicaciones distribuidas.

[4] La empresa Sun en compañía de otros líderes de la industria como IBM, y la comunidad de procesos de Java (JCP-*Java Community Process*).

La plataforma J2EE ofrece muchas ventajas a la empresa entre los que podemos mencionar:

1. Establece estándares para diferentes áreas como conectividad a la base de datos, componentes de lógica de negocio, componentes Web, protocolos de comunicación e interoperabilidad.
2. J2EE promueve implementaciones basadas en estándares abiertos[5] y basados en estándares que aseguran interoperabilidad entre productos de diferentes fabricantes.
3. J2EE incrementa la productividad del programador ya que los programadores Java puede aprender fácilmente las tecnologías J2EE basadas en Java. [Deepak].

En los siguientes capítulos se abordarán con detalle la arquitectura J2EE y los patrones de la capa de presentación involucrados en el desarrollo de la herramienta HIM (Herramienta Integral MoProSoft), explicando el entorno en que se desarrolla la herramienta y justificando las elecciones en el diseño.

[5] Se cuenta con un documento en la que se especifica la arquitectura J2EE y los fabricantes de productos deben adherirse a esta especificación, sin embargo, cada uno de ellos agrega extensiones que no permiten una portabilidad total.

II. LA ARQUITECTURA J2EE

2.0 Introducción.

En este capítulo se presenta una visión general de la plataforma empresarial de Java 2 (J2EE) y las tecnologías involucradas. Se mencionan las bibliotecas más importantes de la especificación mediante una breve descripción. Se describe a manera de plantilla el patrón de arquitectura MVC que se utilizará para el desarrollo de la herramienta HIM.

2.1 Definición de Arquitectura.

La arquitectura es un término usado al diseñar aplicaciones de software. Este término se refiere a la manera en la que se diseña el sistema tanto física como lógicamente. [Gabrick].

En el diseño físico se especifican exactamente donde se encontrarán las piezas de la aplicación como discos, ejecutables, los cables de red y las computadoras.
En el diseño lógico o conceptual se especifica la estructura de la aplicación y sus componentes sin tomar en cuenta dónde se localizará el software, hardware e infraestructura. Tales conceptos incluyen el orden de procesamiento, la organización de los componentes para reducir y hacer eficiente el mantenimiento.

Ambos diseños son muy importantes porque ofrecen una visión complementaria. Un diseño adecuado debe permitir su implantación en varias plataformas y configuraciones. Esta característica llamada portabilidad es un punto deseable para permitir que la aplicación sea flexible y escalable.
Una arquitectura de un sistema es el arreglo funcional del software, hardware y componentes de red que lo integran. [Gabrick].

2.2 Organización en capas de una arquitectura.

Al desarrollar un sistema es conveniente agrupar las tareas en términos de capas o niveles lógicos. Una capa de la aplicación es una agrupación lógica de los componentes que proveen a los usuarios y a otros subsistemas cierta funcionalidad. Los sistemas normalmente operan sobre datos propios, interactúan con sistemas externos y proveen una interfaz a sus usuarios. Esta característica de los sistemas da como resultado un patrón de n-capas como el que se muestra en la figura 2.1:

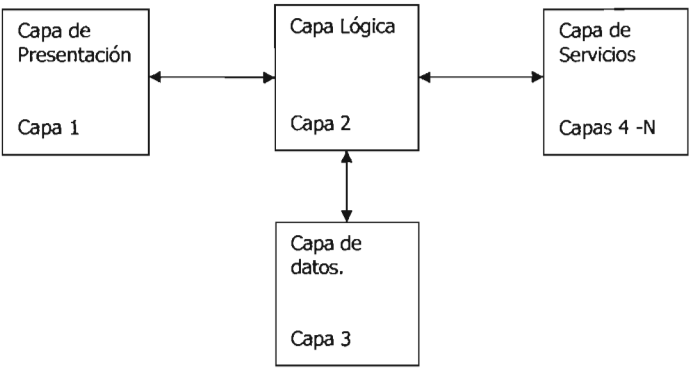


Figura 2.1 Diagrama de capas de una arquitectura.

2.2.1 La capa de presentación

Se refiere a los componentes responsables de crear y administrar la interfaz de usuario de una aplicación. Las tecnologías involucradas en esta capa son: Servidores WEB, procesamiento de plantillas y páginas dinámicas y navegadores WEB. El principal reto de esta capa es mantener vistas sincronizadas de la aplicación para diferentes usuarios. [Gabrick].

2.2.2 La capa lógica

Esta capa se refiere a los componentes responsables de implementar la funcionalidad de una aplicación. Los componentes deben administrar los datos y el estado de la aplicación mientras realiza las operaciones específicas soportadas por la aplicación. Es deseable en esta capa mantener un comportamiento adecuado y asegurar la integridad de los datos. [Gabrick].

2.2.3 La capa de datos.

Esta capa se refiere a los componentes que administran los datos internos de una aplicación. Usualmente esta capa se encuentra bajo control directo de un manejador de bases de datos relacional. No se manejan únicamente bases de datos relacionales, pueden administrarse los datos con un manejador de bases de datos de objetos ó con XML, sin embargo, para las aplicaciones empresariales la tecnología predominante es la relacional.
Es recomendable en esta capa incluir usos efectivos de los recursos del sistema, agrupación de conexiones a la base de datos y mejorar el rendimiento del sistema. [Gabrick].

2.2.4 La capa de servicios.

La capa de servicios se refiere a los servicios externos de la aplicación con la que un sistema colabora en una variedad de formas. El número de la capa de servicio puede ir desde la cuarta hasta la n. Las capas de servicios pueden ser muchas dependiendo si el sistema interactúa con otros sistemas o en su defecto puede no existir si el sistema es muy independiente. Se dice que puede extenderse a un número n porque los servicios pueden ser compartidos con otras empresas y de esta forma extender el número de capas lógicas de la aplicación. [Gabrick].

2.3 Componentes de la Arquitectura J2EE.

2.3.1 Panorama general de los sistemas WEB.

El objetivo principal de una arquitectura multicapa es compartir recursos entre clientes. Los servicios de software no son exclusivos de una máquina en particular, por lo que se desea que sean distribuidos.

La transmisión en tiempo real apareció con el sistema operativo UNIX, que contiene soporte para el Protocolo de Control de Transferencia/ Protocolo de Internet (TCP/IP). Este protocolo estándar indica como crear, traducir y controlar transmisiones entre máquinas en una red de computadoras. Contando con TCP/IP se desarrolló la llamada a procedimientos remotos (RPC), con el cual se compartían funciones entre programas en distintas máquinas y que estaban conectadas en la misma red. Para intercambiar estructuras de datos más complejas se creó XDR (eXternal Data Representation).

Al manejar objetos en lugar de procedimientos RPC no fue suficiente y se crearon dos nuevos protocolos para hacer más eficiente la comunicación remota. Estos fueron la Arquitectura común de gestor de peticiones a objetos (*Common Object Request Broken Architecture* ó CORBA) y el modelo común de objetos distribuidos (*Distributed Common Object Model* ó DCOM). CORBA desarrollado por la empresa Sun y Oracle entre otras, y DCOM desarrollado por Microsoft. Ambos protocolos fueron incompatibles pero se difundieron a gran escala.

Internet mostró una solución al conflicto entre ambos protocolos. Internet es un conjunto de estándares de protocolos abiertos que se centran sobre el protocolo de transporte de hipertexto (HTTP). Se logró un buen avance en el desarrollo de sistemas distribuidos, pero no se podrían compartir aún los servicios. Los servicios WEB comprenden una red de servicios donde los servicios son las piezas de construcción de software que se encuentran en una red, a partir de las cuales los programadores pueden crear sistemas distribuidos a gran escala.

Con la introducción de los servicios WEB se crearon tres nuevos estándares:

1. El lenguaje de descripción de servicios Web (*Web Services Description Language*-WSDL) que representa la descripción.
2. El descubrimiento e integración universales (*Universal Description, Discovery and Integration* – UDDI), y
3. el protocolo de arquitectura orientada a servicios (*Service Oriented Architecture Protocol* - SOAP).

Con todos estos elementos se podían implementar sistemas muy grandes y que compartieran sus servicios en Internet. [Deepak]

2.3.2 Orígenes de J2EE.

J2EE es un conjunto de especificaciones que señalan los lineamientos a los que una aplicación empresarial debe ajustarse para asegurar la interoperabilidad y algunas características deseables como la portabilidad y compatibilidad con sistemas existentes. Todas las aplicaciones J2EE se escriben utilizando Java como lenguaje de programación.

Contando con todos los estándares para desarrollo a gran escala y dado que Java se ha utilizado ampliamente en desarrollos de software, que actualmente están funcionando se abrió un nuevo campo que fue el desarrollo de aplicaciones WEB.
Al lenguaje Java se le agregaron bibliotecas de Servlets, para su aplicación en WEB.
También se crearon los Java Server Pages (JSP) y se enfocó fuertemente al sector de programadores acostumbrados al Lenguaje de Marcado de Hipertexto (*Hypertext Markup Language* – HMTL) y a los lenguajes ejecutados del lado del cliente (Scripts).

Se agregó también una biblioteca JDBC de Java para que los programadores pudieran acceder a distintos manejadores de bases de datos. El acceso a la base de datos es a través del lenguaje de consultas (SQL).

Con todo un conjunto de bibliotecas Java se difundió como un ***kit*** estándar de desarrollo llamado J2SE. (Java ™ 2 Platform, Standard Edition - J2SE™).

La empresa Sun convocó a la comunidad de programadores Java para desarrollar los estándares para las aplicaciones empresariales y se obtuvieron como resultado las bibliotecas Java para desarrollo empresarial, (Java ™ 2 Platform, Enterprise Edition - J2EE™), que finalmente fue una extensión a la plataforma J2SE existente.

2.3.3 Descripción de la arquitectura J2EE

La arquitectura J2EE es una arquitectura multicapa. Como se muestra en la Figura 2.2

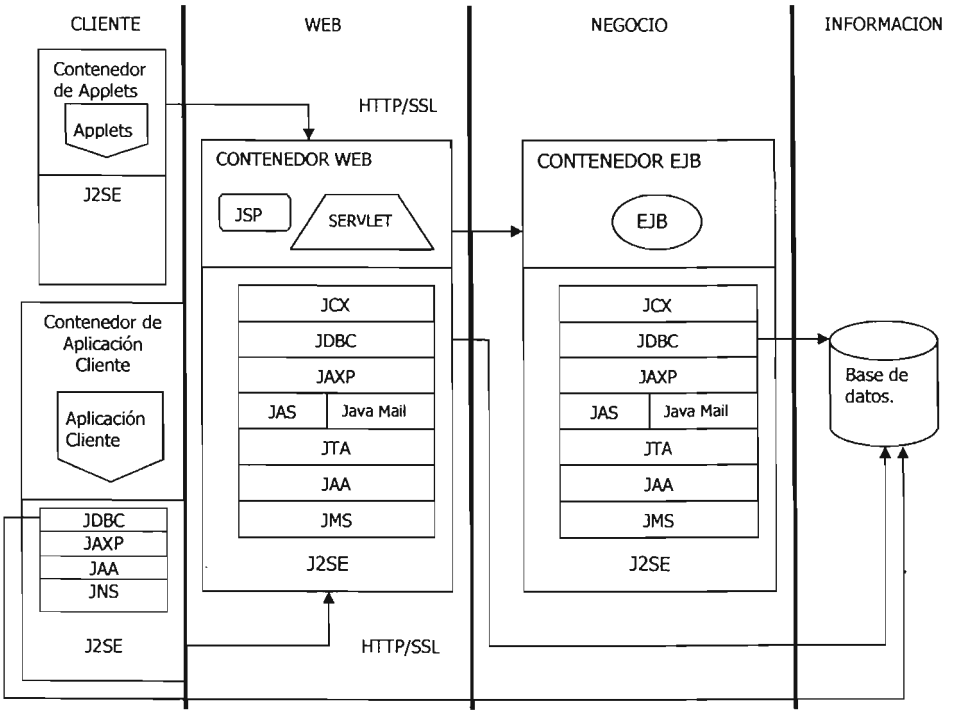


Figura 2.2 Arquitectura J2EE.

Se puede observar de la figura 2.2 que la base de todas las tecnologías utilizadas exceptuando la parte de los datos tiene como soporte el ***kit*** de desarrollo estándar de Java (J2SE). A continuación se describe la funcionalidad de cada una de las capas de la arquitectura J2EE.

2.3.3.1 Capa Cliente

Esta capa interactúa con el usuario y le proporciona información del sistema. La plataforma soporta diferentes tipos de clientes entre ellos, clientes HTML, Applets de Java y aplicaciones Java. En la figura 2.2 la parte que integra la capa cliente se indican como contenedor de applets y contenedor de aplicación cliente.

2.3.3.2 Capa Web

Genera la lógica de la presentación y acepta peticiones del usuario desde la capa cliente que normalmente son páginas HTML, Applets y otros clientes WEB. Basada en la petición del cliente genera la respuesta apropiada. Servlets y JSP implementan esta capa. En la figura 2.2 se muestra esta capa bajo el nombre de Contenedor WEB.

2.3.3.3 Capa de Negocio

Esta capa maneja la lógica de negocio de la aplicación. Provee las interfaces necesarias para los diferentes servicios que se proporcionan. Los servicios se implementan normalmente utilizando EJB[6], con soporte de un contenedor de EJB que facilitan el manejo del ciclo de vida del componente, su persistencia, las transacciones y la localización de recursos. Los EJB se pueden implementar en cualquier capa y tienen la propiedad de soportar diferentes protocolos por lo que el protocolo utilizado dependerá de la capa con la que se comunique el EJB. En la figura 2.2 se muestra esta capa bajo el nombre de contenedor EJB.

2.3.3.4 Capa de Información

Esta capa es responsable de los sistemas de información de la empresa incluyendo la base de datos, el procesamiento de las transacciones, los sistemas heredados y los sistemas de planeación de recursos. Este es el punto donde la plataforma J2EE se integra con aplicaciones que no pertenecen a la plataforma o con sistemas ya existentes. Aquí pueden utilizarse sistemas comerciales dado que la plataforma J2EE utiliza CORBA y conectores Java.
En la figura 2.2 se aprecia esta capa bajo el nombre de bases de datos.

2.3.4 Servicios estándar J2EE

La plataforma J2EE especifica los siguientes servicios estándar que todo producto debe soportar.

HTTP
: Protocolo estándar para comunicaciones WEB, a través de la biblioteca java.net.

HTTPS
: Es el protocolo http con un esquema de seguridad agregado.

---

[6] EJB – Enterprise Java Bean. Componente del lado del servidor que contiene la lógica de negocio de una aplicación y se aloja en un contenedor especial.

JDBC
: Biblioteca estándar para acceso a bases de datos.

JavaMail
: Provee un marco de trabajo para construir aplicaciones de correo y mensajes en Java.

Java Activation FrameWork (JAF)
: Se utiliza para la activación de un marco de trabajo como JavaMail.

RMI/IIOP
: Es un protocolo que habilita la invocación de métodos remotos (RMI) para combinarlos con el protocolo CORBA/IIOP para comunicarse con clientes CORBA.

JavaIDL (Java Interface Definition Language)
: Un servicio que incorpora CORBA en una plataforma Java para proveer interoperabilidad usando el estándar IDL definido por la OMG (*Object Management Group*).

JTA (Java Transaction API)
: Un conjunto de bibliotecas que permiten el manejo de transacciones, para comenzar, ejecutar y deshacer transacciones.

JMS
: Permite habilitar sistemas de mensajes de punto a punto o del tipo publicación/suscripción.

JNDI (Java Naming and Directory Interface)
: Interface unificada para acceder a diferentes tipos de servicios de directorio y de nombres. Se utiliza para registrar y buscar componentes de negocio.

Cada servicio cuenta con una implementación a través de bibliotecas agregadas al lenguaje Java para poder utilizarlas en las aplicaciones.

2.3.5 Distribución de aplicaciones J2EE.

Con la aceptación del lenguaje Java para desarrollo de aplicaciones grandes y el incremento de los sistemas Web, se crearon nuevos productos para dar soporte a muchos usuarios que solicitan servicios al mismo tiempo. Los componentes capaces de prestar estos servicios se llaman servidores de aplicaciones. Los servidores de aplicaciones son el destino final de una aplicación con J2EE.

Dado que J2EE es un conjunto de especificaciones y por sí misma representa documentación base a la que una aplicación debe apegarse, se necesita un soporte de productos que puedan ayudar a que un problema pueda tener su implementación.
Los servidores de aplicaciones fueron una respuesta de los proveedores de servicios e infraestructura para la red a la demanda de las empresas por nuevos servicios y prestaciones.

Estos servidores prestaron los servicios básicos de la infraestructura requerida para desarrollar e instalar aplicaciones empresariales multicapa.Estos servidores ofrecen algunos beneficios entre los que destaca que las empresas no tenían que desarrollar su infraestructura propietaria para soportar sus aplicaciones (el desarrollo de estas plataformas era lento y muy caro, normalmente no se cubrían las expectativas).

Existen diferentes servidores de aplicaciones en el mercado, los hay de distribución gratuita como Jboss de Jboos.org [1], JonAs de Evidian/Object [2] Web y versiones comerciales como WebLogic de BEA Systems [3] y WebSphere de IBM [4]. Cada servidor de aplicaciones tiene sus ventajas y desventajas, porque no hay estándares para desarrollarlos. Existen por lo tanto ciertas variaciones en la funcionalidad que cada servidor provee, sin embargo, si el desarrollo se ajusta a la especificación J2EE la portabilidad está garantizada.

2.4 E++ para el desarrollo de aplicaciones J2EE.

E++ es un lenguaje de patrones que describe el proceso para crear una aplicación con J2EE. Dado que J2EE es un conjunto de especificaciones muy completo se hace alusión a los patrones de software para organizar una aplicación.

Una colección de patrones puede ser útil para diseñar una aplicación, sin embargo se requiere experiencia para elegir un conjunto adecuado de patrones aplicables a la aplicación particular.

E++ es un lenguaje de patrones, provee reglas para que los patrones de diseño puedan trabajar en conjunto para resolver un grupo de problemas relacionados.

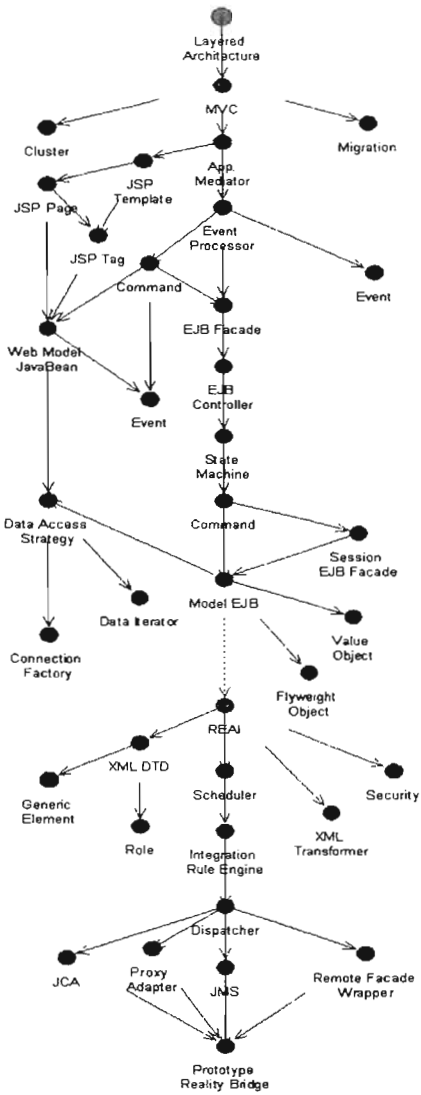


Figura 2.3 Lenguaje de patrones E++.

Construir o migrar una aplicación a una aplicación J2EE puede ser fácil siguiendo la estructura de árbol que plantea el lenguaje de patrones E++, desde la raíz hasta las hojas.

Por ejemplo puede utilizarse para decidir la migración de una aplicación de dos capas a una aplicación de n capas. La aplicación final debe tener características de modularidad, reusabilidad, extensibilidad, portabilidad, consistencia y escalabilidad. El objetivo es construir una aplicación de alto nivel utilizando el estándar J2EE.

El problema se traduce en elegir de un conjunto de patrones GoF, los adecuados a la aplicación y aplicarlos de manera coherente.
E++ hace más sencillo coordinar patrones en proyectos reales, contiene patrones de diseño, y arquitectura que proveen una referencia para desarrollar una aplicación completa.

E++ expone un esquema que permite seleccionar los patrones adecuados a una aplicación y asegurar la interoperabilidad entre ellos. El lenguaje de patrones provee una organización de las secuencias de decisiones tomadas en el marco de trabajo y estas decisiones deben entenderse en el contexto del desarrollador y pueden agregarse nuevas estructuras de colaboración.
El trabajo de GoF sobre patrones se encargó de clasificarlos en plantillas[7]. E++ extendió ésta clasificación a relaciones entre ellos y ésta relación se toma como un método de diseño utilizando la arquitectura J2EE.

La estructura de árbol que plantea E++, se muestra en la Figura 2.3.

Cada círculo del árbol denota un patrón y las flechas representan las relaciones entre patrones. Este lenguaje de patrones sugiere un inicio utilizando una arquitectura por capas si es necesario y después puede adoptarse el patrón MVC como arquitectura general. Este lenguaje de patrones utiliza a J2EE para describir el flujo de desarrollo de una aplicación por lo que cada uno de los nodos puede ser una implementación del patrón utilizando a Java como lenguaje de programación.

2.5 El patrón de arquitectura MVC

El desarrollo de una aplicación es complejo ya que debe dar servicio a muchos clientes de tipos diversos en forma simultánea a través de una estructura distribuida. La aplicación debe ser escalable de forma que tenga un rendimiento aceptable sin importar el aumento en el número de clientes que la utilizan.

La mejor práctica para la distribución de la funcionalidad de la aplicación es utilizar el patrón Modelo-Vista-Controlador (MVC).

[7] En el capítulo 1 se describieron los patrones y el esquema de clasificación en el que fueron agrupados por GoF.

La estrategia MVC divide la aplicación en tres grandes componentes: Modelo, Vista y Controlador, como se muestra en la figura 2.4

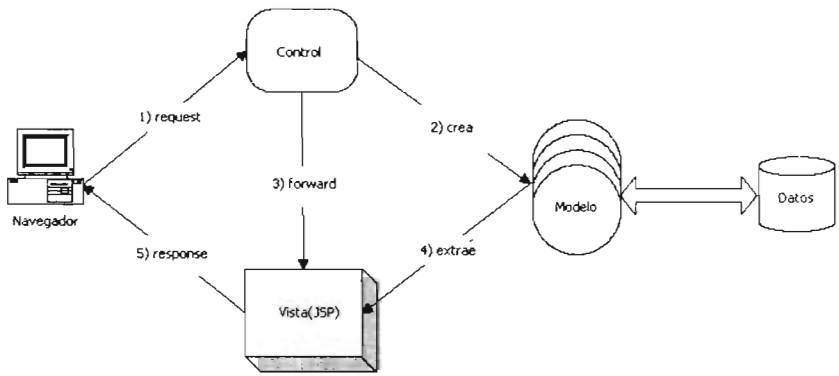


Figura 2.4 Descomposición en capas del patrón MVC. [Ford].

