VNIVERSIDAD NACIONAL AVPNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para 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:

MARCOS OSCAR VÁZQUEZ MORALES

DIRECTOR DE TESIS: DRA. HANNA OKTABA

México, D.F. **2005.**

Índice

Introducción.

En la actualidad algunas empresas que se dedican al desarrollo de software en México no utilizan metodologías ó modelos que les permitan construir software de calidad. La negativa de tal adopción en algunos casos radica en el desconocimiento ó mala interpretación de las normas existentes.

En un esfuerzo por incrementar la calidad en el desarrollo de software en México, la Secretaria de Economía en convenio con la UNAM realizaron un documento cuyo objetivo es servir de base de una norma mexicana para la industria de desarrollo y mantenimiento de software. Dicho documento adoptó el nombre de MoProSoft© (Modelo de Procesos para la Industria de Software) [OKT+03].

MoProSoft presenta un modelo de procesos que fomenta la estandarización de su operación a través de la incorporación de las mejores prácticas en gestión e ingeniería de software. La adopción del modelo permite elevar la capacidad de las organizaciones para ofrecer servicios con calidad y alcanzar niveles internacionales de competitividad.

Como un complemento de MoProSoft, surgió la iniciativa de poder desarrollar una herramienta que fuese capaz de poder gestionar las propias actividades de MoProSoft. Pero **¿por qué desarrollar una herramienta para MoProsoft?. La automatización de MoProSoft busca simplificar aun más la adopción de las prácticas en ingeniería de software, organizando y controlando todos y cada uno de los productos[1] que son generados.**

Así los requerimientos generales de la herramienta contemplan la gestión de cada una de las actividades integradas en los procesos de MoProSoft. Dichas actividades pueden generar, consultar y modificar productos dependiendo del proceso en el que se este interactuando. Cada uno de los productos generados será almacenado en un repositorio llamado "Base del Conocimiento". La funcionalidad de la herramienta busca además proporcionar un mecanismo de seguridad en lo que respecta a la manipulación de los productos, es decir, cada producto solo puede ser generado, consultado ó modificado por la persona ó personas que tengan la facultad (rol) de hacerlo.

La herramienta encargada de gestionar las actividades de MoProSoft adoptó el nombre de HIM (Herramienta Integral para MoProSoft). Para realizar el desarrollo de HIM se conformó un equipo de trabajo de seis integrantes.

---

[1] Un producto en MoProSoft es cualquier elemento que sea generado en un proceso.

Al realizar el análisis de requerimientos de HIM, se concluyó que la herramienta podía ser dividida para su implementación en tres capas ó módulos (capa de la vista, capa del control y capa del modelo) y que cada capa internamente podría contar con un mecanismo de seguridad (capa de seguridad).

En la figura 1 se muestra el diagrama general de casos de uso de HIM, en el cual se plasma la funcionalidad de la herramienta a grandes rasgos.

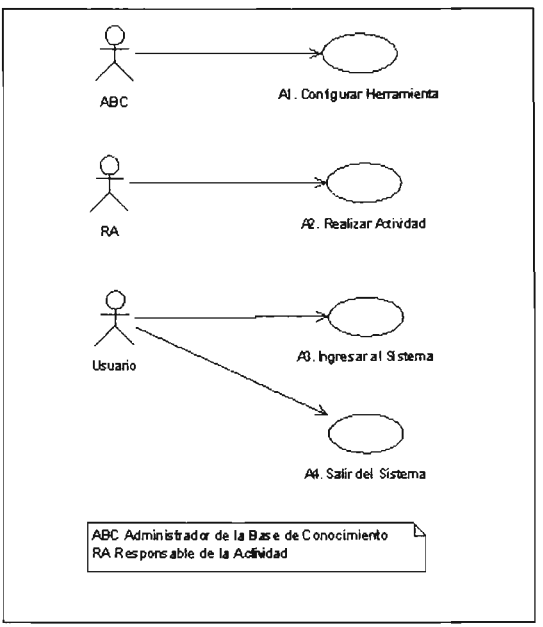


Figura 4.1. Diagrama general de casos de uso de HIM.

El caso de uso "A1" es el encargado de realizar la configuración de la herramienta. Esta configuración consta de la instalación del repositorio. El repositorio es el lugar donde se guardan todos los productos generados en MoProSoft. La implementación de este caso de uso (no cubierto en esta tesis) fue llevada a cabo por la capa de seguridad y la capa del modelo de HIM.