Para el desarrollo de la herramienta HIM se toma como base el patrón de arquitectura MVC que se describe a continuación en su formato de plantilla estándar.

Plantilla del patrón MVC.

PATRÓN MVC (Model, View, Controller.)

Tipo: De comportamiento
Nivel: componente, arquitectura

Propósito:
Divide un componente o un subsistema en tres partes – modelo, vista y controlador – facilitando la modificación o personalización de cada parte.

Introducción:

Se tiene información que se desea mostrar al usuario de distintas formas. Se necesitan algunas clases que contengan la información a manejar, se pueden introducir varios métodos para que cualquier cambio en la interfaz desencadene una llamada a una clase para actualizar la información del negocio.
Puede manejarse el proceso a través de una sola clase pero el código es más complejo y difícil de mantener. El código no es fácilmente ampliable.
La estrategia es dividir la aplicación en tres partes funcionales. Una clase que represente la información visual, otra que maneje la información de negocio y otra que controle la asignación entre la interfaz gráfica y la información del negocio.

Aplicabilidad:

MVC es útil cuando hay algún componente o subsistema que tiene alguna de las siguientes características:

- Es posible ver el componente o subsistema de distintas formas. La representación interna podría ser completamente diferente de la representación en la pantalla.
- Hay diferentes tipos de comportamiento, lo cual quiere decir que múltiples fuentes pueden invocar el comportamiento en el mismo componente pero el comportamiento puede ser muy diferente.
- Se utiliza un comportamiento o representación que se modifica conforme se usa el componente.
- A menudo quiere tener la capacidad de adaptar o reutilizar cierto componente bajo distintas circunstancias con un mínimo de trabajo extra.

Descripción:

Un problema con el que siempre se enfrentan los desarrolladores que utilizan la orientación a objetos, es cómo crear apropiadamente componentes genéricos. El problema es más evidente cuando un componente es complejo o flexible en su uso.
Considere una tabla. El concepto tabla puede ser aplicado de muchas formas dependiendo de las necesidades de una aplicación. Puede utilizarse una tabla como una aproximación para almacenar datos como una estructura lógica formada con celdas, filas y columnas. Sin embargo, hay muchas maneras de controlar lo que se almacena y cómo se representa el almacenamiento.

Una tabla también puede estar presentada de distintas formas en una aplicación. Podría ser mostrada visualmente como una cuadrícula, un gráfico o un diagrama. O podría no tener representación gráfica.

Cuando hay tantas formas de utilizar un control, como es el caso de una tabla, se presenta un dilema. Claramente, la idea sería tener una forma de reutilización, para no tener que empezar a escribir código desde el principio para cada tabla nueva. Al mismo tiempo, es difícil imaginar como se puede crear un componente simple como éste para que sea reutilizable. Una implementación que fuera tan genérica requeriría un gran esfuerzo para modificarla cada vez que fuera necesario, eliminando muchas de las ventajas de la reutilización.
MVC ofrece una alternativa eficiente. Define un elemento complejo en términos de tres subunidades lógicas:

Modelo: el estado del elemento, se ocupa de los cambios de estado.
Vista: la representación del elemento (visual o no visual).
Controlador: controla la funcionalidad del elemento, asignando las acciones a la vista según su impacto en el modelo.

Implementación

En la figura 2.5 se muestra el diagrama de componentes del patrón MVC.

Se muestra el patrón MVC a través de un diagrama de componentes con las siguientes características:

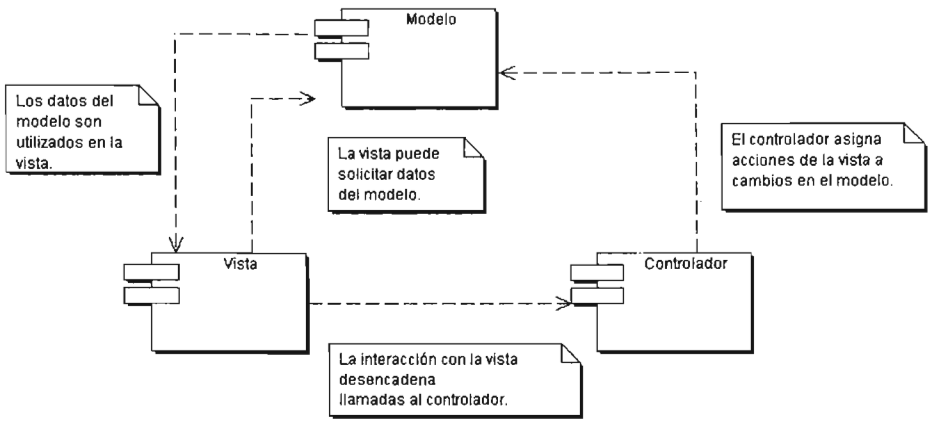


Figura 2.5 Diagrama de componentes del patrón MVC.

Model (Modelo): es el componente que tiene una o más clases e interfaces que son los responsables de mantener los datos. El estado del modelo es mantenido en los atributos y la implementación de los métodos. Para poder enviar notificaciones de los cambios que se producen en el modelo a los componentes de la vista, el modelo mantiene una referencia a todas las vistas registradas, estableciendo una relación con el patrón *Observer*. Cuando ocurre un cambio, cada componente vista que ha sido registrado debe recibir una notificación.

View (Vista): Las clases e interfaces de la vista proporcionan una representación de los datos en el componente del modelo. La vista puede consistir en componentes visuales, aunque no es necesario. Una vista debe ser registrada con el modelo para ser notificada de los cambios en los datos del modelo. Cuando se recibe una notificación de un cambio, la vista es responsable de determinar si hay que representar este cambio y cómo hacerlo. El patrón *Observer* relaciona las vistas con el modelo para mantenerlas actualizadas.
El componente vista también mantiene una referencia al modelo para recuperar los datos de él, pero sólo puede recuperar información, no modificarla. La vista también puede ser utilizada para interpretar el controlador, pero las peticiones de cambio siempre se redirigen a un componente controlador, por lo que a vista necesita mantener una referencia a uno o más controladores.

Controller (Controlador): este componente gestiona los cambios en el modelo. Mantiene una referencia al componente modelo responsable de ejecutar los cambios, mientras que el controlador llama a uno o más métodos de actualización. Las peticiones de cambio pueden provenir de un componente vista.

Ventajas e inconvenientes

MVC proporciona una forma excelente de hacer que un elemento sea flexible y adaptable a distintas situaciones. La flexibilidad puede ser utilizada de una forma estática y dinámica. Las nuevas clases vista o controlador pueden ser introducidos en la aplicación (estáticas), y los objetos vista o controlador pueden ser modificados en la aplicación en tiempo de ejecución.
Usualmente el mayor desafío de MVC es determinar la verdadera representación base; es decir, para definir un conjunto apropiado de interfaces en el modelo, las vistas y los controladores. A menudo un elemento MVC se desarrolla para satisfacer un conjunto específico de necesidades, como en la mayor parte del software, por lo que se necesita visión y un análisis cuidadoso para implementar el elemento, de forma que no se impongan restricciones específicas de la aplicación sobre él.

Variaciones del patrón:

Las variaciones de MVC suelen girar en torno a las diferentes opciones de implementación para las vistas.

Modelo *push* frente a modelo *pull*: en MVC se puede implementar una o más formas de que el modelo envíe las actualizaciones a su vista (o vistas), o que una vista pueda recuperar información del modelo conforme la necesita. La elección afecta a cómo se implementan las relaciones en el sistema.

Múltiples objetivos vista: Un modelo puede proporcionar información a más de una vista. Esto resulta particularmente útil en algunas implementaciones de interfaces gráficas, porque los mismos datos deben llevar algunas veces a distintas representaciones.

Vistas "ver pero no tocar": no todas las vistas necesitan un controlador. Algunas proporcionarán sólo una representación visual de los datos del modelo, pero no soportan ningún cambio en el modelo desde esa vista.

Patrones relacionados:

Observer (Observador): El patrón MVC suele utilizar el patrón *Observer* para gestionar la comunicación. Esto se hace normalmente para las siguientes partes del sistema:
Entre la vista y el controlador, de forma que un cambio en la vista dispara una respuesta en el controlador.
Entre el modelo y la vista, de forma que la vista recibe una notificación de un cambio en el modelo.

Strategy (Estrategia): el controlador a menudo es implementado con éste patrón para simplificar el cambio en los controladores. [Stelting].

En el próximo capítulo se dará un enfoque más especializado a la capa de presentación de J2EE describiendo cada uno de los patrones a utilizar en el desarrollo de la herramienta HIM en la parte de las vistas.

III. Arquitectura de la herramienta HIM y patrones de la capa de presentación.

3.0 Introducción.

En el capítulo anterior se mostró un esquema general de la Arquitectura J2EE y la tecnología que implementa cada una de las capas que la forman.

En este capítulo se muestran algunos de los diferentes marcos de trabajo que se pueden utilizar en un desarrollo con J2EE. Se muestra la arquitectura general que se utiliza en el desarrollo de la herramienta HIM y la descripción de los patrones de la capa de presentación propuestos. También se muestra de manera breve el marco de trabajo *Struts* que permite la implementación de algunos patrones de la arquitectura J2EE.

3.1 Marcos de Trabajo para desarrollos con J2EE.

Contando con la descripción de una arquitectura para una aplicación Java (J2EE), un lenguaje de referencia para el desarrollo de proyectos (E++) y la descripción de un patrón general para la arquitectura (MVC) se requiere definir con más detalle la arquitectura para la herramienta HIM, especificando los patrones que pueden utilizarse en la implementación.

La empresa IBM[8] propone varios esquemas para el desarrollo de aplicaciones J2EE. Se tienen las siguientes opciones para el desarrollo de aplicaciones:

1. Arreglo de patrones[9] para Contenedor *Web*.
2. Arreglo de patrones *Command*.
3. Arreglo de patrones para *bean* de sesión.
4. Arreglo de patrones para *bean* de entidad. [Hagemo]

Los primeros dos arreglos están diseñados para aplicaciones *Web* simples que utilizan Servlets y JSPs que se instalan en un contenedor *Web*.

Los dos últimos arreglos están diseñados para aplicaciones *Web* que utilizan contenedores *Web* y de EJBs.

---

[8] Los autores describen diferentes arquitecturas que pueden implementarse con J2EE en su artículo "*Creating a Framework - J2EE pattern frameworks provide template for flexible and modular architecture*"

9 Un arreglo de patrones en este contexto representa un *framework* o un arreglo adaptable y extensible para obtener una aplicación específica.

A continuación se describen cada uno de los marcos propuestos por la empresa IBM y finalmente se genera un marco en el que puede implementarse la herramienta HIM.

3.1.1 Arreglo de patrones para Contenedor Web.

Este arreglo se utiliza para acceso a una base de datos desde un contenedor Web. Existe un primer punto de contacto para todas las peticiones del cliente. Una vez que las peticiones se reciben y filtran se envían al *Front Controller* que es el despachador central de la aplicación. El *Front Controller* dirige las peticiones al *View Helper* adecuado. El *View Helper* es un conjunto de funciones que recuperan información específica a través de los DAOs. [Hagemo]

El arreglo se muestra en la Figura 3.1

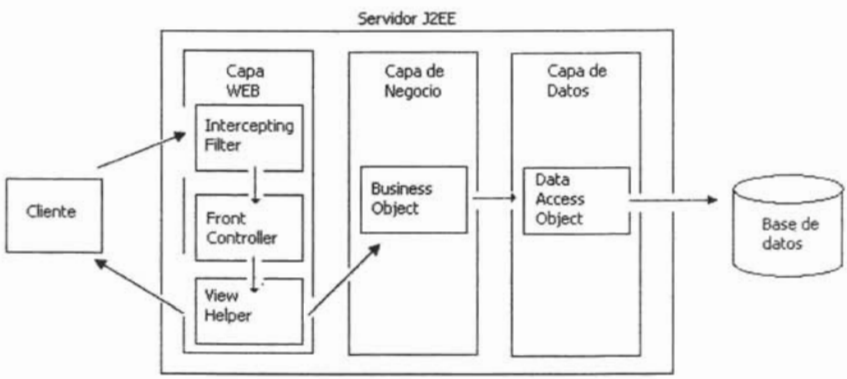


Figura 3.1 Arreglo de Patrones para Contenedor Web. [Hagemo].

3.1.2 Arreglo de patrones Command.

Es similar al arreglo anterior pero utiliza el patrón *Command* como se muestra en la Figura 3.2. El patrón *Command* provee desacoplamiento que puede hacer una aplicación más sencilla y agregarle funcionalidad en un futuro.

El controlador recibe el comando, obtiene los resultados y los envía al *Composite View* que integra la vista. [Hagemo]

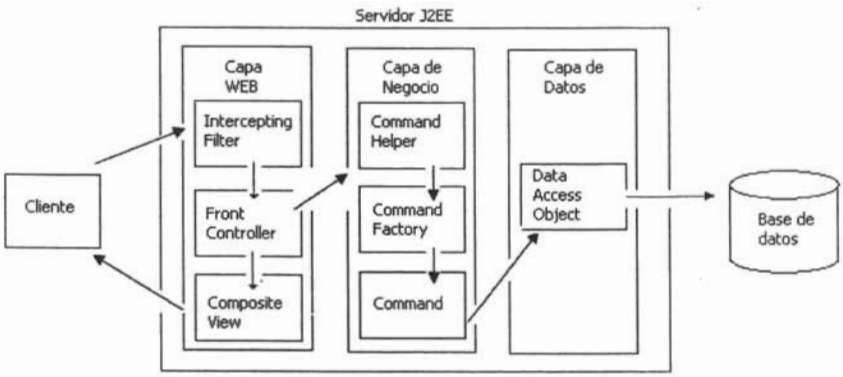


Figura 3.2 Arreglo de patrones *Command*. [Hagemo].

3.1.3 Arreglo de patrones para bean de sesión.

El tercer arreglo agrega el patrón *Session bean* al contenedor Web como se muestra en la Figura 3.3. Se agrega más complejidad por los componentes empresariales pero se pueden añadir componentes adicionales con mayor facilidad.

El patrón *View Helper* llama al patrón *Business Delegate* para delegar las peticiones al *Session Facade*. El patrón *Business Delegate* es responsable de buscar el patrón *Session Facade* usando el patrón *Service Locator*. Los datos se transfieren desde el *Business Delegate* al *Session Facade* utilizando el patrón *Value Object*.
El *Session Facade* llama al DAO para obtener los datos que se solicitan.

Este marco de trabajo requiere un contenedor de EJBs para su completa implementación. [Hagemo].

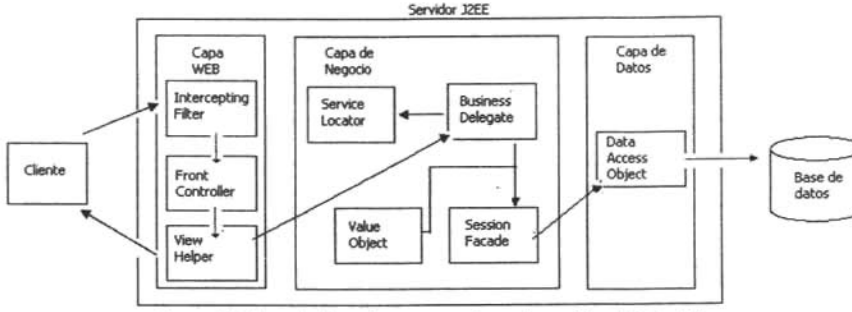


Figura 3.3 Arreglo de patrones para *bean* de sesión. [Hagemo].

3.1.4 Arreglo de patrones para bean de entidad

Este marco de trabajo agrega el patrón de *beans* de entidad al marco anterior. Pueden utilizarse cuando existen *Entity Beans* en una aplicación.[10] Se recomienda que los beans de entidad sean administrados por el contenedor de EJB. Este proceso elimina el SQL del código Java y los coloca en archivos XML. La codificación de SQL en archivos XML toma tiempo aprender y entender. Este marco agrega el patrón *Value List Handler* que es importante cuando se manejan registros muy grandes de la base de datos. El patrón *Composite Entity* y el *Value List Handler* mejoran el rendimiento de la aplicación. El patrón *Session Facade* llama a los DAOs o a los diferentes *Composite Entities*. Cuando se utilizan bases de datos grandes, el marco regresa un conjunto de datos pequeños iterando a través del *Value List Handler*. [Hagemo].

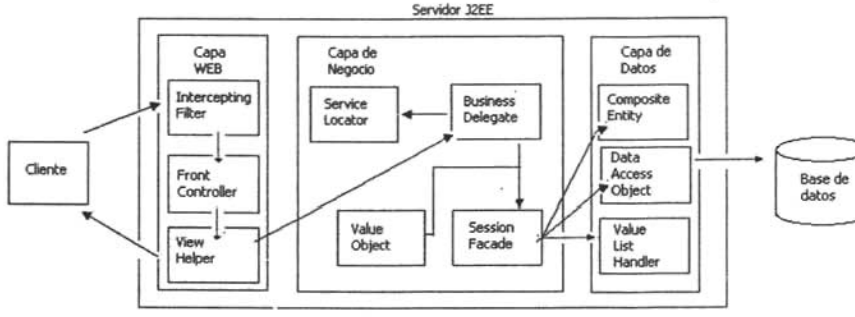


Figura 3.4 Arreglo de patrones para *bean* de entidad. [Hagemo].

[10] *Entity Beans* son componentes empresariales Java (EJB) que permiten la manipulación a una fuente de datos, comúnmente se utilizan enlaces a bases de datos relacionales.

Dado que los dos últimos patrones utilizan un contenedor de EJBs, se dejan como una opción futura en el diseño de la herramienta HIM. En una primera versión se pueden utilizar los dos primeros marcos de trabajo para facilitar el desarrollo[11].
Como un marco de trabajo para el desarrollo de la herramienta HIM se utilizará el modelo mostrado en la Figura 3.5.

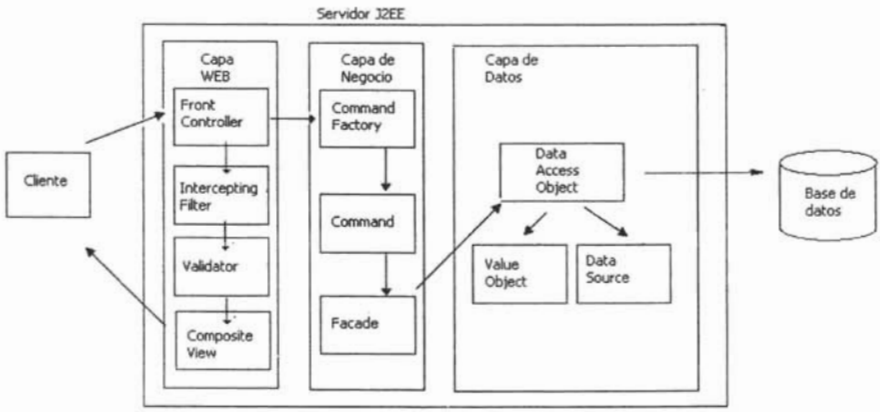


Figura 3.5 Arquitectura elegida para la herramienta HIM.

Se realizarán peticiones del cliente y cada una de las peticiones entrantes se procesarán en un Controlador central que se implementa en el *Front Controller*.
Se utilizarán dos patrones para verificar los datos:

a) Uno de ellos el patrón *Intercepting Filter* se encarga de verificar los roles y permisos, y
b) El patrón *Validator* se encarga de verificar los tipos de datos que recibe en cada petición desde el cliente.

Una vez que se reciben las peticiones se procesan en la capa de negocio aplicando la lógica correspondiente, la fábrica de comandos (*Command Factory*) crea las instancias de los comandos invocados y se comunica con la capa de datos DAOs a través de una fachada (*Facade*), que regresa un grupo de objetos (*Value Object*) o grupo de registros de la base de datos (*Data Source*) como resultado de la petición.
De acuerdo a la respuesta generada se construye la vista adecuada a través del patrón *Composite View*.

Esta arquitectura final es una variante del segundo modelo (Arreglo *Command*) propuesto por la empresa IBM y se adopta porque la herramienta se manejará como una vía de difusión del modelo

[11] En esta primera versión se utiliza Tomcat 5.0 un contenedor de *Servlets* y *JSP* pero la arquitectura elegida puede ser extendida utilizando un contenedor de componentes empresariales EJB.

de procesos MoProSoft que será una aplicación web, en ésta primera versión no se manejan componentes empresariales. Antes de aplicar los filtros debe permitir la consulta de algunas páginas que contienen descripciones de algunos conceptos relacionados con el modelo de procesos por lo que debe aceptar cualquier petición. Cuando se requiera el acceso para manipular los productos generados en las actividades se aplicarán los filtros necesarios para verificar los permisos de cada usuario y se le presentará el marco de trabajo adecuado.

3.2 Patrones de la capa de presentación

Tomando como base la arquitectura de la figura 3.5 podemos obtener un subconjunto de esa arquitectura para poder trabajar y desarrollar la capa de presentación de la herramienta HIM, esta capa de presentación debe interactuar con la capa de Negocio que controla las acciones que un usuario puede realizar. Cada una de las peticiones del cliente debe entrar a través de una página Web y se envía a la capa de Negocio que genera respuestas como resultado de esas peticiones.

Los resultados que surgen de las peticiones deben reflejarse en la capa de presentación a fin de que el usuario pueda obtener suficiente información para conocer el estado del sistema. El control debe interactuar a su vez con la capa de Datos para recuperar la información necesaria.

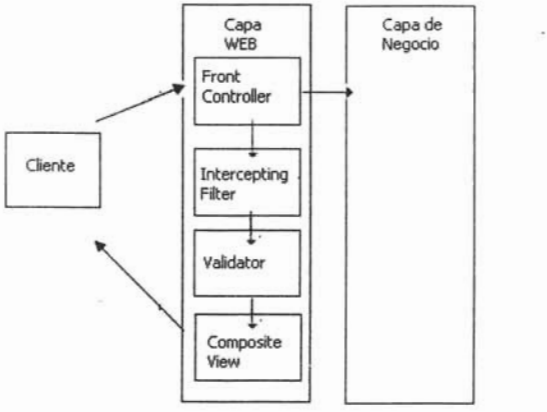


Figura 3.6 Patrones de la capa de presentación.

La tabla de la figura 3.7 resume los patrones que se utilizarán en la capa de presentación.

Nombre del Patrón	Resumen
Front Controller	Provee un controlador centralizado para administrar el manejo de peticiones.
Intercepting Filter	Facilita el PRE y POST procesamiento de una petición.
View Helper	Encapsula la lógica que no está relacionada con la vista en componentes *Helpers*.
Composite View	Crea una vista agregando componentes atómicos.
Validator	Permite la verificación de los tipos de datos y del formato de los mismos.

Figura 3.7 Patrones de la capa de Presentación.

3.2.1 Intercepting Filter

Contexto

El mecanismo de manejo de peticiones de la capa de presentación recibe muchas peticiones de diferentes tipos, mismas que requieren un procesamiento diferente. Algunas de estas peticiones son simplemente redirigidas al manejador correspondiente, otras deben ser modificadas o verificadas antes de procesarse.

Problema

Se requiere pre y post procesamiento de las peticiones y respuestas de un cliente Web. Cuando se hace una petición desde un cliente Web, frecuentemente deben pasarse varias pruebas antes de la etapa de procesamiento principal, por ejemplo:

1. ¿Se ha autenticado al cliente?
2. ¿La dirección IP del cliente proviene de una red confiable?
3. ¿Se viola algún procedimiento en la petición del cliente?
4. ¿El tipo de navegador del cliente es adecuado?
5. ¿Qué tipo de codificación utiliza el cliente para enviar sus datos?

Algunas de esas verificaciones son pruebas que afirman o niegan una situación y con base en las respuestas se determina si el procesamiento continuará. Otras verificaciones convierten los datos de entrada a un formato adecuado para su procesamiento.
La solución clásica consiste de una serie de pruebas condicionales tomando cualquier prueba errónea como base para abortar la petición. Las sentencias *if/else* anidadas son una estrategia

común pero esta práctica dirige a un estilo de programación de copiar y pegar ya que el flujo del filtrado y la acción de los filtros se compilan en la aplicación.

La clave para solucionar el problema de una forma flexible es tener un mecanismo simple para agregar y eliminar componentes de procesamiento, en la que un componente realice completamente una acción de filtrado.

Fuerzas

- El procesamiento común como la verificación de la codificación de los datos o la información de acceso al sistema se realiza en cada petición.
- Se requiere centralizar la lógica común.
- Debe ser fácil agregar o quitar componentes sin afectar componentes existentes, de forma que puedan utilizarse en una variedad de combinaciones como:
    - Autenticación de ingreso al sistema
    - Depuración y transformación de la salida para un cliente específico.
    - Manipulación del esquema de codificación de la entrada.

Solución

Crea filtros para procesar servicios comunes de una forma estándar sin requerir cambios al código de procesamiento de transacciones. El filtro intercepta las peticiones entrantes y las respuestas generadas. Existe la capacidad de agregar o remover filtros sin afectar a los filtros existentes.
Se puede decorar el procesamiento principal con una variedad de servicios comunes como la seguridad, la bitácora, depuración, etc. Estos filtros son componentes que son independientes de la aplicación principal y se deben agregar o quitar de forma declarativa. **[Cruz].**
Por ejemplo, un archivo de configuración para la instalación puede modificarse para configurar una cadena de filtros. El mismo archivo de configuración puede incluir un mapeo de *URL´s* a esta cadena de filtros. Cuando un cliente solicita un recurso que coincide con la *URL* configurada, los filtros en la cadena son procesadas en orden antes que se invoque al recurso solicitado.

Estructura.

La figura 3.8 representa el diagrama de clases del patrón *Intercepting Filter*.

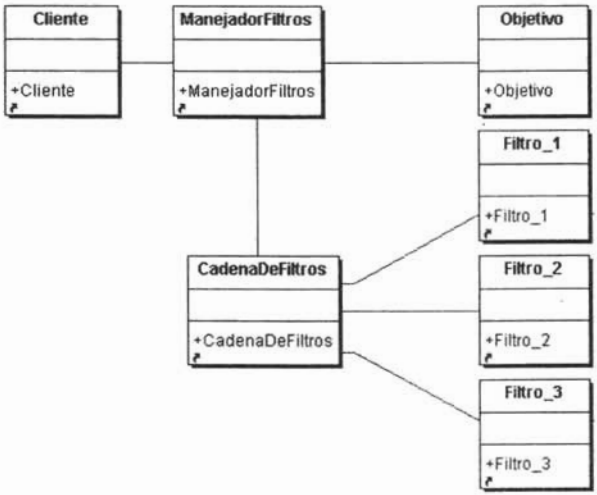


Figura 3.8

Participantes y Responsabilidades

La figura 3.9 representa el diagrama de secuencia del patrón *Intercepting Filter*.

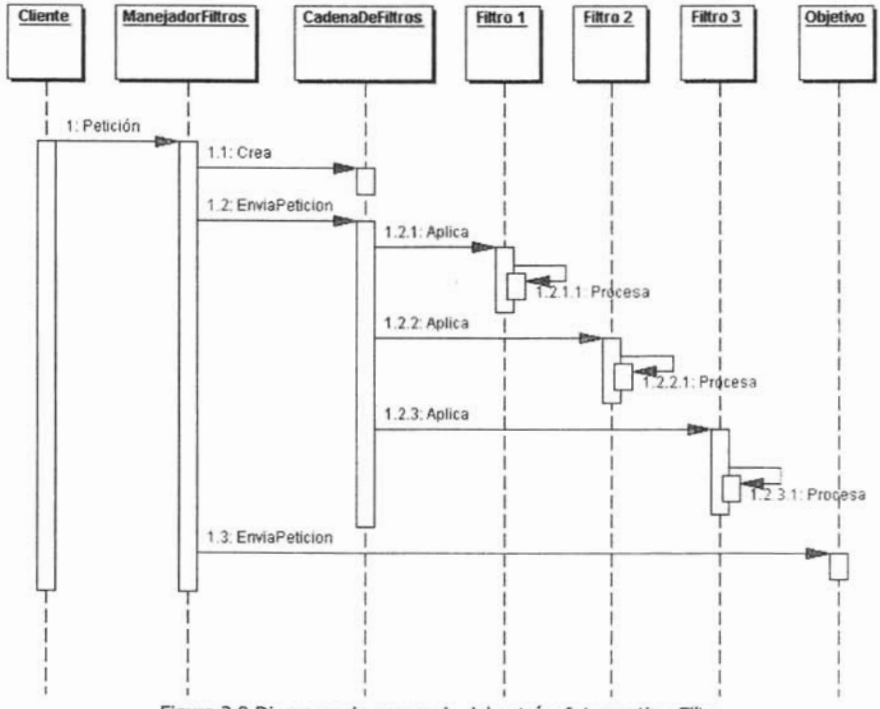


Figura 3.9 Diagrama de secuencia del patrón *Intercepting Filter*.

Patrones Relacionados

Front Controller
El controlador resuelve problemas similares pero es más eficiente para el manejo de procesamiento central.
Decorator
El patrón *Intercepting Filter* está relacionado con el patrón *Decorator*, el cual provee capacidad para insertar dinámicamente elementos con formato definido.
Template Method
Se utiliza para implementar una estrategia para filtro de plantillas.
Interceptor
Permite agregar servicios de manera transparente y que los servicios se disparen automáticamente. [Deepak].

3.2.2 *Front Controller*

Contexto

El sistema maneja peticiones Web.

El procesamiento de la presentación requiere manejo, navegación y entrega compleja de vistas.

Problema

El sistema requiere un punto de acceso centralizado para el manejo de las peticiones para soportar la integración de los servicios, recuperación del contenido, administración de las vistas y navegación.

Cuando el usuario accede a la vista directamente sin un mecanismo de acceso centralizado pueden ocurrir dos problemas:

1. Cada vista necesita proveer sus propios recursos del sistema, esto dirige a duplicación de código.
2. La navegación se deja a las vistas.

Fuerzas

- El procesamiento común de servicios del sistema se completa por petición. Por ejemplo, el servicio de seguridad termina las verificaciones de autenticación y autorización.
- La lógica se maneja mejor desde un punto central en lugar de ser copiada en numerosas vistas.
- Existen puntos de decisión con respecto a la recuperación y manipulación de datos.
- Muchas vistas se utilizan para responder a peticiones similares.
- Un punto central de contacto para el manejo de peticiones puede ser útil, por ejemplo para controlar y registrar el progreso del usuario a través del sitio.
- Los servicios del sistema y la administración de las vistas son relativamente sofisticadas.

Solución

Usar un controlador como el punto de contacto inicial para el manejo de peticiones. El controlador administra el manejo de las peticiones, incluyendo la invocación de los servicios de seguridad como la autenticación y autorización, delegación de procesamiento de reglas de negocio, administración de la elección de una vista apropiada, el manejo de los errores y estrategias de selección de creación de contenido.

El Controlador (*Controller*) provee un punto de entrada centralizado que controla y administra las peticiones Web.

Centralizando el control se reduce la cantidad de código Java, el uso de *scripts* y código dinámico en el JSP.

Centralizando el control en el controlador y reduciendo la lógica de negocio en las vistas se promueve la reutilización del código en las peticiones. Es una práctica preferible a la alternativa de involucrar código en las vistas, porque esto puede causar más errores debido al uso de esquemas como copiar y pegar.

Típicamente un controlador trabaja en coordinación con un despachador (*Dispatcher*). Un despachador es responsable de administrar la vista y la navegación. Un despachador puede elegir la siguiente Vista a presentar al usuario y provee un esquema para organizar los accesos a este recurso.

El patrón *Front Controller* sugiere centralizar el manejo de las peticiones, pero no limita el número de controladores. Puede haber varios *Front Controllers* en un sistema, cada uno de ellos relacionando servicios distintos.

Estructura

El diagrama de la Figura 3.10 representa el diagrama de clases del patrón *Front Controller*:

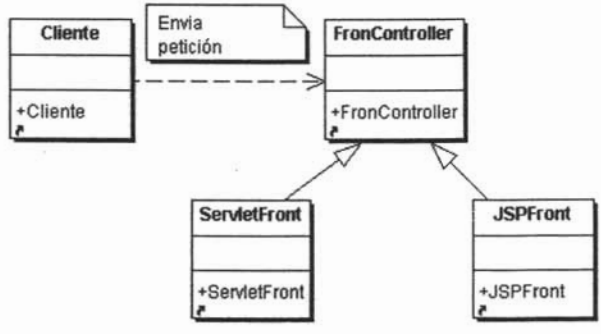


Figura 3.10 Diagrama de clases del patrón *Front Controller*.

Participantes y responsabilidades.

En la Figura 3.11 se representa el diagrama de secuencia del patrón *Front Controller*. Muestra como el *Front Controller* maneja una petición.

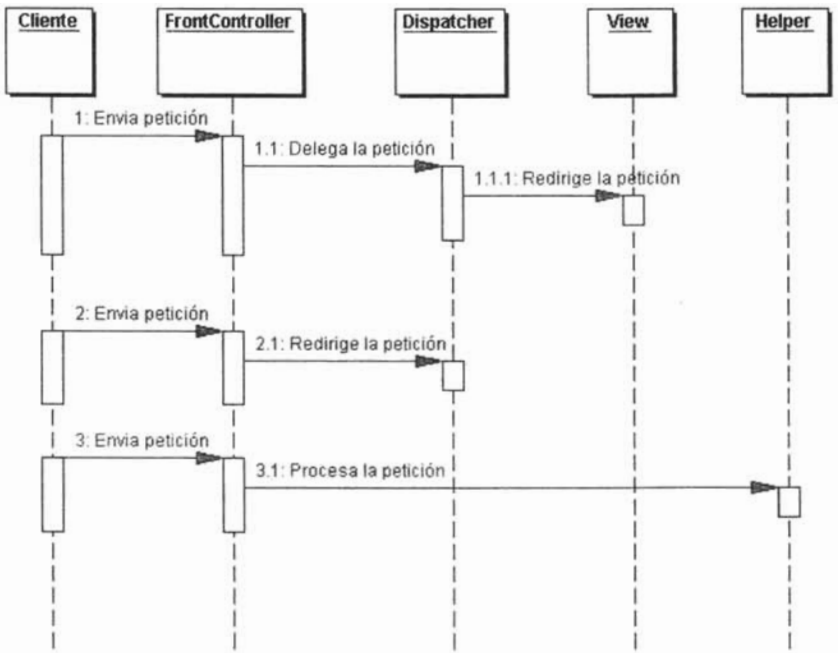


Figura 3.11 Diagrama de secuencia del patrón *Front Controller*.

Controller

Es el punto de contacto inicial para manejar todas las peticiones al sistema. Típicamente interactúa con el *Dispatcher* (Cuando se usa con el patrón Service *To Worker*).

Puede delegar completamente a un *Helper* la autenticación y la autorización de un usuario.

Dispatcher

Es responsable del manejo de las vistas y la navegación, administrando las opciones de la siguiente vista que se presentará al usuario y provee un mecanismo para el control de todas las vistas.

Un *Dispatcher* puede encapsularse dentro de un controlador o puede trabajar en conjunto como una parte separada. El *Dispatcher* puede proveer un mecanismo estático o dinámico para el manejo de las vistas.

El *Dispatcher* usará el objeto *Request Dispatcher* (que se encuentra en la especificación del *Servlet*) y encapsulará algún procesamiento adicional.

Helper

Un *Helper* es responsable de ayudar a una Vista o un Controlador a que finalice su tarea. Los *Helpers* tienen muchas responsabilidades como obtener datos o manipular esos datos para su presentación en las vistas.

Una vista puede trabajar con muchos *Helpers*, que se implementan típicamente como *JavaBeans* y etiquetas dinámicas. Además, un *helper* puede representar un objeto *Command*, un *Delegate*, o un transformador de datos (XSLT) de XML que se utiliza con una hoja de estilo para adaptar y convertir el modelo a un formato apropiado.

View

Representa y despliega información al cliente. La información que se utiliza en una pantalla se regresa al modelo. Los *helpers* apoyan a las vistas encapsulando y adaptando un modelo para su uso en una pantalla.

Patrones relacionados.

Patrón *View Helper.*
El patrón *Front Controller* se combina con el *View Helper* para proveer a los contenedores la capacidad de manejar la lógica de negocio fuera de las vistas y proveer un punto central de control y redirección de las vistas. La lógica se redirige en el *controller* y se recupera de los *helpers*.

Patrón *Service to Worker.*
El patrón *Service to Worker* es el resultado de combinar el patrón *View Helper* con un *Dispatcher*, en conjunción con el patrón *Front Controller*.

Patrón *Dispatcher View.*
Es el resultado de combinar el patrón *View Helper* con un *Dispatcher* en conjunto con el patrón *Front Controller*. [Deepak].

3.2.3 *View Helper*

Contexto

El sistema crea la presentación del contenido, el cual requiere procesamiento dinámico de los datos del negocio.

Problema

Los cambios en la capa de presentación ocurren frecuentemente y son difíciles de implementar y mantener cuando la lógica de acceso a datos y la presentación se mezclan. Esto hace al sistema menos flexible, menos reusable y, en general, resistente al cambio.
Mezclar la lógica de negocios y el manejo de datos con el procesamiento de las vistas reduce la modularidad y provee una separación muy pobre de los roles entre el equipo de desarrollo.

Fuerzas

La asimilación de los requerimientos de los datos de negocio no es trivial.
Integrar la lógica de negocio en la vista promueve el reuso de código de la forma Copiar-Pegar. Esto ocasiona problemas de mantenimiento porque una pieza de lógica se usa nuevamente en la misma vista o en una diferente simplemente duplicándolo en un nuevo lugar.
Se desea promover una separación limpia de labores con un equipo que desarrolle la aplicación y un equipo que realice las vistas apropiadas para la versión final.
Una vista comúnmente se utiliza para responder a una petición particular del negocio.

Solución

Una vista contiene código de formato, delegando las responsabilidades a sus ayudantes, implementados como JavaBeans o etiquetas dinámicas. Los ayudantes también almacenan el modelo de datos intermedios de las vistas y sirven como adaptadores de los datos del negocio. Hay muchas estrategias para implementar el componente de la vista, se pueden utilizar JSPs, Servlets, JavaBeans y etiquetas dinámicas.

Estructura

La figura 3.12 es el diagrama de clases que representa al patrón *View Helper*.

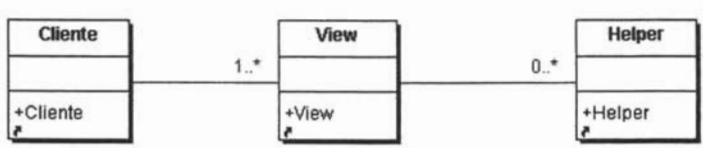


Figura 3.12 Diagrama de clases del patrón *View Helper*.

Participantes y Responsabilidades

La figura 3.13 muestra el diagrama de secuencia que representa el patrón *View Helper*. Típicamente se utiliza un controlador como intermediario entre el cliente y la vista. En algunos casos, un controlador no se utiliza y la vista puede ser el punto de contacto inicial para manejar la petición.

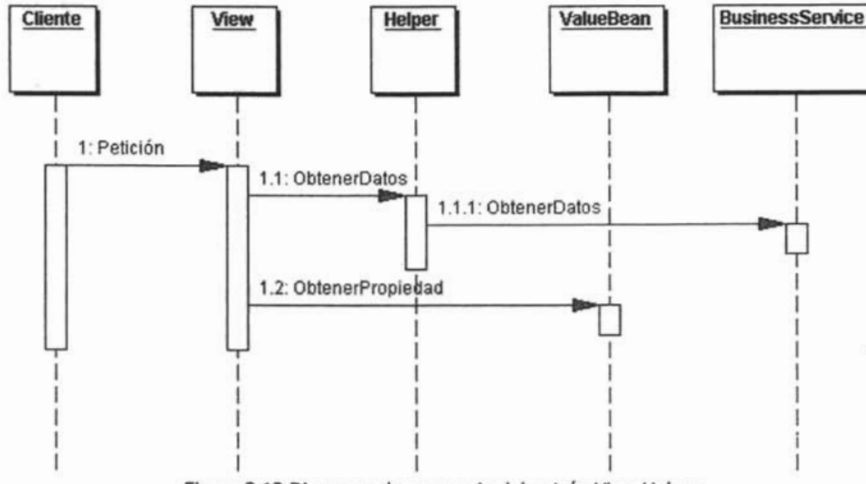


Figura 3.13 Diagrama de secuencia del patrón View Helper.

Puede no existir un *Helper* asociado a una vista. La página puede ser completamente estática o incluir cantidades muy pequeñas de código en línea.

View

Representa y muestra información al cliente. La información que se utiliza en una presentación dinámica se recupera del modelo. Los *Helpers* ayudan a las vistas encapsulando y adaptando el modelo para su utilización en la pantalla.

Helper

Un *Helper* es responsable de ayudar a una Vista o un Controlador a que finalice su tarea. Los *Helpers* tienen muchas responsabilidades como obtener datos o manipular esos datos para su presentación en las vistas.

Una vista puede trabajar con muchos *helpers*, que se implementan típicamente como JavaBeans y etiquetas dinámicas. Además, un *helper* puede representar un objeto *Command*, un *Delegate*, o un transformador de datos (XSLT) de XML que se utiliza con una hoja de estilo para adaptar y convertir el modelo a un formato apropiado.

ValueBean

Es otro nombre para un *helper* que es responsable de almacenar el estado intermedio del modelo que se utilizará en una vista.
Un caso típico se muestra en el diagrama de secuencia 3.13 tiene una regla de negocio que regresa un *ValueBean* como respuesta a una petición, en este caso un *ValueBean* cumplen el rol de un *Value Object*.

BusinessService

Es un rol que se cumple por el servicio que un cliente está buscando acceder. El rol de un *bussines delegate* es proveer control y protección a la regla de negocio.

Patrones Relacionados

Business Delegate

Los *helpers* necesitan acceder a los métodos de las reglas de negocio. Es importante reducir el acoplamiento entre los *helpers* y entre las reglas de negocio en la capa de presentación. Se recomienda usar otro componente porque los componentes de esta capa pueden estar distribuidos ocultando de esta forma los detalles de búsqueda y acceso a las reglas del negocio.

Dispatcher View* y *Service to Worker
Cuando se desea un control centralizado es común utilizar estos patrones por seguridad, administración de contenido y la navegación. [Deepak].

3.2.4 *Composite View*

Contexto

Algunas páginas Web requieren contenido de varias fuentes usando varias subvistas que componen una sola pantalla. Además, una variedad de individuos con diferentes habilidades contribuyen al desarrollo y mantenimiento de estas páginas Web.

Problema

En lugar de proveer un mecanismo de combinación modular de porciones atómicas de una vista en un todo, las páginas se construyen incluyendo código con formato directamente dentro de cada vista.
La modificación de la distribución de múltiples vistas es difícil y propensa a errores debido a la duplicación de código.

Fuerzas

Porciones atómicas de las vistas cambian frecuentemente.
Múltiples vistas compuestas usan subvistas similares, tal como una tabla de inventarios.

Estas porciones atómicas se decoran con diferentes plantillas de texto o aparecen en distinto lugar dentro de la página.
Los cambios en la distribución son más difíciles de administrar y mantener cuando las subvistas son directamente integradas y duplicadas en múltiples vistas.
Integrar frecuentemente porciones de plantillas de texto que se modifican en las vistas también afecta potencialmente la disponibilidad y administración del sistema. El servidor puede necesitar reiniciarse antes que los clientes puedan ver las modificaciones o actualizaciones a estas plantillas.

Solución

Usar vistas compuestas que se conforman de múltiples subvistas atómicas.
Cada uno de los componentes de la plantilla pueden incluirse dinámicamente en la página y la distribución de la página puede utilizarse de manera independiente del contenido.
Se proporciona esta solución para la creación de una vista compuesta basada en la inclusión o sustitución de fragmentos dinámicos o estáticos de una plantilla. Promueve el reuso de porciones atómicas de la vista mediante un diseño modular.

Es apropiado usar una vista compuesta para generar páginas que contienen componentes que pueden combinarse en una variedad de formas.
Otro beneficio es que los diseñadores de la página Web pueden insertar contenido estático en cada región de la plantilla. A medida que el diseño avanza el contenido se puede cambiar por las versiones finales.

Existe cierto procesamiento que debe considerarse al manejar plantillas en un navegador. Al manejar plantillas complejas debe cuidarse la distribución y administración de las plantillas.

Estructura

La Figura 3.14 muestra el diagrama de clases que representa al patrón *Composite View.*

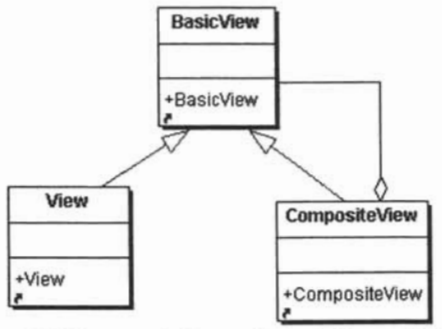


Figura 3.14 Diagrama de Clases del patrón Composite View.

Participantes y responsabilidades.