En el caso de uso "A2" se lleva a cabo todo el proceso de creación, consulta y modificación de un producto en MoProSoft dependiendo de la actividad que lo gestione.

El caso de uso "A3" es el encargado de gestionar el proceso de ingreso al sistema. Este caso de uso fue implementado conjuntamente por la capa de control y la capa de seguridad de HIM.

El caso de uso "A4" es el encargado de salir del sistema, que consta de la liberación de todos los recursos que utilizó un usuario al trabajar con HIM.

Dependiendo de los factores relacionados con las cuatro capas, cada uno de los integrantes del grupo de trabajo decidió investigar y realizar su respectiva tesis con los siguientes temas (ver tabla 1):

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. Temas de tesis del grupo desarrollador de HIM.

El desarrollo de una de las cuatro capas conocida como **"la capa de control"** es la abarcada en el contenido de esta tesis la cual como se aprecia en la tabla tiene por título: "Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para MoProSoft (HIM)".

La capa de control (de manera general) es la encargada de procesar, coordinar y dirigir cada una de las peticiones generadas por el usuario hacia la herramienta, es decir, toda acción ejecutada por el usuario al navegar por la funcionalidad de HIM siempre será manejada por la capa de control, la cual, atenderá la petición y visualizará los resultados.
Por ejemplo cuando el usuario ingresa al sistema, la capa de control intercepta dicha petición e interactúa tanto con la capa de seguridad como con la capa de modelo para verificar que dicho usuario existe y que cuenta con un Rol específico. Hecho lo anterior, la capa de control interactúa con la capa de la vista para que se visualicen las interfaces correspondientes a tal Rol.

Para poder abarcar dicho tema de tesis es necesario conocer los objetivos sobre los cuales está sustentada la tesis:

- Revisar, analizar y seleccionar los patrones J2EE que más se adecuen a la capa de control de la herramienta.
- Implementar los patrones seleccionados en la capa de control de la herramienta.
- Aprovechar la funcionalidad que proporciona el marco de trabajo conocido como "struts" para el desarrollo de la herramienta en la capa de control.
- Integrar la capa de control con el resto de las capas para poder así utilizar la herramienta.

Con base en lo anterior es conveniente tener una breve descripción sobre el contexto en el cual se basan tales objetivos. A continuación se presentan algunos antecedentes que ayudarán al lector a conocer factores importantes contemplados en los objetivos.

Antecedentes

A lo largo de los años los ingenieros de software se dieron cuenta que en ciertas ocasiones los problemas que se presentaban en el desarrollo de software para un determinado proyecto tendían a repetirse en otro proyecto diferente. Por ello si se encontraba una cierta solución un tanto cuanto generalizada (abstracta) para tal problema, entonces esa misma solución sería útil para resolver un problema del mismo contexto pero en un proyecto diferente.

De aquí, surgió el concepto de "patrón" que se refiere a la solución recurrente que se le da a un problema dentro de un "contexto" (entorno, situación o condiciones interrelacionadas dentro de las cuales existe algo) determinado.

Java 2 Edición Empresarial (J2EE) es un estándar creado por ciertas empresas de software para el desarrollo de aplicaciones empresariales. Con el transcurso del tiempo y con el desarrollo cada vez más frecuente de tales aplicaciones, se empezó a documentar (por la misma comunidad de desarrolladores) una serie de patrones que se utilizan en el desarrollo de dichas aplicaciones.

Para el desarrollo de la Herramienta Integral para MoProSoft (HIM), se implementó aquellos patrones J2EE que más se adecuaron a las necesidades de la herramienta.

En lo que respecta a la arquitectura de la herramienta se utilizó el patrón conocido como Model-View-Controller (modelo-vista-control), el cuál divide el diseño de la herramienta en tres capas. Cabe señalar que el "control" es el encargado de manejar la implementación de la lógica del negocio (requerimientos de la herramienta HIM), para ello interactúa tanto con la capa de "vista" como con la capa del "modelo" para lograr la completa funcionalidad de la herramienta.