La Figura 3.15 muestra el diagrama de secuencia para el patrón *Composite View.*

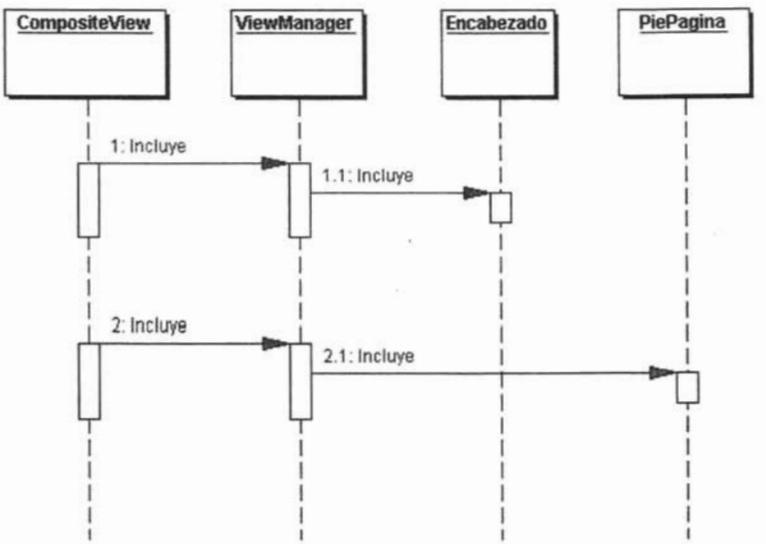


Figura 3.15 Diagrama de secuencia del patrón Composite View.

Composite View

Una vista compuesta es una vista que es un agregado de múltiples subvistas.

View Manager

Administra la inclusión de porciones de fragmento de plantilla en la vista compuesta.
El administrador de la vista puede ser parte de un JSP estándar, en la forma de una etiqueta de inclusión (*<jsp:include>*) o encapsularse dentro de un *JavaBean* o etiqueta dinámica.

Un beneficio de usar un mecanismo en lugar de una etiqueta es que la inclusión condicional se hace más fácilmente.

Included View
Una vista incluida es una subvista que es una pieza atómica de una vista completa más grande. Esta vista incluida puede ser compuesta, incluyendo ella misma múltiples subvistas.

Patrones relacionados.

View Helper
Puede utilizarse como la vista en el patrón *View Helper*.

Composite
Se basa en el patrón *Composite*, que describe jerarquías de parte o total donde un objeto compuesto se compone de muchas piezas, que son tratadas como iguales. [Deepak].

3.3 Diseño de interfaces de usuario.

3.3.1 Introducción.

Los objetivos de la Interacción Persona Computadora son desarrollar o mejorar la seguridad, utilidad, efectividad, eficiencia y usabilidad de sistemas que incluyan computadoras. Cuando decimos sistemas no nos referimos tan solo al hardware y al software sino también a todo el entorno.

Para hacer sistemas interactivos hace falta:

1) Comprender los factores tales como psicológicos, ergonómicos, organizacionales y sociales, que determinan cómo la gente trabaja y hace uso de las computadoras y trasladar esta comprensión para

2) Desarrollar herramientas y técnicas que ayuden a los diseñadores a conseguir que los sistemas informáticos sean los idóneos según las actividades a las cuales se quieran aplicar, para

3) Conseguir una interacción eficiente, efectiva y segura, tanto a nivel individual como de grupo.

Es muy importante comprender que los usuarios no han de cambiar radicalmente su manera de ser, sino que los sistemas han de ser diseñados para satisfacer los requisitos del usuario. [Lorés].

3.3.2 La interfaz de usuario.

La interfaz es una superficie de contacto entre dos entidades. En la interacción persona-computadora estas entidades son la persona y la computadora.
En la vida cotidiana tenemos muchos ejemplos de interfaz. En el caso de una puerta, la chapa de la puerta es la interfaz entre la puerta y la persona. El volante, el acelerador y otros instrumentos y herramientas son la interfaz entre un auto y su conductor. Es muy importante darse cuenta en un primer nivel de que la interfaz refleja las cualidades físicas de las dos partes de la interacción. La chapa está hecha de un material sólido y está bien pegada a la puerta, la cual por otra parte, como tiene que interactuar con la mano, está puesta a la altura de ésta, y tiene la forma que se le adapta. Esta es una idea muy importante en el diseño que puede concretarse en dos conceptos:

1) **visibilidad:** para poder realizar una acción sobre un objeto ha de ser visible, y

2) **comprensión intuitiva**, o propiedad de ser evidente la parte del objeto sobre la que hemos de realizar la acción y cómo hacerlo.

Una interfaz es una superficie de contacto que refleja las propiedades físicas de los que interactúan, y en la que se tienen que intuir las funciones a realizar y nos da un balance de poder y control. [Laurel].

La interfaz de usuario es un lenguaje de entrada para el usuario, un lenguaje de salida para la computadora y un protocolo para la interacción. [Lorés].

Aparte de la interacción física entre usuario y ordenador hemos de añadir un nivel cognitivo referido a que es necesario que el lado humano comprenda el protocolo de interacción y actúe sobre la interfaz e interprete sus reacciones adecuadamente. La interfaz de usuario de un sistema consiste de aquellos aspectos del sistema con los que el usuario entra en contacto, físicamente, perceptivamente o conceptualmente. Los aspectos del sistema que están escondidos para el usuario se denominan la implementación. [Lorés].

La interfaz de usuario es el principal punto de contacto entre el usuario y la computadora; es la parte del sistema que el usuario ve, oye, toca y con la que se comunica. El usuario interactúa con la computadora para poder realizar una tarea.

Dependiendo de la experiencia del usuario con la interfaz, el sistema puede tener éxito o fallar en ayudar al usuario a realizar la tarea. El tipo de problemas que origina una interfaz de usuario pobre incluye la reducción de la productividad, un tiempo de aprendizaje inaceptable y niveles de errores que pueden producir frustración y probablemente lleven a desechar el sistema. [Lorés].

En la interfaz hemos de tener en cuenta también cómo está sentado el usuario, cómo es la organización de la que forma parte el usuario, el ámbito cultural, etc.

3.3.3 Usabilidad.

Para que un sistema interactivo cumpla sus objetivos tiene que ser *usable* y, además, debido a la generalización del uso de las computadoras, *accesible* a las personas que la utilizarán.

La utilidad de un sistema como medio para conseguir un objetivo, tiene un componente de funcionalidad (llamada utilidad funcional) y otra basada en el modo en que los usuarios pueden usar dicha funcionalidad. [Nielsen].

Podemos definir la usabilidad como la medida en la que un producto se puede usar por determinados usuarios para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en un contexto de uso especificado. [Nielsen].

3.3.3.1 Importancia de la usabilidad.

El establecimiento de unos principios de diseño en ingeniería basados en la usabilidad ha tenido como consecuencia:

a) una reducción de los costos de producción.
b) una reducción de los costos de mantenimiento y apoyo.
c) una reducción de los costos de uso.
d) una mejora en la calidad del producto.[Lorés]

Las cosas a veces son difíciles de usar porque en el desarrollo de un producto se da mayor énfasis a la tecnología, en vez del usuario, la persona para la cual esta hecho el dispositivo. [Norman].

La Interfaz de Usuario (IU), es la puerta del usuario a la funcionalidad del sistema subyacente. Que las interfaces de usuario estén mal diseñadas es un factor que frena el uso de las funcionalidades. Por tanto, es muy importante diseñar interfaces de usuario usables.

Podemos entender la usabilidad como aquella característica que *hace que el software sea fácil de utilizar y fácil de aprender*. Un software es fácil de utilizar si realiza la tarea para la que lo estamos usando de una manera fácil, eficiente y intuitiva. La facilidad de aprendizaje se puede medir por lo rápidamente que realizamos una tarea, cuantos errores se cometen y la satisfacción de la gente que lo utiliza. También incluye aspectos como que sea seguro, útil y que tenga un costo adecuado. Una aplicación usable es la que permite que el usuario se concentre en su tarea y no en la aplicación.[Lorés].

3.3.3.2 Principios generales de la usabilidad

Algunos principios generales que se pueden aplicar a un sistema interactivo para mejorar la usabilidad:

1) Facilidad de aprendizaje.

a) Que sea mínimo el tiempo necesario que se requiere desde el no conocimiento de una aplicación a su uso productivo.
b) Proporcionar ayuda a usuarios intermedios. Esta característica permite que los usuarios nuevos comprendan como utilizar inicialmente un sistema interactivo y, a partir de esta utilización, llegar a un nivel de conocimiento y uso del sistema máximo.
c) El usuario tiene que poder evaluar el efecto de operaciones anteriores en el estado actual. Es decir, cuando una operación cambia algún aspecto del estado anterior, es importante que el cambio sea captado por el usuario.
d) Los nuevos usuarios de un sistema poseen una amplia experiencia interactiva con otros sistemas. Esta experiencia se obtiene mediante la interacción en el mundo real y la interacción con otros sistemas informáticos. La familiaridad de un sistema es la correlación que existe entre los conocimientos que posee el usuario y los conocimientos requeridos para la interacción en un sistema nuevo.

2) Consistencia. Un sistema es consistente si todos los mecanismos que se utilizan son siempre usados de la misma manera, siempre que se utilicen y sea cual sea el momento en que se haga.

Recomendaciones para diseñar sistemas consistentes:

a) Seguir guías de estilo siempre que sea posible.
b) Diseñar con un "*look & feel*" común.
c) No hacer modificaciones si no es necesario hacerlas.
d) Añadir nuevas técnicas al conjunto preexistente, en vez de cambiar las ya conocidas.

3) Flexibilidad. La flexibilidad se refiere a la multiplicidad de maneras en que el usuario y el sistema intercambian información.

4) Robustez. La robustez de una interacción cubre las características para poder cumplir sus objetivos y su asesoramiento.

5) Recuperabilidad. Grado de facilidad que una aplicación permite al usuario para corregir una acción una vez reconocido un error.

6) Tiempo de respuesta. Se define generalmente como el tiempo que necesita el sistema para expresar los cambios de estado del usuario. Es importante que los tiempos de respuesta sean soportables para el usuario.

7) Adecuación de las tareas. Grado en que los servicios del sistema soportan todas las tareas que el usuario quiere hacer y la manera en que éstas las comprenden.

8) Disminución de la carga cognitiva. Esto significa que:

a) Los usuarios tienen que confiar más en los reconocimientos que en los recuerdos.
b) Los usuarios no tienen que recordar abreviaciones y códigos muy complicados.[Dix].

3.4 El marco de Trabajo *Struts*

3.4.1 Introducción al marco de trabajo *Struts*

El marco de trabajo *Struts*[12] fue creado para desarrollar más fácilmente aplicaciones Web basadas en tecnologías Java Servlets y JSPs. Se trata de unificar la infraestructura sobre la que se basan las aplicaciones de Internet, concentrando la mayor parte de trabajo en las reglas de negocio.

El marco de trabajo *Struts* fue creado por *Craig* R. *McClanahan* y donado a la fundación Apache en el 2000. El proyecto Jakarta de la fundación Apache trata de proveer productos para servidor con calidad comercial basados en lenguaje Java, de una forma cooperativa.

3.4.2 Paquetes del marco de Trabajo Struts.

El marco de trabajo *Struts* está formado de aproximadamente 300 clases, divididas en 8 paquetes principales (estos paquetes crecen y se modifican continuamente.). Los paquetes se muestran en la Figura 3.16. **[Cavaness].**

[12] Struts es un framework para desarrollo de aplicaciones web utilizando el patrón MVC.

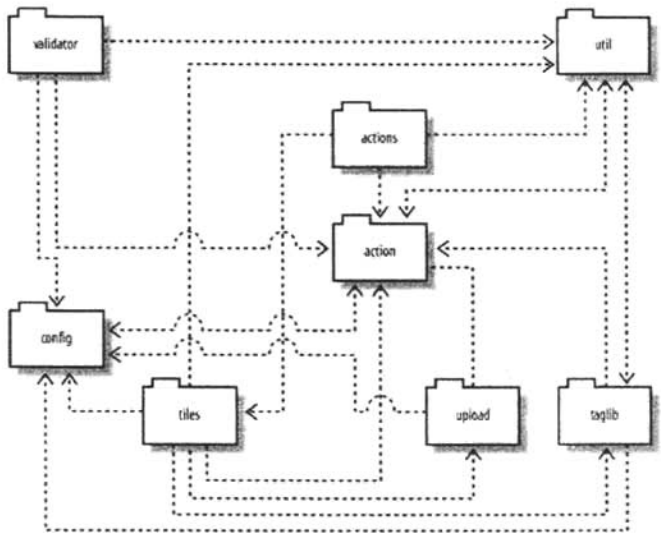


Figura 3.16 Organización de paquetes de la estructura Struts.

Paquete	Descripción
action	Contiene las clases controladoras, el *ActionForm*, *ActionMessages* y otros componentes requeridos.
actions	Contiene algunas clases como el *DispacthAction* que pueden extenderse en la aplicación.
config	Incluye las clases de configuración que son representaciones en memoria del archivo de configuración de *Struts*.
taglib	Contiene las clases manejadoras de etiquetas para las bibliotecas de etiquetas de Struts.
tiles	Contiene las clases utilizadas por el *framework Tiles*.
upload	Contiene las clases usadas para cargar y descargar archivos del sistema local desde un navegador.
util	Contiene clases de propósito general utilizadas por el *framework* completo.
validator	Contienen las clases validadoras para el *framework Struts*.

3.4.3 Componentes de Struts.

Los componentes de *Struts* se muestran en el diagrama de la Figura 3.17.

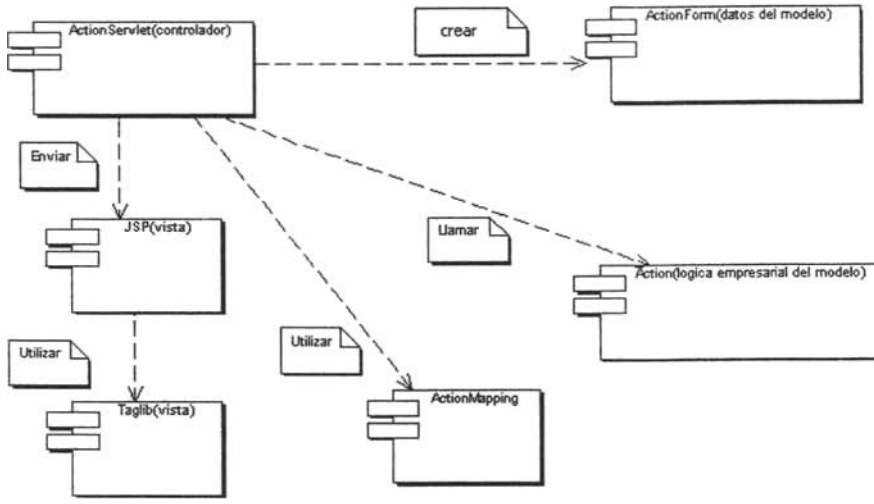


Figura 3.17 Componentes *Struts*.

El nivel de presentación de una aplicación basada en *Struts* se diseña utilizando las bibliotecas de etiquetas de *Struts* (taglib).

Las solicitudes de un cliente se transmiten a un *servlet* denominado *ActionServlet* que actúa como controlador. En una aplicación que utiliza *Struts* todas las solicitudes deben pasar por el *ActionServlet* que, a su vez, envía los datos de la solicitud a un *JavaBean ActionForm*.

Un *ActionForm* es un *JavaBean* que representa los datos desde un componente en vista formulario. Dichos formularios son generados por las páginas JSP con ayuda de la biblioteca de etiquetas html de *Struts*. El *ActionServlet* le añade parámetros *request* al *bean*, que también puede solicitar al *ActionForm* que valide los datos enviados por el usuario.

El *ActionServlet* se configura al definir una serie de objetos *ActionMappings*. Un *ActionMapping* es un objeto que analiza la dirección *URL* mediante una solicitud de componentes ofrecida por el diseñador de la aplicación para procesar la solicitud. La definición de *ActionServlet* y *ActionMappings* se ejecutan en los archivos de configuración XML.

Los componentes específicos de la aplicación se denominan clases *Action* y representan el modelo de la aplicación. Se puede utilizar para validar la información introducida por el usuario. En caso de error la clase *Action* puede crear instancias de los objetos *Error* que se pueden almacenar en el

objeto *HTTP request*. Si la lógica de la clase *Action* se realiza exitosamente se pasa un objeto *ActionForward* que representa la página JSP necesaria para redirigir la respuesta al controlador. [Falkner].

3.4.4 Componentes de Struts de la Vista.

Una vista representa una pantalla del dominio del problema en una interfaz de usuario. Pueden existir muchas vistas diferentes del mismo modelo. El modelo contiene entidades del negocio, que mantiene el estado de la aplicación.

Debido a que los objetos del negocio no tienen una forma natural de representarse a sí mismas externamente, le corresponde a las vistas presentar la información del dominio del problema a los clientes. Esta presentación puede ser en formato de XML y XSLT, mensajes SOAP regresados a un cliente Web o HTML presentado en un navegador. Un dispositivo inalámbrico podría presentar la información en un formato más sencillo usando el mismo modelo. El modelo contiene el estado, mientras la vista se utiliza para representar el modelo o una porción de él al cliente. [Cavaness].

3.4.5 Uso de Vistas en el marco de trabajo *Struts*

Las vistas en el marco *Struts* se construyen utilizando páginas JSP.

Además de los JSPs pueden utilizarse componentes adicionales como:

a) Documentos HTML

b) Bibliotecas propias de etiquetas dinámicas.

c) JavaScript y hojas de estilo.

d) Archivos multimedia.

e) Clases ActionForm

f) Servicio de Localización de Mensajes. [Cavaness].

3.4.5.1 Documentos HTML

Aunque los documentos HTML solo generan contenido estático, no hay nada que impida utilizarlos dentro de aplicaciones con *Struts*. No se podrán enviar datos dinámicos pero pueden utilizarse en contenido estático y no se tienen que usar siempre las capacidades de los JSPs. Por ejemplo una página de entrada (Login) puede tener solo HTML que invoque un *action login*. Sin embargo, se pueden aprovechar las ventajas de los JSPs para manejar contenido dinámico y como una opción para cambios futuros. [Cavaness].

3.4.5.2 Etiquetas dinámicas JSP.

Pueden jugar un rol importante dentro de la arquitectura *Struts*. Aunque en las aplicaciones no es forzoso aplicarlas, algunos escenarios son más difíciles de programar sin ellos.Versiones futuras del *framework* deben hacer más fácil el uso de tecnologías alternas pero las etiquetas dinámicas mantienen su importancia. [Cavaness].

3.4.5.3 *JavaScript* y Hojas de estilo.

Struts no prohibe el uso de JavaScript dentro de una aplicación. Por el contrario, provee funcionalidad dentro de las bibliotecas de etiquetas para facilitar su uso.

Tampoco se prohibe el uso de las hojas de estilo. Se pueden incluir hojas de estilo dentro de un JSP, solo se enviará en un navegador de la misma forma que una página HTML estándar.

Las hojas de estilo se usan para que el diseñador tenga más control sobre la apriencia de la página Web. El tamaño, color, tipo de letra y otras características de *Look and Feel* pueden cambiarse en un punto central y tener un efecto inmediato a través del sitio completo. [Cavaness].

3.4.5.4 Archivos Multimedia

Los archivos multimedia se utilizan en muchas aplicaciones Web. Estos pueden contener:

a) Imágenes (.gif, .jpg, etc.)

b) Audio (.wav, .mp3, etc.)

c) Video (.avi, .mpg, etc.)

Las imágenes son los recursos más utilizados aunque en aplicaciones de negocios los archivos de audio y video son muy importantes. *Struts* permite el uso de archivos multimedia dentro de una aplicación, esto se logra mediante el uso de etiquetas dinámicas o de HTML para enviar este tipo de contenido. [Cavaness].

3.4.5.5 Localización de mensajes.

Proveen un medio de soportar la localización y reducir el tiempo de mantenimiento y rendimiento a través de una aplicación.Por ejemplo si existen muchas etiquetas o mensajes en muchos lugares. En lugar de copiar el texto en cada página puede recuperarse el contenido a través de etiquetas dinámicas, así si el contenido o texto necesita modificarse se hará en un solo lugar.

En el siguiente capítulo se mostrarán los diagramas de clases de la herramienta HIM obtenidos en la etapa del diseño. En esta etapa se aplican los patrones descritos en los dos capítulos anteriores. Se mostrará la aplicación de los distintos patrones elegidos en la arquitectura de la herramienta HIM correspondientes a la capa de presentación: *Front Controller, Validator, Composite View, View Helper.* [Cavaness].

IV. ANÁLISIS Y DISEÑO DE LA HERRAMIENTA INTEGRAL MOPROSOFT.

4.0 Introducción

En capítulos previos se definieron los distintos patrones del marco de trabajo J2EE que se utilizaron en el diseño de la herramienta HIM, se describió en detalle el patrón de arquitectura MVC que se implementa mediante el marco de trabajo *Struts* y se comunicaron algunos patrones para definir el funcionamiento global.

En este capítulo se muestra el análisis y el diseño de la herramienta HIM aplicando el conjunto de patrones seleccionados para la capa de presentación. Se muestran los casos de uso simplificados para tener un punto de partida en el análisis y el diseño.

4.1 Casos de uso de la herramienta HIM

En el desarrollo de la herramienta HIM utilizamos el Proceso Unificado de Desarrollo de Software **[Jacobson]**, por lo que es importante contar con un modelo de casos de uso para guiar todo el proceso de desarrollo.

Con base en requerimientos iniciales que aparecen como descripciones de lo que se pretende que realice la herramienta se generan casos de uso que sirven para guiar el desarrollo.

Entre las características iniciales de la herramienta están:

- Se pretende realizar una herramienta de software que apoye a las actividades de las personas involucradas en el desarrollo de proyectos de software.

- Características deseables de la herramienta incluyen:

    1. Que forme un ambiente integrado para soportar las actividades de *MoProSoft*.
    2. Que use la tecnología que permita su distribución como software libre.
    3. Que tenga las siguientes cualidades:
        a. Portabilidad
        b. Soporte de diversos formatos de documentos
        c. Seguridad en el resguardo y en el acceso
        d. Usabilidad
        e. Soporte de grupos de trabajo
        f. Mantenible por sus propios usuarios.

Cada uno de estos requerimientos aporta información importante para definir la totalidad del sistema.

Aún no existe una herramienta que permita el uso de *MoProSoft* como guía de procesos, por lo que se realizó un análisis del Modelo de Procesos de Software para determinar las características de la herramienta.

Entre las características más importantes están la identificación de todos los roles que se cumplen a lo largo de todo el proceso, la jerarquía de roles del modelo se muestra en la Figura 4.1.

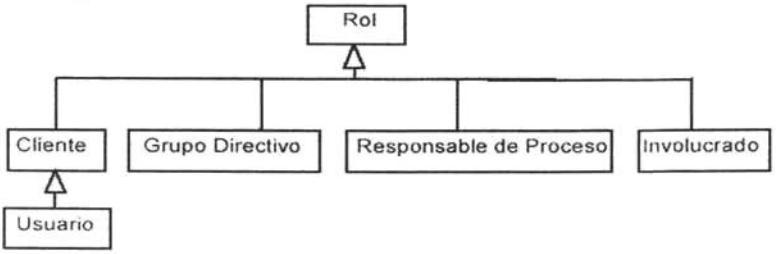


Figura 4.1 Jerarquía de roles de *MoProSoft*

Cada uno de los roles de *MoProSoft* entra en una de estas categorías, cualquiera de estos roles puede interactuar con la herramienta. Los roles tienen un nombre asociado y una responsabilidad en cada uno de los procesos en que interviene.

El modelo de procesos que se utiliza define una serie de actividades para cada uno de los procesos en el modelo. Las actividades están relacionadas unas con otras en el proceso o en otros procesos.

Como resultado de las actividades realizadas se generan una serie de productos que tienen una jerarquía como se muestra en la figura 4.2.

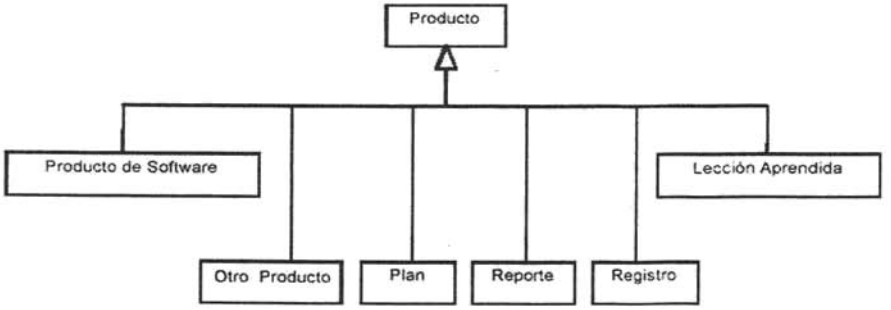


Figura 4.2. Productos generados en el modelo de procesos.

Cada uno de los productos generados podrá guardarse, modificarse y consultarse en alguna actividad del modelo de procesos.

Un usuario cualquiera podrá validarse en el sistema y si tiene asociado algún rol dentro de los procesos o proyectos existentes podrá recuperar los diferentes roles a los que está asociado y podrá realizar todas las tareas que tenga asignadas.

Se propone entonces una herramienta en la que un usuario debe autenticarse para identificarse y adaptarse a los diferentes roles, procesos y proyectos a los que está asociado, de esta forma se recuperan todas las actividades permitidas.
A la salida del sistema los cambios que haga el usuario deben permanecer disponibles a todos los usuarios permitidos y también debe liberar todos los recursos que tenga bajo control.

Se generan dos casos de uso básicos para la herramienta.

Figura 4.3 Caso de uso: Ingresar al Sistema

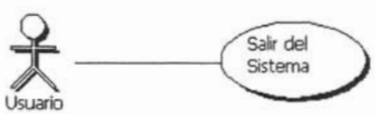
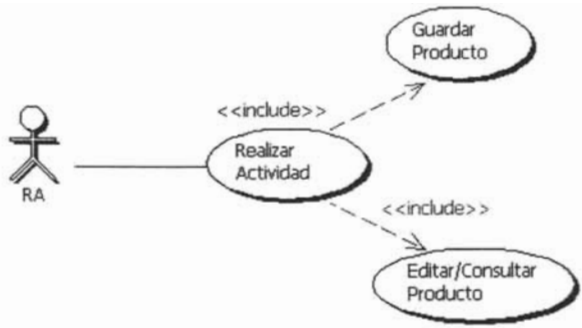


Figura 4.4 Caso de uso: Salir del Sistema

Cada usuario de acuerdo a su rol podrá realizar una actividad y como resultado de esa actividad podrá generar productos o modificar los existentes.

Figura 4.5 Caso de uso: Realizar Actividad



Extensión del caso de uso: Realizar Actividad

La herramienta debe permitir el manejo de archivos y el resguardo de la información generada, este proceso lo agrupamos en otro caso de uso.

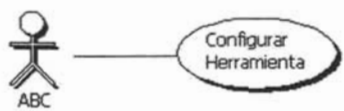


Figura 4.6 Caso de uso: Configurar Herramienta

Se tiene un conjunto de casos de uso básicos para el funcionamiento de la herramienta.

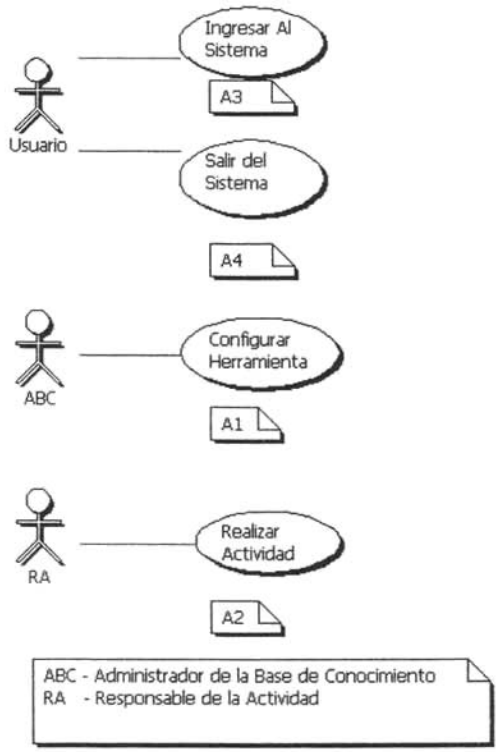


Figura 4.7 Diagrama General de Casos de Uso de la Herramienta HIM.

Para la herramienta se requiere una arquitectura estable que permita la recuperación de datos de una base, la manipulación y presentación de estos datos de distintas formas hasta lograr una versión estable.
Queremos desarrollar una herramienta Web, modificar gradualmente cada una de las pantallas que se presenten al usuario, adaptar reglas de negocio y mejoras con relativa facilidad y manipular todos los datos de manera organizada, esto requiere separar las responsabilidades de cada componente. Se propone el patrón de arquitectura MVC[13] (*Model-View-Controller*), que separa la aplicación en tres componentes: modelo, vista, control.

[13] El patrón de arquitectura MVC se detalla en el capítulo 2, y puede verse el diagrama de componentes en la Figura 2.5.

El modelo se encarga de agrupar y clasificar los datos, el control se encarga de manejar toda la lógica de negocio y la vista se encarga de presentar información al usuario para que pueda trabajar con los datos de acuerdo a las reglas definidas en el negocio.

De acuerdo a esta división lógica en modelo, vista y control se seleccionan un conjunto de patrones que al comunicarse permitan la realización de todas las tareas, una selección de patrones para la aplicación se muestra en la Figura 4.8. Este es un esquema de la arquitectura utilizada en el desarrollo de la herramienta.

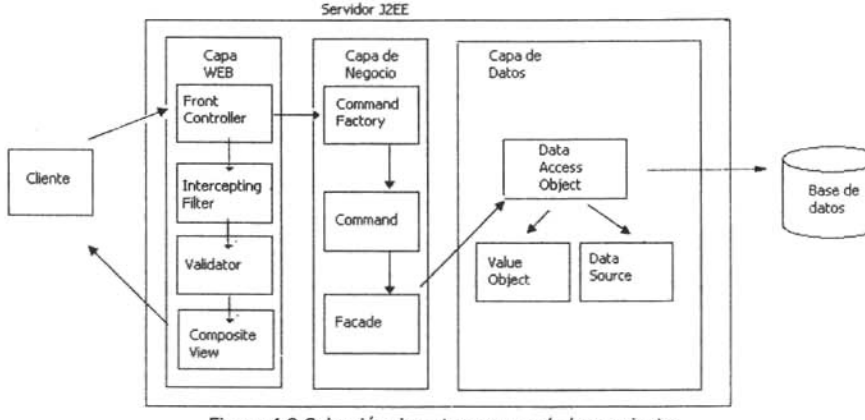


Figura 4.8 Selección de patrones para la herramienta.

La capa WEB representan al componente de las vistas, la capa de negocio representa el control y la capa de datos representa al modelo. **[Vázquez] [Uribe]**.

Con un diagrama de componentes la arquitectura propuesta arriba queda como en la Figura 4.9.

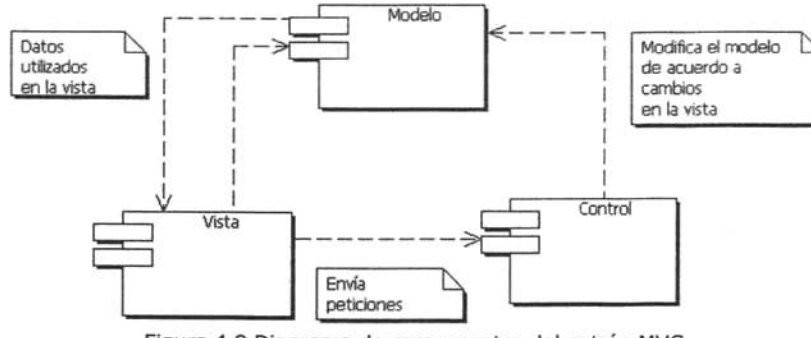


Figura 4.9 Diagrama de componentes del patrón MVC.

Cada uno de los componentes resultantes implementa algunos de los patrones mencionados.

La vista implementa o hace uso de los patrones: *Front Controller, View Helper, Validator y Composite View,* mismos que se aplican y describen en el diseño de los casos de uso.

4.2 Análisis de la capa de presentación de la herramienta HIM.

4.2.1 Caso de uso "Ingresar al sistema".

Para el análisis se observan los aspectos importantes del sistema desde el punto de vista de los desarrolladores, en el caso de uso "Ingresar al sistema". El usuario debe autenticarse y validarse, por lo que requiere interactuar con una parte del sistema que es la interfaz. Las clases del negocio aplicables deben procesar la información que introduce el usuario y de acuerdo a la información de la base de conocimiento se le presenta nueva información al usuario en la misma interfaz.

Un diagrama de análisis de clases para el caso de uso "Ingresar al sistema" se muestra en la Figura 4.10.

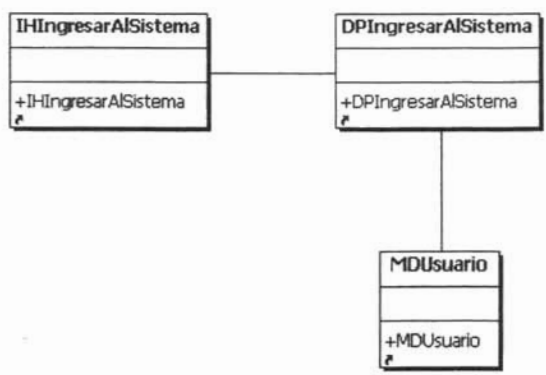


Figura 4.10 Diagrama de clases de análisis: Ingresar al sistema.

El modelo de análisis muestra las asociaciones requeridas. El usuario al ingresar al sistema representado por la clase de interfaz *IHIngresarAlSistema* a través de la cual el usuario debe introducir su nombre y contraseña.
Los datos que introduce el usuario deben verificarse y validarse a través de una clase que representa la lógica del negocio, *DPIngresarAlSistema*, que recupera los datos del usuario en la clase del modelo *MDUsuario*.

4.2.2 Caso de uso "Realizar Actividad"

Las clases de análisis para el caso de uso "Realizar Actividad" parte de la interfaz de usuario que le permitirá cualquier tarea válida al usuario, este es el punto con el que el usuario interactúa con el sistema (*IHMarcoTrabajo*)
Toda la información asociada a un usuario se le llama marco de trabajo, incluye todos los procesos, proyectos, actividades y roles en los que participa. Dependiendo de estas propiedades se le permitirán realizar determinadas actividades (*DPRealizarActividad*) a un usuario que tiene su representación en la base de conocimiento a través de la clase de entidad usuario (*MDUsuario*).

Cada una de las actividades define reglas para el acceso a los productos (*DPProducto*), reglas que permitirán modificar las diferentes propiedades que tiene un determinado producto (*MDProducto*) en la base de conocimiento.

Un diagrama de clases del análisis se muestra en la Figura 4.11.

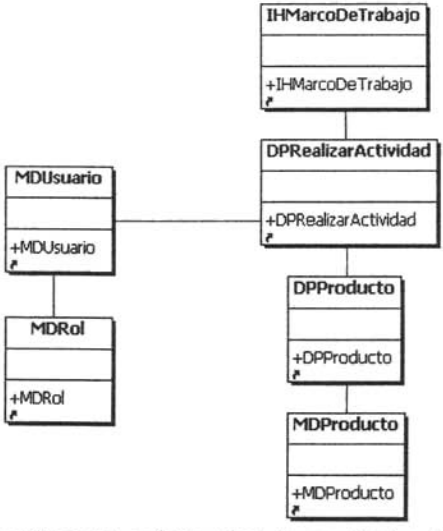


Figura 4.11 Clase del análisis: Caso de uso Realizar Actividad.

4.2.3 Caso de uso "Salir del sistema"

El caso de uso salir del sistema requiere liberar todos los recursos utilizados por un usuario que ha ingresado anteriormente al sistema y tiene un marco de trabajo asignado (*IHMarcoTrabajo*). En general, desde la interfaz de usuario debe seleccionarse esta opción para poder generar la secuencia de acciones en la capa de negocio (*DPSalirDelSistema*) que liberen todos los recursos utilizados por el usuario (MDUsuario). El diagrama de clases se muestra en la Figura 4.12.

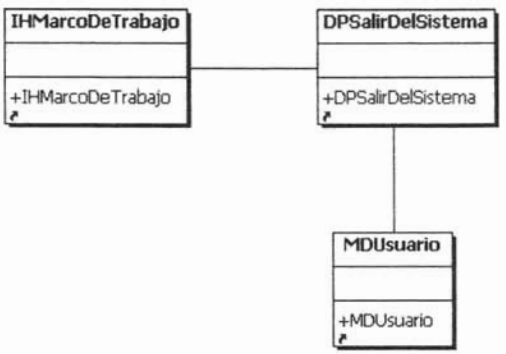


Figura 4.12 Clases del análisis: Salir del sistema

4.3 Diseño de la capa de presentación de la herramienta HIM

En el diseño de la capa de presentación se utilizarán elementos como imágenes, *JSPs* y *Servlets*, archivos de *JavaScript* y etiquetas dinámicas. Se pretende modularizar la interfaz en componentes que se puedan sustituir con facilidad en etapas posteriores.

La interfaz de usuario es uno de los elementos más dinámicos de un sistema, se pretende que con las mismas reglas de negocio y con el mismo modelo se puedan sustituir las pantallas a través de los cuales navega el usuario.
De igual forma se pretende que se adapten nuevas reglas de negocio con facilidad y la creación de nuevas entidades en la base de conocimiento. La organización modular de la interfaz de usuario pretende facilitar la modificación y agregación de pantallas en caso de mantenimiento al sistema.

Se realiza el diseño de la capa de presentación aplicando los patrones seleccionados en los casos en que sea conveniente, ya que en esta etapa se modela la lógica de la página Web.

En particular el patrón *Composite View* es útil para administrar las pantallas, puede formar el marco de trabajo y el patrón *Validator* para verificar los tipos de datos y la información que introduce el usuario. El patrón View Helper se utiliza para presentar cierta información variable del sistema, la lógica de manipulación de datos se maneja en *beans* o en etiquetas dinámicas y hacemos uso de estos en las pantallas (*JSPs*). El patrón *Intercepting Filter* se encarga de verificar los permisos y define las reglas de acceso a los recursos.

La etapa de diseño extiende las clases del análisis detallando cada una de las entidades que intervienen, colocando cada uno de los atributos importantes y los métodos de acceso identificados. Además se identifican nuevas clases y nuevas asociaciones entre las clases identificadas.

Los diagramas de clases muestran el comportamiento estático de las clases para el caso de uso a tratar. Para mostrar el comportamiento dinámico hacemos uso de los diagramas de secuencia. En un caso de uso particular pueden existir diferentes alternativas por lo que pueden manejarse varios diagramas de secuencia para un mismo caso de uso.

En cada uno de los casos de uso se muestran las clases del análisis extendidas y se muestran los diagramas de secuencia correspondientes.

4.3.1 Diseño del caso de uso "Ingresar al sistema".

El caso de uso ingresar al sistema necesita algunos elementos básicos de la interfaz de usuario para poder acceder al sistema, *IHIngresarAlSistema* se mapea desde la clase del análisis. Se presenta al usuario un campo de clave y otro campo de contraseña. Estos campos deben integrarse a un formulario por lo que se crea un *bean* que pueda ayudar al proceso de recuperación de datos del formulario desde la interfaz. El *bean* creado, por la estructura de Struts tiene una nomenclatura que termina en *"form"* aludiendo a los formularios que se muestran en la pantalla, los datos se guardan en la clase *ActionFormUsuario*.

En el diagrama de clases del análisis se ofrece una primera aproximación del caso de uso. Al detallar cada uno de los casos de uso surgen nuevas clases que permiten realizar todas las tareas definidas en ese caso de uso.

El diagrama de diseño ofrece mayor detalle ya que se utiliza como guía en la implementación. En el diagrama de diseño no se muestran todas las clases del modelo ya que se accede a los ellos a través de una fachada[14].

La pantalla de Ingresar al sistema no necesita descomponerse en varias secciones ya que es una pantalla relativamente simple. Se utiliza un punto central de acceso (*ActionServlet*), este encapsula los datos introducidos en un *Helper*

[2] En cada uno de los diagramas se manejan varias clases definidas en el modelo, sin embargo, se acceden a ellos a través de una fachada, por lo que en adelante se menciona solamente a la fachada (BaseConocimiento) como el enlace con el modelo.

(*ActionFormUsuario*), la petición se redirige del punto central a otro clase específica (*ActionUsuario*) que aplica una serie de filtros (*FiltroIP, FiltroPassword*), los filtros verifican y validan la identidad del usuario. Al recuperar el nombre del usuario se aplica una llave a los datos y después se compara con los datos existentes en el repositorio.

Los datos se recuperan a través de una fachada (*BaseConocimiento*) que regresa los datos que se necesitan para verificar la información del usuario. En caso de que los filtros se pasen exitosamente se puede desplegar una pantalla con el marco de trabajo (*IHMarcoDeTrabajo*) o un mensaje de error (*IHError*) en caso de que los datos no sean los adecuados.

En el diagrama de clases se muestra una implementación del patrón *FrontController*[15], ya que cualquier petición por parte del usuario se recibe en un controlador central (ActionServlet), a partir de ese punto central se redirige la petición a otro componente que contiene la lógica necesaria para proporcionar el servicio que se solicita.
Un *ActionForm* opera como un *helper,* esta característica se utilizará para aplicar el patrón *View Helper* y el mismo *Action* opera como un *Dispatcher* redirigiendo a una nueva página que puede ser en este caso el marco de trabajo o un mensaje de error, de esta forma se controla el flujo de las pantallas que se muestran posteriormente, en el componente *Control flujo*.

[15] Todos los casos de uso hacen uso del patrón Front Controller recibiendo las peticiones en un punto central y de ese punto se redirige a otro componente. Es la interacción de la interfaz de usuario y el *ActionServlet.*

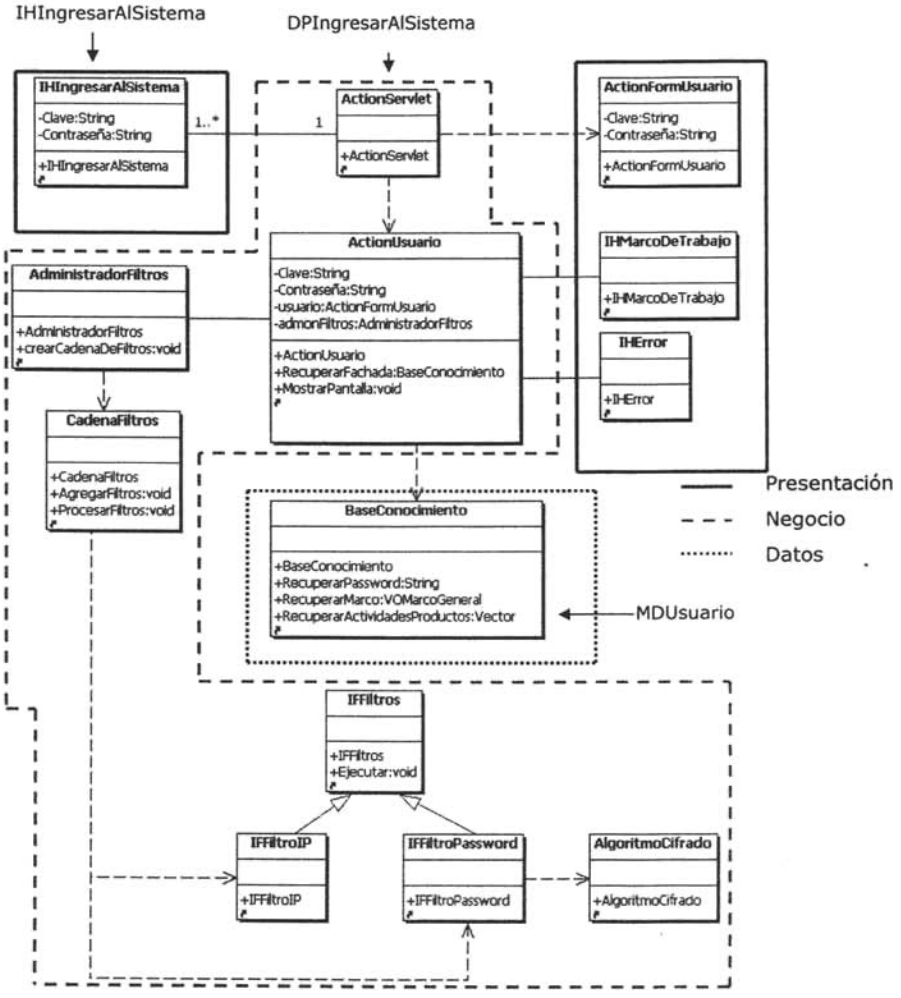


Figura 4.13 Clases del diseño: Caso de uso *IngresarAlSistema*

El diagrama de secuencia del caso de uso Ingresar al sistema se muestra en la Figura 4.14.