Existen en la actualidad "*Frameworks*" también conocidos como "marcos de trabajo" que proporcionan la implementación de algunos patrones. El "*Framework*" conocido con el nombre de "*Struts*" ayuda en gran medida a implementar la lógica del negocio de la herramienta, que como ya se mencionó es una de las funciones encargadas de la capa de control y el motivo por el cual se abarca en el contenido de esta tesis.

Una vez que el lector se adentró de una forma breve al contexto del tema de esta tesis, a continuación se describen de una manera rápida los capítulos que la conforman.

Descripción de capítulos

El contenido de esta tesis está conformado por cinco capítulos.

- En el capítulo uno titulado "Patrones" se presenta el origen, descripción y clasificación de los patrones de diseño GoF (Gang of Four) que fueron utilizados en el desarrollo de HIM.

- En el capítulo dos titulado "Arquitectura de HIM" se abarca la arquitectura utilizada en HIM, para ello se presenta los tipos de arquitectura más usados y los patrones J2EE. Con la selección e interacción de los patrones J2EE adecuados se constituyó la arquitectura de HIM.

- Struts es un marco de trabajo que permite la implementación del patrón J2EE conocido como "*Front Controller*". Debido a que tal patrón es de suma importancia para el desarrollo de HIM, en el capítulo tres titulado "Struts" se presenta su funcionamiento.

- En el capítulo cuatro titulado "Análisis y diseño de la capa de control de HIM" se presenta el análisis y diseño de los casos de uso de la herramienta. El proceso del análisis abarca la arquitectura propia de la capa de control, los diagramas de clases del análisis, los diagramas de clases del diseño y el diagrama de subsistemas.

- En el capítulo cinco titulado "Implementación de la capa de control de HIM" se presenta la fase de construcción de la herramienta. Dicha fase de construcción abarca los diagramas de componentes y el diagrama de despliegue.

Para terminar se presentan las conclusiones generadas en el desarrollo de la capa de control de HIM.

CAPÍTULO 1. PATRONES

Este capitulo describe los orígenes, clasificación y documentación de los patrones de diseño utilizados en la capa de control de la Herramienta Integral para MoProSoft (HIM).

Hoy en día en el proceso de creación de aplicaciones de *software* se deben de contemplar ciertas características que le proporcionen a dicha aplicación el atributo de alta calidad.

Características como escalabilidad[2], portabilidad[3], disponibilidad[4] y reusabilidad[5] le permiten a las aplicaciones ser eficientes y duraderas. Para la obtención de estas características los diseñadores de *software* deben de apoyarse en soluciones bien documentadas que, por ser muy abstractas (generales) pueden aplicarse a la mayoría de los proyectos dependiendo de sus requerimientos.

Estas soluciones evolucionaron en el concepto conocido como "**patrón**" que se refiere a la solución bien documentada que los expertos aplican para resolver nuevos problemas debido a que dichas soluciones ya han sido utilizadas con éxito en el pasado [SM02].

1.1 Orígenes.

La idea de los patrones de diseño fue concebida por *Christopher Alexander* miembro de la Universidad de *Berkeley*. En el final de la década de los setentas escribió varios libros dentro de los cuales figura "*A pattern Language*" [AIS+77]. En este libro se exhibe el concepto de patrón aplicado a arquitectura, además de presentar una clasificación de los mismos.

El trabajo de *Alexander* atrajo la atención de la comunidad de la programación orientada a objetos (OOP) de ese entonces y a lo largo de esa década algunos pioneros desarrollaron los primeros patrones para el desarrollo de *software*. Entre los pioneros más destacados de esa época estuvieron *Kent Beck* y *Ward Cunningham* los cuales presentaron varias conferencias donde se discutían algunos patrones de diseño para *SmallTalk*. Tiempo después *James Coplien* escribió uno de los primeros libros de patrones [Cop92] para el desarrollo en C++ a principios de los noventas.

---

[2] Capacidad de soportar mas carga de trabajo sin necesidad de modificar el software (solo se añade más hardware).
[3] Una aplicación puede ser ejecutada en cualquier sistema operativo.
[4] Una aplicación está disponible 365 dias las 24 horas.
[5] Capacidad de reutilizar componentes de software de otros proyectos.