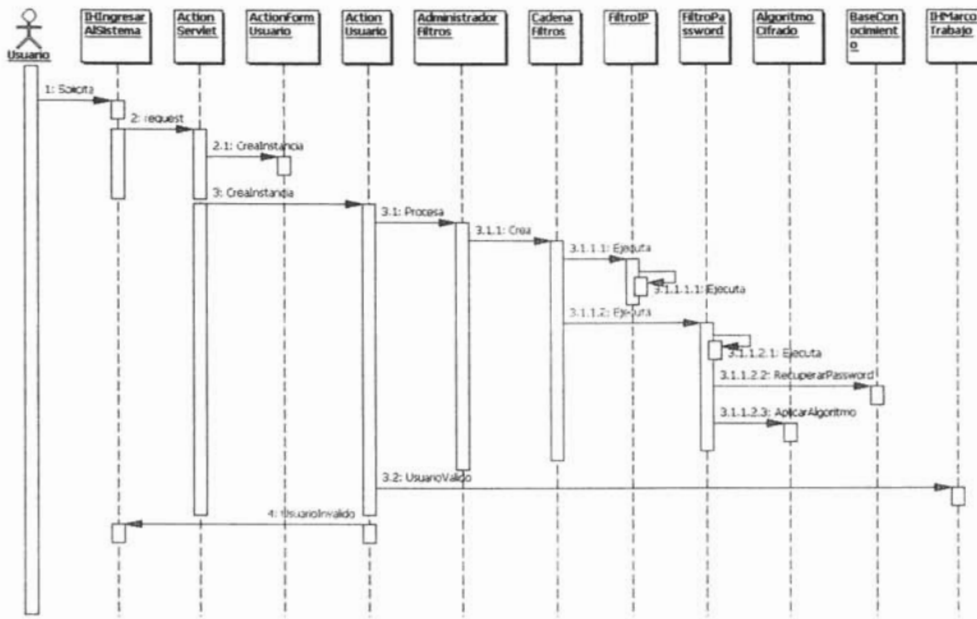


Figura 4.14 Diagrama de secuencia: Caso de uso *IngresarAlSistema*

En este diagrama de secuencia se aprecia el orden cronológico de los eventos al ingresar al sistema, un usuario solicita la página de ingreso al sistema (*IHIngresarAlSistema*), desde el controlador central (*ActionServlet*) se crean las instancias para procesar los datos del usuario (*ActionForm* y *ActionUsuario*). Desde el ActionUsuario creado se recuperan los datos del usuario y se aplica una cadena de Filtros (*CadenaFiltros*) mediante un administrador de filtros que selecciona los filtros convenientes por los que deben pasar los datos. Se validan dos partes principalmente, que la dirección IP que hace la solicitud sea de una fuente válida y que la clave y la contraseña sean las correctas. La contraseña se recupera de la base de conocimiento y se compara con los datos que ha introducido el usuario previa aplicación de un algoritmo de cifrado.
En caso de que los filtros sean pasados sin problemas se presenta al usuario el marco de trabajo (IHMarcoTrabajo) o bien la pantalla de ingreso para que el usuario haga el intento nuevamente.

4.3.2 Diseño del caso de uso "RealizarActividad".

El caso de uso RealizarActividad requiere un poco más de cuidado en el diseño pues es el que nos permitirá llevar a cabo una secuencia de acciones que producirá un producto como salida.
Realizar una actividad requiere autenticarnos, recuperar los procesos, proyectos, roles, actividades definidas en cada proceso o proyecto a los que estamos asociados, es decir su marco de trabajo. Manejar toda esta lógica de recuperación de datos de un usuario corresponde al control y el mantener disponible la información de un usuario corresponde al modelo. La responsabilidad de la interfaz es presentar un estado actualizado del modelo en todo momento.
También al realizar una actividad debe permitir cada uno de los comandos permitidos en esta versión de la herramienta:

a) Crear producto
b) Consultar producto
c) Modificar producto

Al manejar cualquiera de los comandos básicos se requiere actualizar la pantalla para mostrar la información correspondiente. Cada acción del usuario requiere retroalimentación adecuada por lo que se pretende descomponer la interfaz de usuario en cuatro secciones básicas:

1) Encabezado
2) Pie de página
3) Barra lateral
4) Contenido

El encabezado contiene información relevante a la página completa, el proceso o proyecto y los roles asociados a un usuario, la barra lateral contiene información sobre las diferentes actividades que puede realizar un usuario, el pie de página puede servir para comentarios sobre un producto particular o cualquier información pertinente para el usuario, el contenido está pensado para enfocar la información relevante al usuario en un momento determinado y que requiere mucha o toda la atención del usuario.

El esquema para las páginas *JSPs* para realizar una actividad quedaría como se muestra en la Figura 4.15.

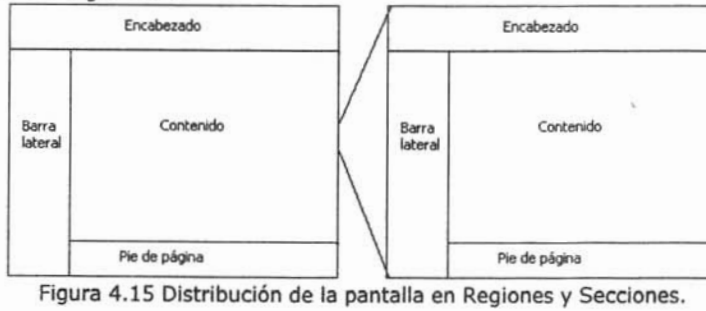


Figura 4.15 Distribución de la pantalla en Regiones y Secciones.

Cada una de las secciones se puede definir como una región, y una región puede definir a su vez todas o cualquiera de las secciones básicas, esto significa que es posible aplicar el patrón de composición de vistas (*Composite* View), para distribuir la información de las diferentes pantallas por las que pasa el usuario.

Un diagrama de clases para realizar cualquier actividad requiere de un marco de trabajo que se forma de la siguiente manera:

Después de autenticar al usuario éste tiene una sesión abierta (*HttpSession*), y se tiene acceso a su marco de trabajo a través de la fachada (BaseConocimiento), para recuperar todas sus actividades, proyectos, procesos y roles. Todos estos datos estarán presentes en la interfaz de usuario (*IHMarcoTrabajo*) que permitirá el manejo de los productos a través de los comandos básicos como peticiones al controlador central (*ActionServlet)* que redirige la petición a un *servlet* específico y se determina cuál es la pantalla siguiente a mostrar (*ControlFlujo*). El servlet específico analiza todos los datos del usuario a través de la fachada (BaseConocimiento) y lanza la nueva pantalla utilizando una plantilla (*JSPPlantilla*) que contiene una distribución específica(Contenido) definiendo las regiones (*Region*) y las secciones (*Seccion*) necesarias. En la plantilla se imprime el contenido de las páginas JSP específicas para la nueva información que requiere el usuario, en cada caso el encabezado, pie de página, contenido ó barra lateral adecuados.

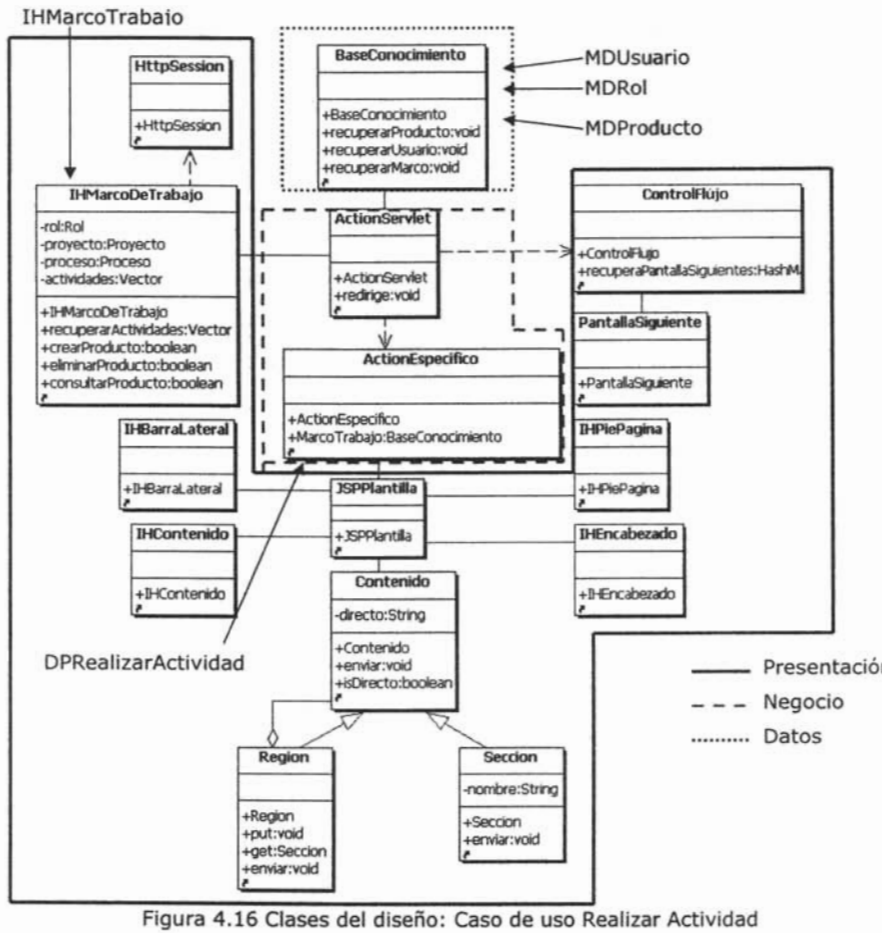


Figura 4.16 Clases del diseño: Caso de uso Realizar Actividad

El comportamiento dinámico para el caso de uso realizar actividad se muestra en la Figura 4.17.

Desde el marco de trabajo (IHMarcoDeTrabajo) se reciben todas las peticiones (consultar, crear o modificar), el *actionServlet* determina cual es la siguiente pantalla a presentar a través de la lectura del archivo de control de flujo (*ControlFlujo*), después de seleccionar la pantalla redirige la petición a un *action* en particular (*ActionEspecífico*). Este *action* particular busca la pantalla siguiente (*PantallaSiguiente*) que se debe mostrar y la almacena en la sesión (*HttpSession*) que el usuario tiene

actualmente. Después de ejecutar toda la lógica y procesamiento de los datos que obtiene del marco de trabajo se envía a una plantilla, ésta plantilla ejecuta el código insertado a través de etiquetas dinámicas (Region/Seccion), las etiquetas dinámicas imprimen el contenido de diferentes JSPs (*IHEncabezado*, *IHContenido*, *IHBarraLateral*, *IHPiePagina*) que forman completamente la pantalla que se debe mostrar al usuario (*IHCrearProducto*, *IHModificarProducto*, *IHConsultarProducto*).

El proceso y la secuencia de acciones descrita se aplican a la manipulación de cualquier producto.

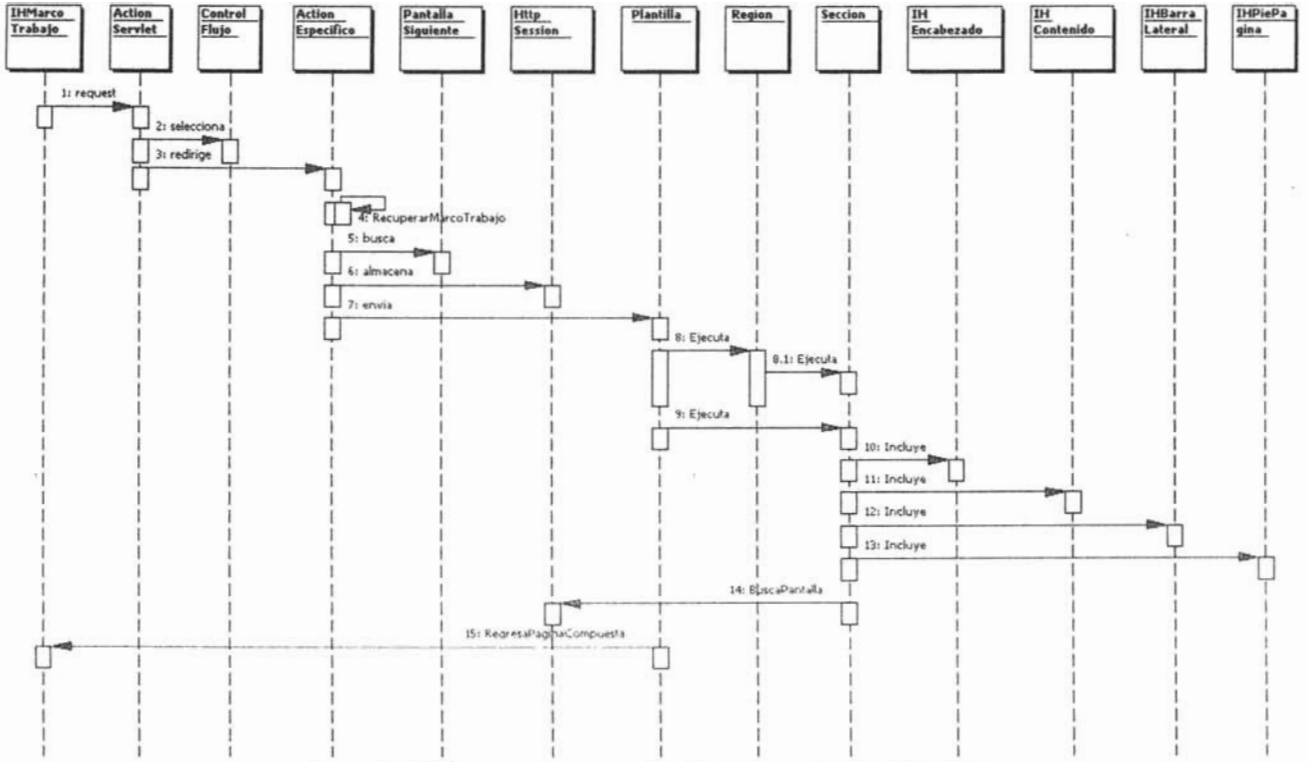


Figura 4.17 Diagrama de secuencia: Caso de uso RealizarActividad.

4.3.3 Caso de uso especial "CrearProducto".

En el caso de uso "*RealizarActividad*" existe la posibilidad de manejar algún producto que se genera como consecuencia de la misma actividad. En general, existen tres posibilidades: en esta versión de la herramienta: Crear un nuevo producto, Consultarlo, Modificarlo.
La lógica para el manejo de productos es muy semejante, desde una pantalla de creación de productos(IHCrearProducto) que está integrada en el marco de trabajo (*IHMarcoTrabajo*) se envía una petición al controlador central que determina la siguiente pantalla a presentar, los datos referentes al producto se almacenan en un *bean ActionFormProducto,* desde el *ActionCrearProducto* correspondiente se hace uso de la *FábricaComandos* para invocar el comando CrearProducto relativo al tipo de producto de que se trate, ya sea un producto de caja negra en forma de archivo o un producto de información especializada el cual el usuario debe ver su contenido en la pantalla. Una vez que se ha guardado la información del producto debe enviarse una confirmación en la pantalla tomando la plantilla y actualizando el contenido únicamente. El diagrama de clases del diseño para crear un producto se muestra en la Figura 4.18.

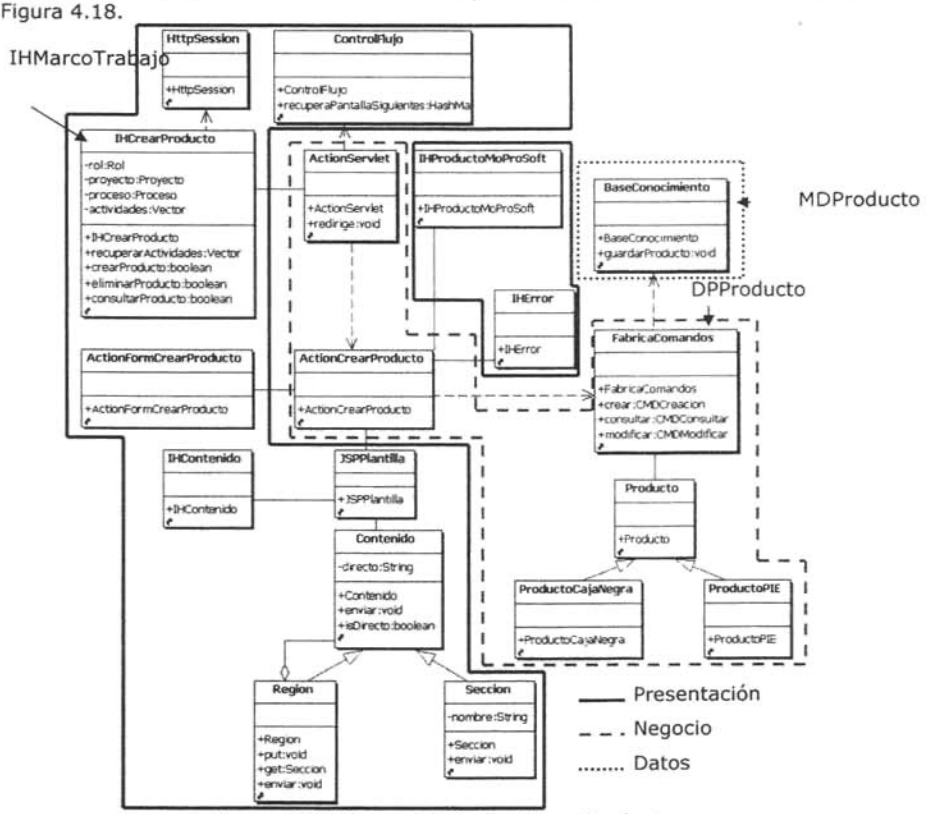


Figura 4.18.Clases del diseño: Crear Producto

El diagrama de secuencia para crear un producto se muestra en la Figura 4.19. El responsable de la actividad hace una solicitud al *ActionServlet* a través de la interfaz de usuario (IHCrearProducto), el *ActionServlet* selecciona la siguiente pantalla a mostrar y redirige a un servlet que contiene toda la lógica para crear el producto (*ActionCrearProducto*), en el *servlet* se recupera el marco de trabajo del usuario, es decir, los proyectos, procesos, roles a los que está asociado, también se busca la pantalla siguiente a mostrar y se guarda en la sesión existente (*HttpSession*), ahí mismo se crea un *bean* ActionFormProducto para almacenar todos los datos referentes al producto. Como se pretende guardar un producto se crea una instancia de la fábrica de comandos (*FabricaComandos*), que crea un comando para guardar el producto (Producto), este comando llama a un método guardar producto visible desde la fachada (*BaseConocimiento*). El resultado de guardar el producto que puede ser cierto o falso se envía a una plantilla (*JSPPlantilla*). En la plantilla se interpretan las etiquetas dinámicas en la pantalla que se encuentra en la sesión. Si el resultado de guardar el producto es satisfactorio se muestra una nueva página con el contenido actualizado (*IHProductoCreado*) o si existe algún problema se presenta el problema al usuario (*IHError*).

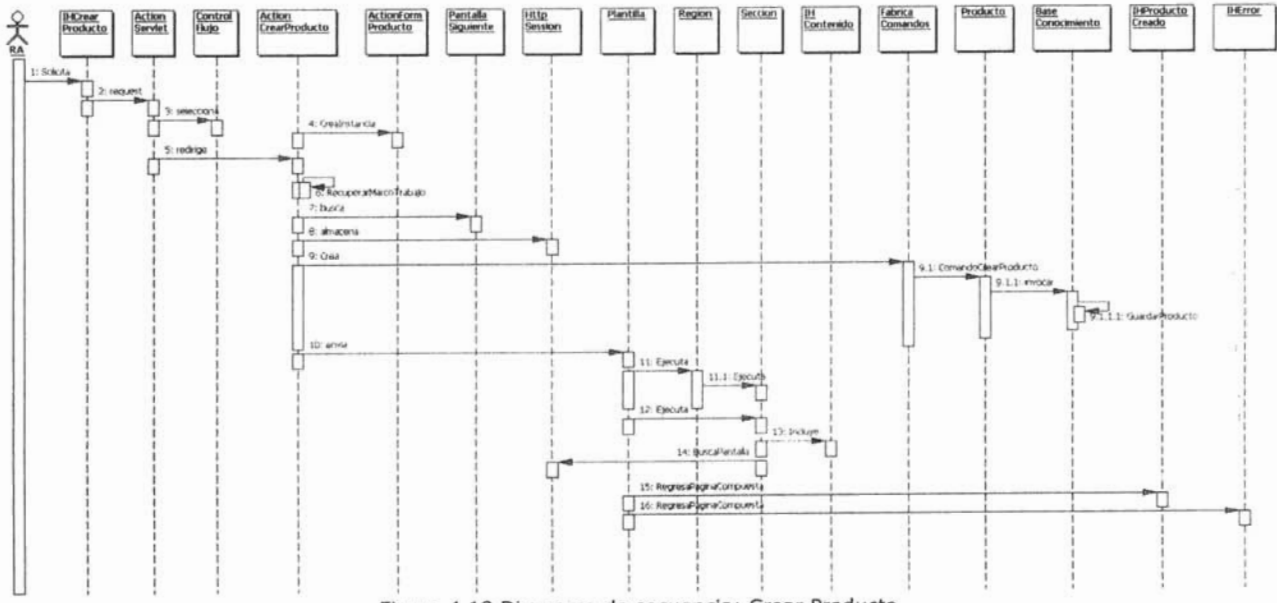


Figura 4.18.Diagrama de secuencia: Crear Producto

4.3.4 Caso de uso especial "ConsultarProducto".

Un paso importante al realizar actividades es la consulta de algún producto. Todos los procesos tienen productos de entrada y salida, cada una de las actividades puede generar un documento de salida y para realizar esos documentos es necesario consultar las fuentes correspondientes. Un documento creado debe ser consultado para que tenga algún valor. Estos productos pueden consultarse mediante un proceso similar al de guardar un producto.
Desde una pantalla del sistema el usuario (*IHConsultarProducto*) hace una petición relacionando el controlador central (*ActionServlet*) con las clases correspondientes del *framework* Struts (*ActionFormConsultarProducto, ActionConsultarProducto*). Se invoca al comando consultar mismo que regresa información que se despliega en pantalla (*IHProductoMoProSoft*) que utiliza una plantilla como base y etiquetas dinámicas para actualizar el contenido.
El diagrama de clases para consultar un producto se muestra en la Figura 4.20

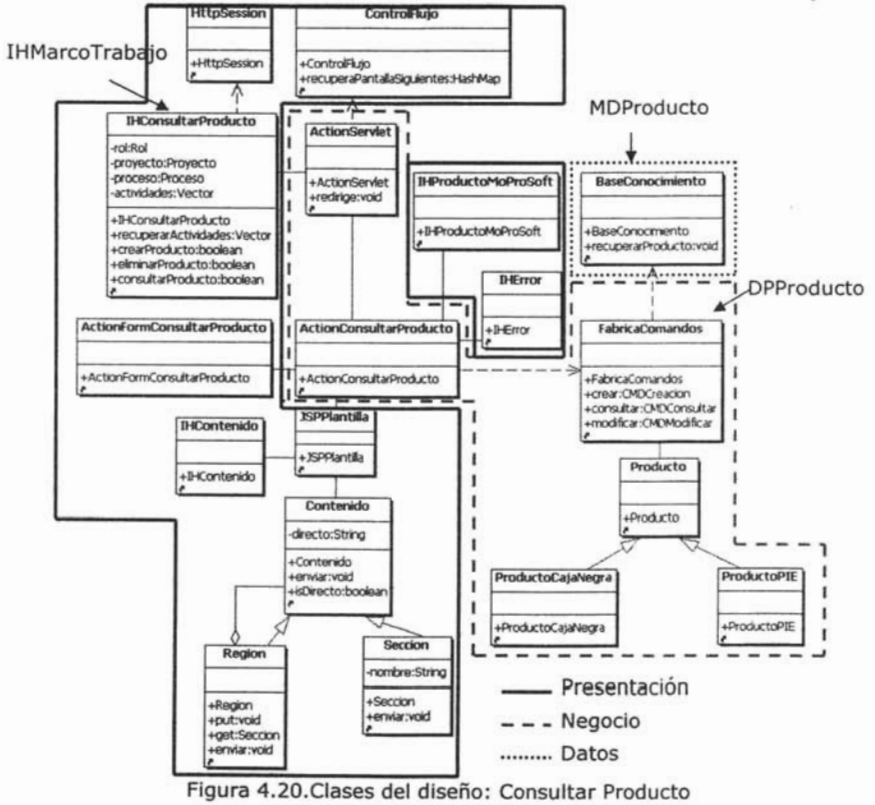


Figura 4.20.Clases del diseño: Consultar Producto

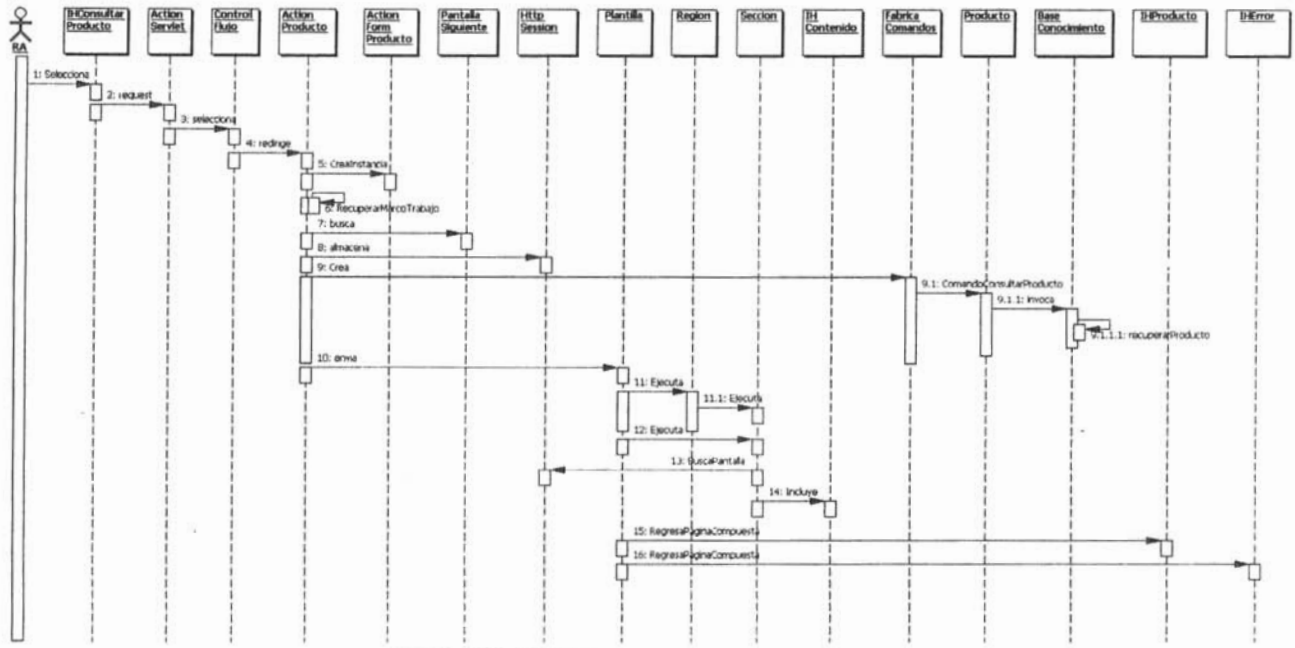


Figura 4.21.Clases del diseño: Consultar Producto

En la Figura 4.21 se muestra que el proceso de consulta de un producto sigue una secuencia similar que la de crear un producto. Se invoca la consulta de un producto y desde el *ActionProducto* se presenta la información recuperada en la base de conocimiento actualizando el contenido y mostrando al usuario el resultado.

4.3.5 Caso de uso especial "ModificarProducto".

La Figura 4.22 muestra el diagrama de clases del diseño del caso de uso ModificarProducto, se realiza una petición desde la interfaz de usuario recuperando todos los datos actuales de la base de conocimiento, estos datos se presentan al usuario para que los pueda modificar. La parte que se actualiza de la pantalla es el contenido. Este presenta los campos necesarios para que se modifique la información del producto. Una vez que se han modificado los datos se hace la petición para guardar la información nueva en la base de conocimiento.

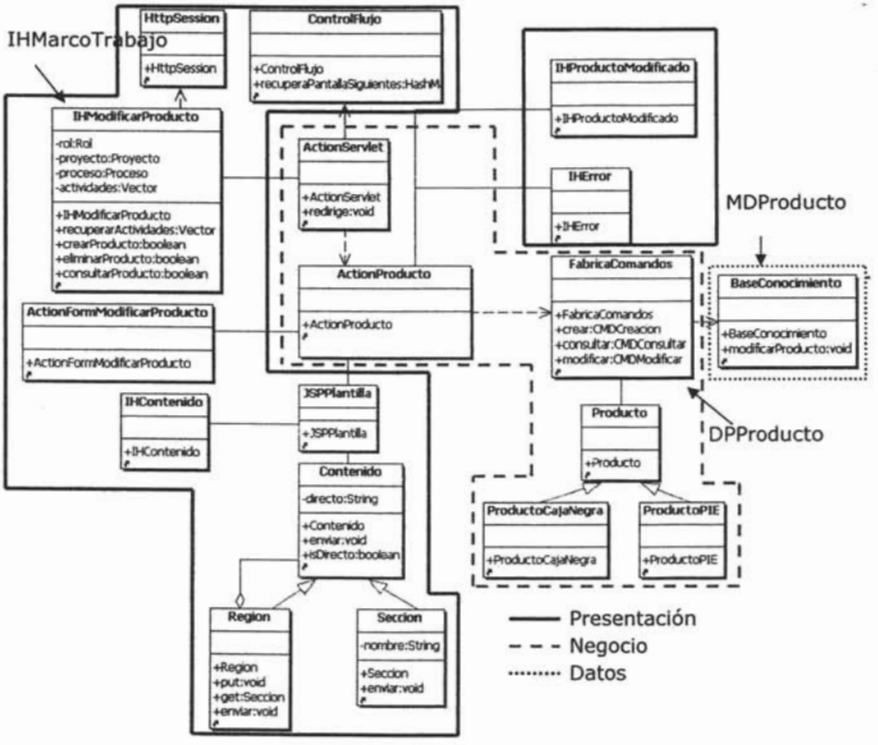


Figura 4.22 Clases del diseño: Modificar Producto

El diagrama de secuencia se muestra en la Figura 4.23. Desde la interfaz de usuario se hace la petición de la modificación del producto, la petición se recibe y redirige desde el *ActionServlet*, el *ActionProducto* crea la instancia de un *bean ActionFormProducto*, donde se guardan todos los datos que se recuperan de la base de conocimiento. Estos

datos recuperados se presentan al usuario al interpretar las etiquetas dinámicas en la plantilla que se encarga de incluir el contenido actualizado. Con el contenido presentado al usuario puede modificar cualquier parámetro del documento y hacer la petición de guardar el producto, esta petición se recibe en el *ActionServlet*, que redirige la petición al *ActionProducto*, desde ahí se invoca a la fábrica de comandos, se crea el comando que invoca el método *actualizarProducto* de la base de conocimiento. El resultado se presenta al usuario actualizando el contenido de la plantilla, se forma la página (*IHProductoModificado*) o se presenta el error en la pantalla (*IHError*).

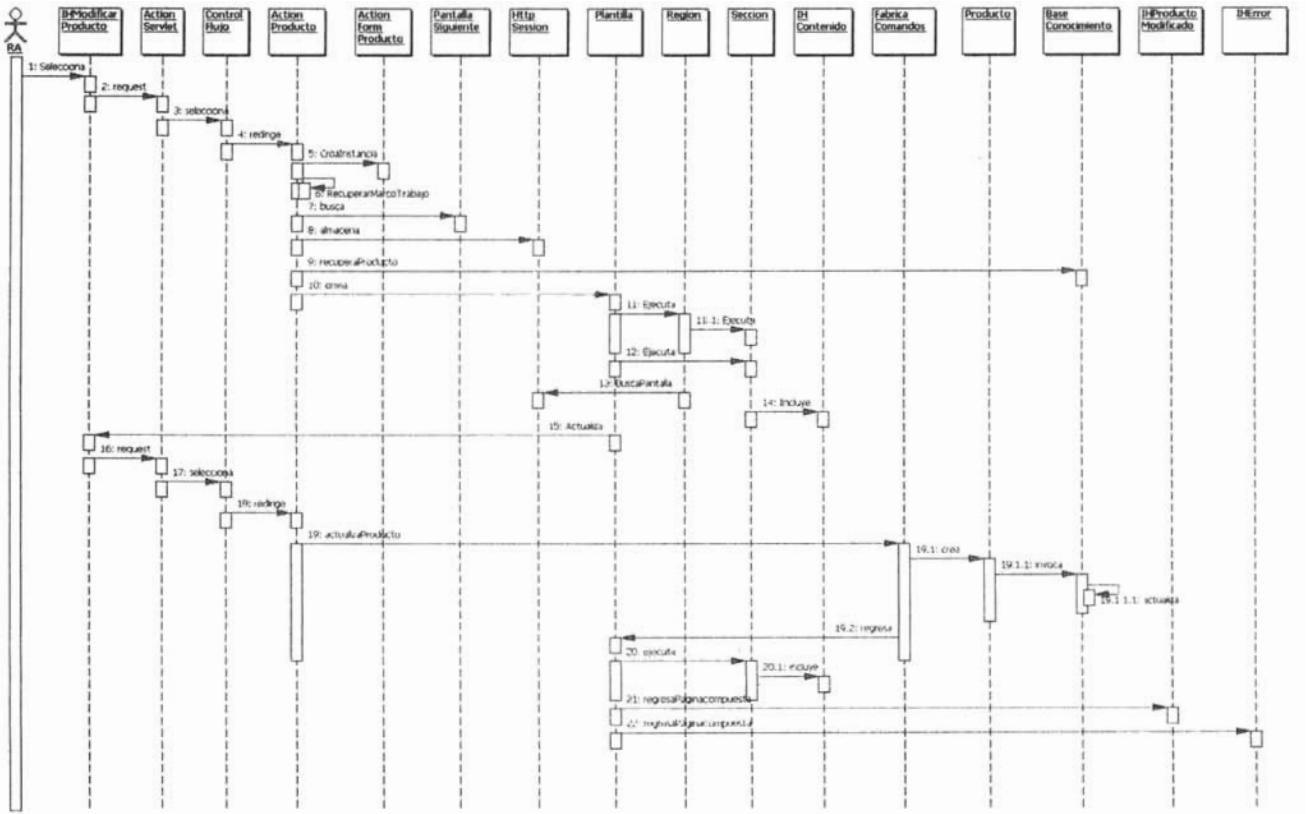


Figura 4.23 Diagrama de secuencia: Modificar Producto.

4.3.6 Caso de uso "Salir del sistema".

El caso de uso salir del sistema pretende liberar todos los recursos que el usuario ha utilizado y que aún tiene en uso. Esta petición le corresponde directamente al usuario validado a través de la interfaz (*IHMarcoTrabajo*). Un usuario validado tiene abierta una sesión de trabajo en la que puede manejar toda la información asociada a su marco de trabajo. Un controlador específico indica a la base de conocimiento que debe realizar una rutina de finalización.
Una nueva pantalla se utiliza para indicar al usuario que su sesión se ha cerrado con éxito. El diagrama de clases se muestra en la Figura 4.24.

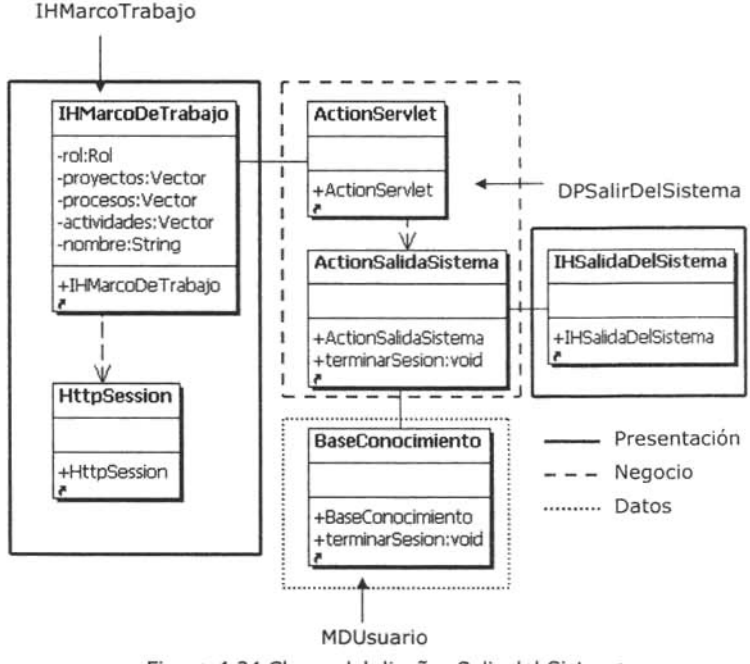


Figura 4.24 Clases del diseño: Salir del Sistema.

La figura 4.25 muestra el diagrama de secuencia para el caso de uso salir del sistema. Desde el marco de trabajo (*IHMarcoDeTrabajo*) un usuario validado debe seleccionar la opción de salir del sistema. La petición que se hace desde la interfaz del usuario se dirige a un controlador central (*ActionServlet*), este controlador redirige la petición a un controlador específico (*ActionSalidaSistema*) que recupera todo el marco de trabajo que el usuario tiene en la sesión (*httpSession*). Todos los recursos del usuario que tienen que ver con la base de conocimiento se liberan mediante el método *terminarSesion* de la fachada (*BaseConocimiento*), los recursos que aún no se han

liberado se liberan en el *action* específico. Cuando se han liberado todos los recursos se presenta la información en la interfaz de usuario (*IHSalidaDelSistema*).

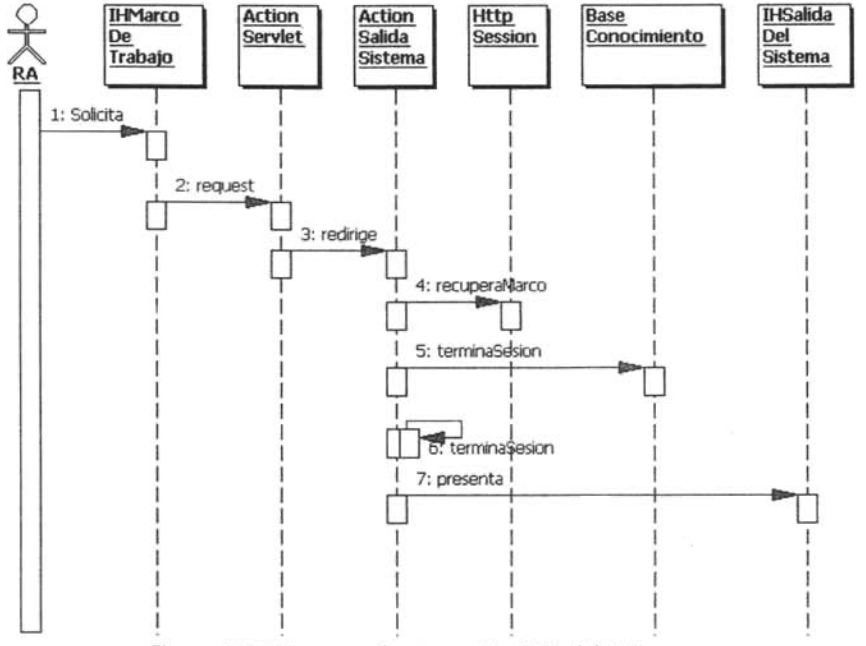


Figura 4.25 Diagrama de secuencia: Salir del Sistema.

En este capítulo se mostró el análisis y diseño utilizado en el desarrollo de la herramienta HIM, se mostró la implementación de los patrones *FrontController* y *Composite View*, e *Intercepting Filter, View Helper* a través de los diagramas de clases y análisis.
En el siguiente capítulo se muestra la implementación de los casos de uso a través de componentes y paquetes en los que se organizan las clases mostradas.

V. IMPLEMENTACION DE LA CAPA DE PRESENTACIÓN DE LA HERRAMIENTA HIM

5.0 Introducción

Este último capítulo muestra el modelo de implementación utilizado en la herramienta HIM, se muestran los diferentes casos de uso del diseño desde una perspectiva física orientada a la construcción de cada una de las clases generadas en el diseño.
En esta etapa de implementación se toman las clases generadas en el diseño y se implementa el sistema en términos de componentes[16], archivos de código fuente, *scripts,* archivos ejecutables y demás tipos de archivos necesarios para realizar las tareas señaladas en los casos de uso.

El objetivo final de la implementación es utilizar todo el trabajo previo desde los requisitos, pasando por el análisis y el diseño, para generar los componentes físicos del sistema. Para que este sistema se pueda utilizar es importante señalar que cada uno de los componentes generados en la implementación deben asignarse a los nodos[17] correspondientes, estos nodos se muestran en un diagrama de distribución que representa la distribución física del sistema para que pueda funcionar correctamente.

5.1 Diagrama de componentes

En la implementación es muy importante contar con los diagramas de componentes porque sirven como guía para planear la construcción de los componentes físicos del sistema. A menudo los diagramas de componentes sirven para ejecutar un plan de implementación, se toman secciones o bloques de los diagramas para que se generen versiones ejecutables y a medida que se avanza en el tiempo estas construcciones iniciales se incrementan para añadir nuevos componentes haciendo una construcción iterativa y a la vez incremental. Este plan de construcción parcial de los componentes tiene como objetivo hacer que los componentes generados se puedan probar e integrar al sistema, de esta forma se pueden detectar además algunos problemas que se han generado y que no se detectaron en etapas anteriores.
En la herramienta HIM se manejan todos los productos generados en la gestión de los proyectos y los procesos. Como resultado de estas gestiones se generan productos que hay que resguardar en la base de conocimiento. Estos productos básicamente son de dos tipos, productos de información especializada (PIE) y productos de caja negra, cada uno de estos productos tiene una secuencia lógica dada por el modelo de procesos MoProSoft.

---

16 Un componente es una parte física y reemplazable de un sistema que se ajusta a, y proporciona la realización de, un conjunto de interfaces.
17 Es un elemento físico que existe en tiempo de ejecución y que representa un recurso computacional, y que en general tiene memoria y capacidad de procesamiento.

La lógica para generar un documento es diferente en cada caso y la información que se le muestra al usuario depende de varias fuentes, es responsabilidad de la interfaz del usuario presentar la información relevante al usuario obteniéndola de la fuente de datos, esto es la base de conocimiento, pero sin pasar por alto las reglas definidas en la capa de control.

En la Figura 5.1 se muestra una lista de documentos que se pueden manejar desde la interfaz de usuario y cuyas reglas de manejo se implementan en la capa de control. Estos productos ya se pueden manejar en esta primera versión y aunque existen más documentos por implementar, éstos se manejan como cajas negras por lo que la lógica ya está dada.

Proceso de MoProSoft	**Tipo de Producto**	**Producto implementado**
Gestión de Negocio	PIE	Lecciones aprendidas
	Caja negra	Plan estratégico.
Gestión de Procesos	PIE	Equipo de Trabajo de Procesos. Lecciones aprendidas.
Recursos Humanos y Ambiente de Trabajo	PIE	Lecciones aprendidas. Registro de recursos humanos. Asignación de recursos humanos al proyecto.
Bienes Servicios e Infraestructura	PIE	Lecciones aprendidas. Solicitud de bienes ó servicios. Registro de proveedores. Registro de bienes ó servicios.
Gestión de Proyectos	PIE	Lecciones aprendidas. Descripción del proyecto. Registro del proyecto.
Administración de Proyectos Específicos	PIE	Equipo de Trabajo. Lecciones aprendidas.
Desarrollo y Mantenimiento de Software	PIE	Lecciones aprendidas.

En las secciones siguientes se muestra el diagrama de componentes para cada caso, estos diagramas de componentes son una continuación a los diagramas de diseño obtenidos en el capítulo anterior.

5.2 Diagrama de componentes del caso de uso "Ingresar al sistema"

El diagrama de componentes para el caso de uso Ingresar al sistema se obtiene a partir del modelo del diseño. Cada una de las clases mostradas en el modelo de diseño debe tener una representación física, un archivo o un componente Java. En el caso de las clases Java se representan como un componente y las clases que corresponden a páginas que se le presentan a la interfaz de usuario se representan como archivos *jsp*. El diagrama de componentes para el caso de uso "*IngresarAlSistema*" se muestra en la figura 5.1.

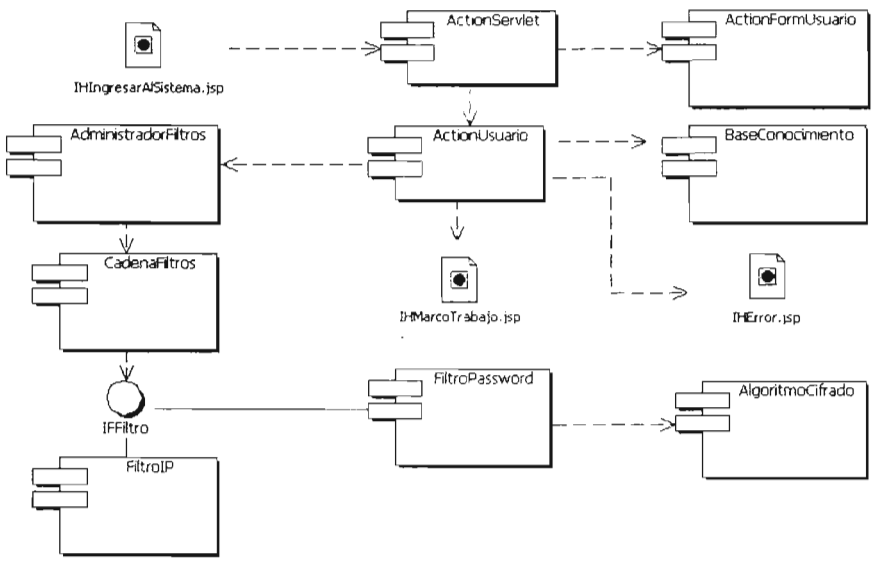


Figura 5.1 Diagrama de componentes del caso de uso "*IngresarAlSistema*".

En este diagrama de componentes se mapea la interacción del modelo del diseño, con la diferencia que se le da más importancia al tipo de archivo que se representa en el modelo.