La contribución más importante a los patrones fue el libro que se publicó en 1995 cuyo titulo es "Patrones de diseño" [GHJ+95] el cual fue escrito por *Erich Gamma, Richard Helm, Ralph Johnson* y *John Vlissides* comúnmente bautizados con el sobrenombre de *Gang of Four* (GoF) ó la pandilla de los cuatro.
El contenido de dicho libro presentaba una amplia gama de patrones junto con diversos ejemplos en el lenguaje de programación C++, esto ayudó de forma substancial en la correcta comprensión de cada uno.

Con la publicación de estos libros en conjunto con su comprobada efectividad empezó a surgir el interés por toda la comunidad de *software* para utilizar tales patrones.

En general un patrón consta de cuatro elementos esenciales

1. **Nombre del patrón**. Teniendo un nombre, cada patrón puede ser fácilmente identificado por la comunidad que los consulta.

2. **Problema**. Se explica cuando debe de ser aplicado el patrón. Aquí se describe el problema y el contexto en el cual se presenta. En algunas ocasiones se pueden describir problemas específicos de diseño, es decir, como clases o estructuras de objetos. Algunas veces el problema debe incluir una lista de condiciones que se deben conocer previamente.

3. **Solución**. Se describen los elementos que componen el diseño, su relación, sus responsabilidades y colaboraciones. La solución no debe describir un diseño concreto en particular ó implementación, debido a que un patrón es como una plantilla que puede ser aplicada en muchas situaciones diferentes, por ello el patrón proporciona una descripción abstracta de un problema y como un arreglo de elementos, ya sean clases u objetos, lo resuelven.

4. **Consecuencias**. Son el resultado de la aplicación del patrón. Típicamente las consecuencias son indispensables para la evaluación de las alternativas de diseño y, además, con ellas se obtienen los costos y beneficios de aplicar el patrón. Dado que la reutilización es un factor siempre considerado en el diseño orientado a objetos, las consecuencias de un patrón plasman el impacto en la portabilidad, disponibilidad, o escalabilidad de un sistema.

1.2 Descripción de los patrones de diseño.

Lo que se buscó desde un principio en los patrones de diseño fue su estandarización en lo concerniente a la información sobre un problema en común y su respectiva solución. Para ello *Alexander* desarrollo una plantilla (mencionada previamente) en la cual se describía el patrón. La plantilla en su forma más simple contaba con cuatro áreas que formalizaban la explicación del patrón y su solución.
En la mayoría de la literatura estas cuatro áreas siempre están presentes, sin embargo, para una mejor comprensión en dicha plantilla se fueron incorporando más aspectos.

Aunque las plantillas varían de acuerdo a las necesidades de los desarrolladores la esencia descriptiva del patrón de diseño siempre es la misma.

A continuación se presenta el formato de la plantilla de los patrones de diseño GoF, él cual, fue tomado como referencia [SM02].

- Nombre. Es el nombre descriptivo del patrón.
- También conocido como. Algunos patrones cuentan con un nombre alternativo.
- Propiedades. Indica la clasificación o grupo al cual pertenece el patrón.
- Propósito. Descripción breve de lo que hace el patrón.
- Motivación. Se realiza una pequeña descripción de un problema que puede ser resuelto con dicho patrón.
- Aplicabilidad. Son las situaciones (el cuando y el por qué) en las cuales es deseable utilizar dicho patrón.
- Descripción. Aquí el patrón ya es descrito en su totalidad y se indica que es lo que hace y como se comporta comúnmente con diagramas de clases.
- Implementación. Se indica lo que se debe hacer para implementar dicho patrón por lo general se visualiza con un diagrama de clases.
- Ventajas e inconvenientes. Se mencionan las consecuencias de usar el patrón y los compromisos que se presentan en el uso del mismo.
- Variantes. Se analizan posibles alternativas de implementación del patrón.
- Patrones relacionados. Se mencionan los patrones con los que se está asociado o aquellos con los que se tiene una relación.
- Código de ejemplo. Se ilustran fragmentos de código que permiten la implementación del patrón.

Gracias a los patrones de diseño se ha evolucionado tanto en la **abstracción** como en la **reutilización** del *software*, que son dos de los conceptos más importantes en la programación orientada a objetos. En la abstracción se busca resolver los problemas complejos dividiéndolos en problemas más simples, las soluciones a dichos problemas pueden ser utilizadas en conjunto para resolver proyectos complicados. Por su parte la reutilización es de vital importancia para el desarrollo de software, de hecho, siempre se buscan formas de poder reutilizar el código existente. Estas dos características casi siempre son consideradas en los patrones existentes.