5.2 Diagrama de componentes del caso de uso "RealizarActividad"

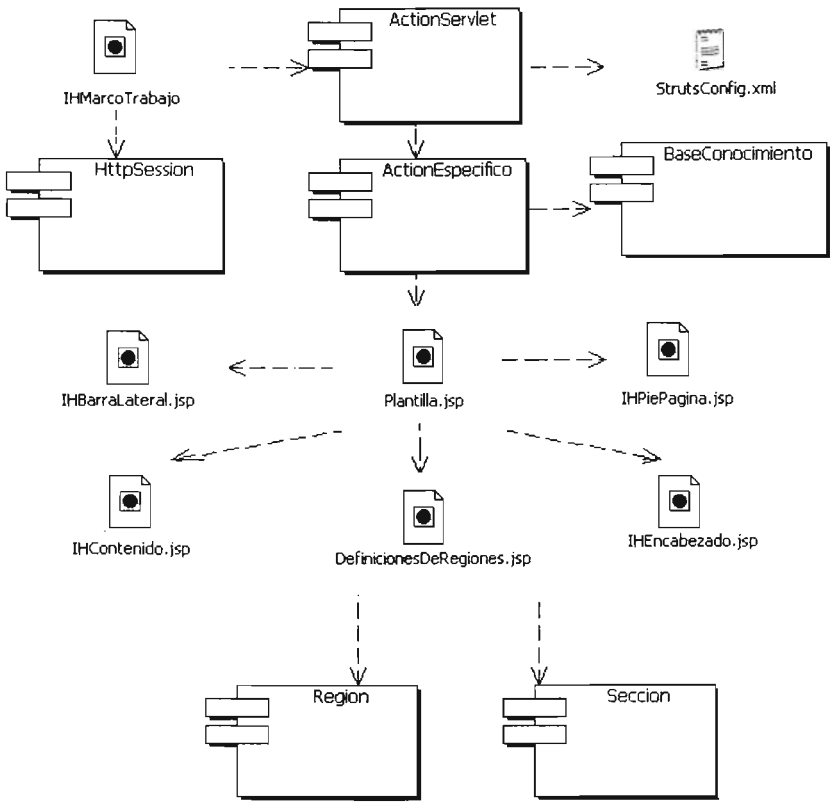


Figura 5.2 Diagrama de componentes del caso de uso "RealizarActividad".

El diagrama de la Figura 5.2 muestra la implementación del caso de uso "RealizarActividad". Este diagrama representa una generalización de todas las actividades que se realizan en el modelo de procesos. El marco de trabajo es un JSP (IHMarcoTrabajo) y el control de flujo se lleva a cabo a través de un archivo de configuración xml que define las pantallas que siguen a continuación y que se almacenan en la sesión del usuario.
La plantilla es una página dinámica y las definiciones de regiones se dan de igual forma en un JSP. La definición de región utiliza las etiquetas dinámicas <region: ... />, para distribuir el contenido en la pantalla de acuerdo a una plantilla.

<region:define id='Region_BarraLateral' scope='application' template ='/Plantillas/plantilla.jsp'>

Todas las partes que componen una nueva página se forman a través de páginas JSPs: el encabezado, contenido, el pie de página y la barra lateral.

Todos los demás componentes se mantienen como archivos con extensión java.

5.2.1 Diagrama de componentes del caso de uso "CrearProducto"

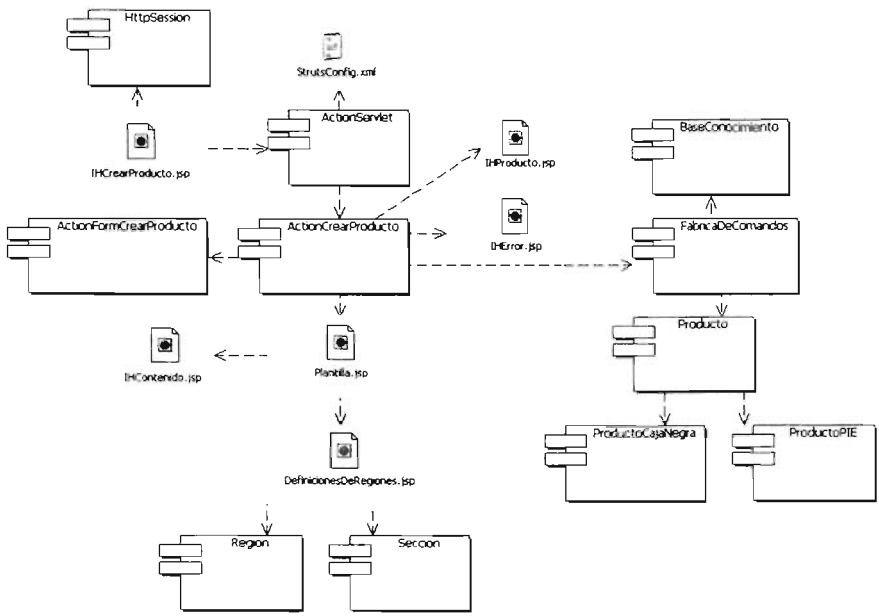


Figura 5.3 Diagrama de componentes del caso de uso "CrearProducto".

Para crear cualquier producto es necesario realizar la petición desde un JSP (*IHCrearProducto*), todos los datos que se ingresan para guardarse se hacen en un JSP (*IHContenido*) y es la parte que se actualiza según la distribución de la plantilla. Si el producto se crea sin ningún problema se muestra una pantalla compuesta (*IHProducto*), o una página de error (*IHError*). Además de los JSPs y del archivo de configuración xml todos los demás componentes se manejan como archivos con extensión java.
Al crear un producto puede ser de dos tipos, un PIE o una caja negra, en cada caso se presenta una interfaz de usuario adecuada. Si se trata de crear un PIE, la sección del contenido se sustituye por *IHCrearPIE.jsp*, un formulario con todos los campos adecuados para que el usuario pueda guardar todos los datos del producto. Si se trata de una caja negra la sección IHContenido se sustituye por *subirCajaNegra.jsp*, se presenta una opción para que el usuario pueda guardar en la base de conocimiento el archivo, existe una opción para que el usuario pueda guardar en la base de conocimiento el archivo sin importar su tipo. El diagrama de componentes se muestra en la Figura 5.3.

5.2.2 Diagrama de componentes del caso de uso "ConsultarProducto"

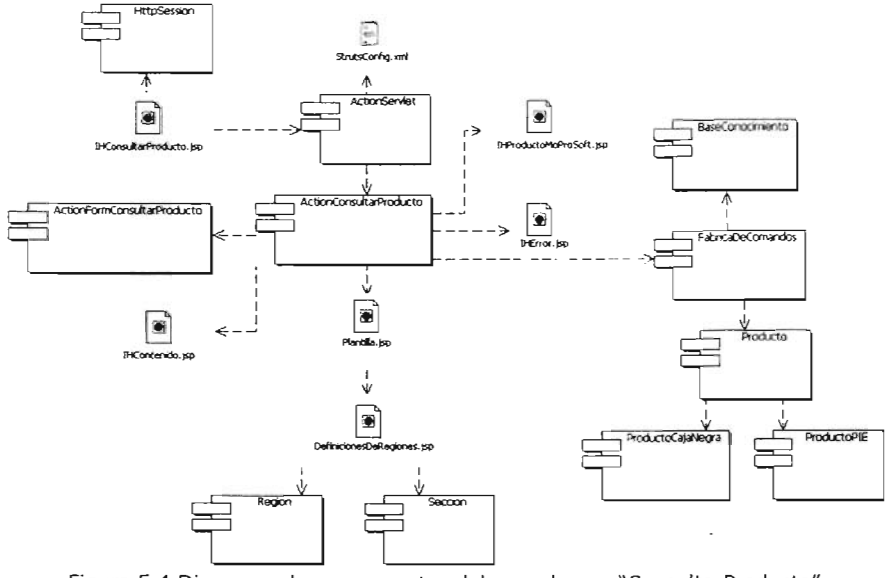


Figura 5.4 Diagrama de componentes del caso de uso "ConsultarProducto".

El diagrama de la Figura 4.20 muestra el diagrama de clases de diseño para el caso de uso especial ConsultarProducto, en el diagrama de la Figura 5.4 se muestran los componentes que se utilizan. Se sigue un esquema similar que en la creación de un producto. Al consultar un producto es posible que se trate de un PIE o de una caja negra, en cada caso se presenta una pantalla diferente al usuario. Para presentar un PIE el *jsp*, *IHContenido.jsp* se sustituye por *ConsultarPIE.jsp* que presenta directamente los atributos del producto en la pantalla a través de un formulario. En caso de presentar una caja negra la sección de contenido se sustituye por *descargarCajaNegra.jsp*, que presenta una opción para descargar el archivo que se pretende manejar. La caja negra se descarga y una herramienta externa permite su manejo como un procesador de textos, una herramienta de modelado, un ambiente de programación, etc.

5.2.3 Diagrama de componentes del caso de uso "ModificarProducto"

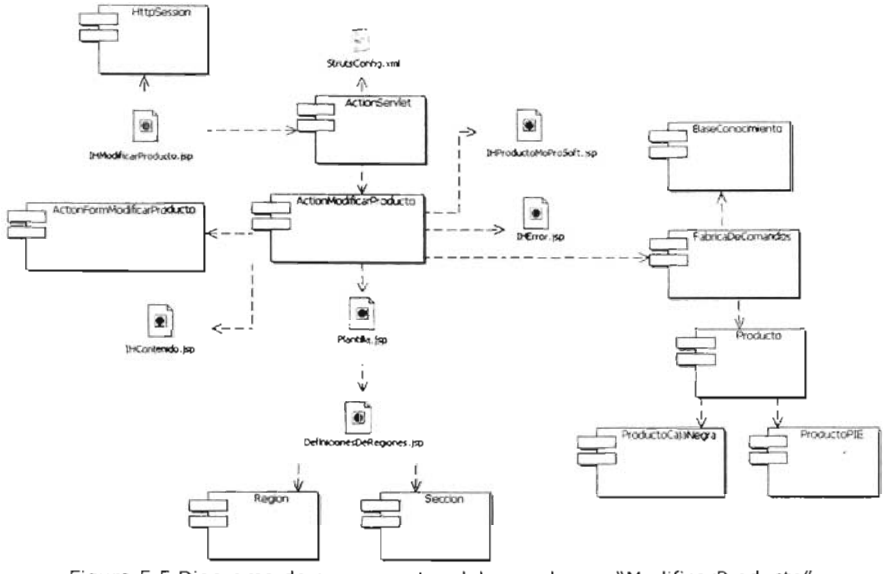


Figura 5.5 Diagrama de componentes del caso de uso "ModificarProducto".

Al modificar un producto también existe la posibilidad de que sea de cualquiera de los tipos, PIE o caja negra y en cada caso la información se presenta en una interfaz de usuario. Para modificar cualquiera de los productos se presenta una lista en *IHModificarProducto.jsp*, se selecciona el producto y se presentan los atributos en la sección de contenido, en ese momento se pueden editar y posteriormente se envían para su actualización, en el caso de un PIE.
En el caso de una caja negra es necesario descargar el archivo, esta opción se presenta en la sección de contenido mediante descargarCajaNegra.jsp, después de editarlo debe enviarse nuevamente a la base de conocimiento mediante subirCajaNegra.jsp. En el diagrama de la Figura 5.5 se muestra la implementación del caso de uso ModificarProducto.

5.3 Diagrama de componentes del caso de uso "Salir del Sistema"

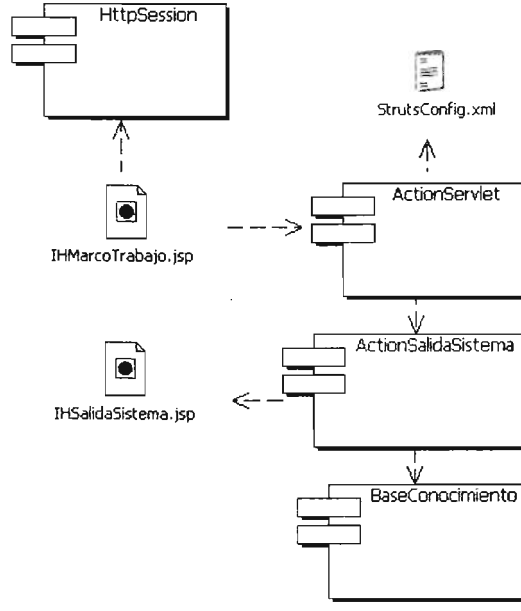


Figura 5.6 Diagrama de componentes del caso de uso "SalirDelSistema".

Para salir del sistema el usuario debe seleccionar la opción de salida desde un JSP que presenta información al usuario y la información de realimentación se presenta de igual forma en otro JSP. El componente que redirige a la salida es el archivo de configuración .xml que se relaciona con los componentes java necesarios para anular la sesión y los recursos utilizados a través de la fachada.

5.4 Diagrama de distribución de la herramienta HIM

El diagrama de distribución proporciona información sobre la configuración final de la herramienta HIM, informa cuáles componentes pertenecen a los diferentes nodos del sistema. En una aplicación puede existir más de un servidor por lo que los datos deben distribuirse en varios nodos.
En el caso de la herramienta HIM los componentes se pretenden distribuir en un solo punto que mantiene las páginas Web, las reglas de negocio y acceso a datos. Todos los datos manejados se guardan en el servidor y cada uno de los clientes puede acceder a ellos a través de un navegador Web.
En la Figura 5.7 se muestra el diagrama de distribución de los componentes generados en la herramienta HIM. Todas las clases y JSPs generados para el funcionamiento de la herramienta se guardan en un archivo WebModuleHIM.war.

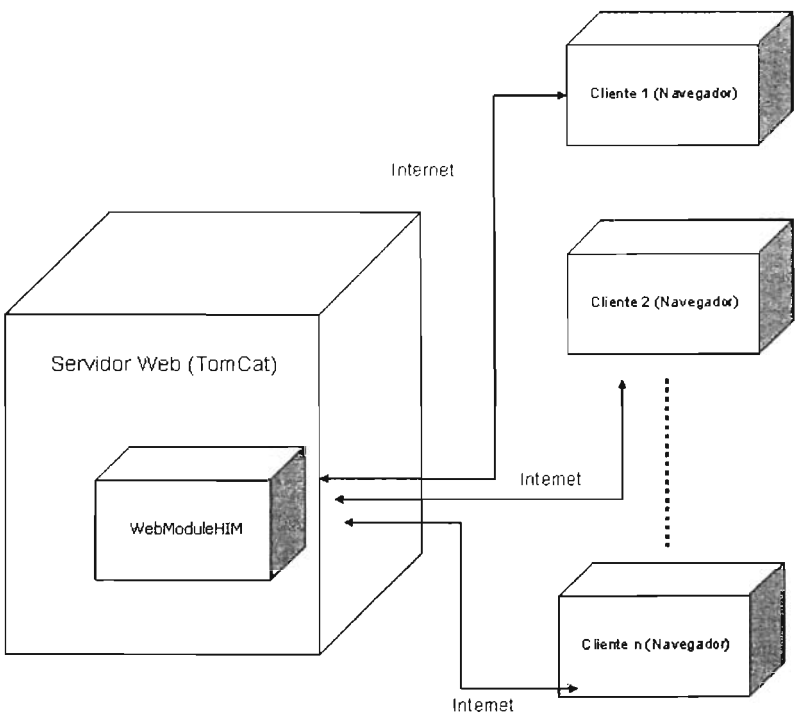


Figura 5.7 Diagrama de distribución de la herramienta HIM

La aplicación empaquetada en un módulo .war contiene todos los archivos necesarios para que la aplicación pueda alojarse en un servidor Web y ejecutarse como prueba.

No todos los productos que contempla MoProSoft se pueden manejar en ésta primera versión de la herramienta se define una serie de productos PIE´s que pueden servir de base para que en una versión futura la herramienta se extienda y se mejore contemplando todos los productos que el modelo MoProSoft requiere.

Algunos productos en los que la información es un poco más compleja y requiere de un procesador de textos probado o algún software en particular, se han manejado como cajas negras y únicamente se envían de un lugar a otro.

Los productos manejados como PIE´s se muestran en el navegador como datos de un formulario o como texto. En el caso de las cajas negras se presenta una opción para que se descargue el archivo o se haga disponible en el servidor. Después de descargar la caja negra será necesaria una aplicación particular para manejar el tipo de archivo que se trate.

CONCLUSIONES

MoProSoft es un modelo de procesos que incorpora una serie de actividades bien definidas para el desarrollo y la gestión de proyectos de software. Las actividades se relacionan unas con otras en base a una persona encargada de llevarlas a cabo, es decir, un rol. Se tiene una jerarquía de roles que se enfocan en alcanzar objetivos a diferente nivel en una empresa. Estos objetivos se cumplen y comunican a otros procesos a través de la generación de productos que sirven de entrada a otro proceso y se obtiene alguna realimentación importante generando documentos de salida, cada uno de los productos generados tienen un proceso de revisión y validación que lo habilita para que pueda servir de referencia.

J2EE es una especificación muy completa que utiliza a Java como lenguaje de referencia y la elección de los patrones que se utilizarán requiere un análisis de los elementos que constituye la especificación y acoplarlos a los requerimientos de un sistema.

La capa de presentación contiene un conjunto de patrones que permiten modularizar un poco más la aplicación. Cuando se tiene una división en módulos, el mantenimiento puede realizarse de manera más sencilla.

La utilización de algunos marcos de trabajo conocidos como *frameworks*, como en el caso de *Struts* permite la implementación más sencilla de los patrones. Con *struts* se implementan los patrones Front Controller y Validator. La especificación de Java para Web (JSPs y servlets) permite la implementación de los patrones *Composite View*, y *View Helper* mediante la definición de regiones y secciones en la interfaz de usuario y nuevas etiquetas dinámicas que pueden integrarse en los *jsps*.

También la utilización de una metodología en el desarrollo de la herramienta es muy importante, ya que *MoProSoft* no señala estrictamente cuál es el método al que debe apegarse el equipo de trabajo, al contrario deja abierta las posibilidades de ajustarse incluso a las buenas prácticas que ya se tienen en una organización. El equipo de trabajo de la herramienta HIM optó por el Proceso Unificado**[Jacobson]** de desarrollo de software como un método guía. El resultado es que no hay ninguna contrariedad al utilizarse *MoProSoft* como modelo de procesos y el proceso unificado como método de desarrollo.

La documentación generada en el desarrollo de la herramienta permitirá modificarla, mejorarla, agregarle nuevas funcionalidades que puedan incluso trascender el modelo de procesos. Queda pendiente la investigación de un *framework* para la aplicación de los patrones en la capa de presentación, el caso especial del *framework Tiles* permite la división de las pantallas en regiones y secciones, también *struts* permite la definición de plantillas que dividen una zona de trabajo en regiones. *Tiles* es compatible con struts de tal forma que se pueden realizar pruebas de definición de pantallas y la división de ellos en regiones y secciones, ésta puede ser una solución un poco más sencilla, sin embargo, se adquiere cierta dependencia al *framework* que se encuentra cambiando. La opción que se presenta aquí es implementada y puede lograrse una prueba para decidir si es conveniente adoptar *Tiles*.

La documentación generada en el diseño y el análisis puede utilizarse para lograr una implementación en otro marco de trabajo como .NET y cada una de las empresas que adopten *MoProSoft* como su guía de procesos pueden extender la herramienta HIM ó construir su propia herramienta y optimizar de alguna forma sus recursos.

La usabilidad del sistema y los criterios para el diseño de la interfaz de usuario son importantes porque definen la versión final del sistema. Estos dos temas no se han tratado en profundidad en éste trabajo y se dejan como una tarea futura, porque el manejo de la usabilidad y la optimización de la interfaz de usuario es una tarea que por sí misma demanda un trabajo aparte. Debe existir un estudio formal de la construcción de la interfaz de usuario con toda la complejidad que esto involucra, analizar las tareas del usuario, realizar pruebas y evaluar los resultados. Se han aplicado los patrones de la capa de presentación de J2EE que permiten la implementación de la construcción de la interfaz de usuario. Los patrones aplicados permiten la modificación de las interfaces y su integración de manera sencilla. También ocultan el código que pudiera tenerse en los distintos JSPs que se generaron , a través de etiquetas a la medida. La idea general es que en base a los resultados obtenidos en las pruebas con los usuarios, la interfaz pueda modificarse y adaptarse fácilmente, mediante el uso de los patrones, teniendo siempre como fuente principal las necesidades de los usuarios y no la construcción del sistema para que el usuario se adapte a él.

BIBLIOGRAFIA

[Alex] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King Shlomo Angel "A Pattern Language", Oxford University Press, New York, 1977.

[Alexander] C. Alexander. The Timeless Way of Building. Oxford University Press, 1979.

[Cavaness] Chuck Cavaness, "Programming Jakarta Struts", O'Reilly. 2002.

[Coplien] J.O. Coplien, "Advanced C++ - Programming Styles and Idioms", Addison-Wesley, 1992.

[Cruz] Jorge Cruz Vázquez, "Análisis e implementación de esquemas de seguridad aplicados a la herramienta integral MoProSoft (HIM)", Tesis de Maestría en Ingeniería. UNAM. 2005.

[Deepak] Deepak Alur, John Crupi, Dan Malks. "Core J2EE Patterns. Best Practices and Design Strategies", Sun Microsystems, 2002.

[Dix] Dix A. "*Human computer interaction*". Prentice Hall, 1993.

[Falkner] Jayson Falkner, Ben Galbraith, Romin Irani, "Fundamentos. Desarrollo Web con JSP", 1ª Edición , Anaya Multimedia, 2001.

[Ford] Neal Ford. "Art of Java Web Development", Manning Publications Co. 2004

[Fowler] Martin Fowler, "Analysis Patterns: Reusable Object Models", Addison Wesley, 1997.

[Gabrick] Kurt A. Gabrick, David B. Weiss. "Java 2EE and XML Development", Manning Publications Co., Greenwich. 2002.

[Gabriel] Richard P. Gabriel, "Patterns of Software: Tales from the Software Community", Oxford University Press, 1998.

[Gamma] E. Gamma, R. Helm, R. Johnson, & J. Vlissides. Design Patterns: Elements of Object-Oriented Software. Addison-Wesley, 1995.

[Hagemo] Lloyd Hagemo & Ravi Kalidindi. "Creating a Framework-J2EE pattern frameworks provide template for flexible and modular architecture". IBM. 2004.

[Jacobson] Ivar Jacobson, Grady Booch, James Rumbaugh. "El Proceso Unificado de Desarrollo de Software". Addison Wesley-Pearson Education, 2000.

[Laurel]. Laurel B. "The art of human-computer interface design". Addison-Wesley. 1992

[Lorés]. Jesús Lorés, Toni Granollers, Sergi Lana, "Introducción a la Interacción Persona-Ordenador". 2002

[Nielsen] Nielsen J. "*Usability engineering*". AP Professional. 1993

[Norman] Norman D. "*The invisible computer*". MIT Press, 1999

[Stelting] Stephen Stelting, Olav Maassen. "Patrones de diseño aplicados a JAVA", Pearson, Prentice Hall. Primera edición. 2003.

[Uribe] Ernesto Uribe Hernández, "Uso de la tecnología RDF para representar y manejar los procesos MoProSoft y su aplicación en HIM", Tesis de Maestría en Ingeniería. UNAM. 2005.

[Vázquez] Marcos O. Vázquez Morales, "Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para MoProSoft (HIM)", Tesis de Maestría en Ingeniería. UNAM. 2005.

[Yacoub] Sherif M. Yacoub, Hany H. Ammar. "Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems", Addison Wesley. 2003.

MoProSoft. Modelo de Procesos para la industria de Software. Version 1.1
http://www.software.net.mx/

[1] http://www.jboss.org/
[2] http://jonas.objectweb.org/
[3] http://www.bea.com/
[4] http://www-306.ibm.com/software/websphere/