1.3 Clasificación de los patrones de diseño.

Debido a que los patrones de diseño varían en su nivel de abstracción se necesita una forma de organizarlos. Existen dos maneras de clasificar dichos patrones de diseño [GHJ+95].

En la primera forma conocida como de **propósito** se refleja lo que hace el patrón en si, es decir, los patrones pueden tener un propósito de **creación**, **estructural** ó de **comportamiento**.

Los patrones de creación como su nombre lo indica se encargan de crear objetos. Ahora bien, los patrones estructurales tratan ó gestionan las relaciones estáticas y la composición de clases y objetos, mientras que los patrones de comportamiento coordinan la forma en que interactúan y se distribuye las responsabilidades entre clases y objetos.

La segunda forma conocida como de **alcance** especifica si el patrón se aplica primariamente a las clases o a los objetos. Los patrones de **clases** tratan con la relación que existe entre las clases y sus subclases, estas relaciones son establecidas a través de la herencia, es decir, se fijan en tiempo de compilación. Los patrones de **objeto** tratan con las relaciones entre objetos, es decir, puede ser cambiadas en tiempo de ejecución. A continuación se muestra en la tabla 1.1 estas dos clasificaciones.

J		Propósito		
		Creación	Estructural	Comportamiento
Alcance	Clase	*Factory Method*	*Adapter(clase)*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter(objeto)* *Bridge* *Composite* *Decorador* *Facade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Tabla 1.1. Clasificación de los patrones de diseño

1.4 Patrones de diseño GoF aplicados a HIM.

Según los requerimientos que se presentaron en el desarrollo de la capa de control de (HIM), algunos de los patrones mostrados en la tabla anterior, fueron utilizados. Por ello, a continuación se realizará una breve descripción de éstos para que el lector pueda tener una idea de cómo funcionan. Cabe destacar que es una descripción breve y en capítulos posteriores se ahondará en su implementación.

1.4.1 Patrón *Abstract Factory* (Fábrica Abstracta).

- También conocido como: *ToolKit* (grupo de herramientas).
- Propiedades: patrón del tipo creación-objeto.
- Propósito: lleva a cabo la creación de familias de objetos que pueden ó no estar relacionados sin tener que especificar una clase concreta[6].
- Motivación: si se requiere realizar por ejemplo la administración de datos personales, originalmente se podrían crear clases que representen dichos datos, sin embargo estos datos pueden variar de país en país lo que acarrearía la modificación constante del código. Una solución más factible sería añadir de forma flexible estas clases al sistema. Usando este patrón se podría crear una fábrica que contenga los datos personales y en tiempo de ejecución se crearían fábricas concretas para distintos países. Así no se recurriría a la modificación constante del código, sólo se agregaría otra fábrica de comandos que heredará todas las características comunes de la fábrica original.
- Aplicabilidad: **se puede usar el patrón cuando la aplicación en cuestión deba configurarse con una ó más familias de objetos, ó también cuando se quiera proporcionar una colección de clases y únicamente se muestren las firmas de los métodos sin que éstos sean implementados.**
- Descripción: en ciertas ocasiones existen aplicaciones que necesitan utilizar varios recursos (sistema de archivos, gestión de ventanas, etc.) por ello se busca que sean capaces de utilizar los diversos recursos sin tener que reescribir de nuevo el código cada vez que se introduce un recurso nuevo. Para resolver este problema se define un creador de recursos genéricos o sea el patrón fábrica abstracta en donde dicha fábrica cuenta con uno o más métodos de creación que pueden ser invocados para generar recursos genéricos (clases abstractas).

- Implementación: por simplicidad, la implementación de este patrón se realizará en la parte de análisis y diseño de la capa de control. por lo que se mostrará en el capitulo 4.
- Ventajas e inconvenientes: durante el diseño no se necesita predecir todos los usos futuros de dicha aplicación, en su lugar se crea un marco general de trabajo en donde se desarrollan implementaciones independientes del resto de la aplicación, esto trae consigo que en tiempo de ejecución pueden integrarse nuevas características y recursos. Una de sus principales desventajas es que si los productos abstractos no se definen apropiadamente resultaría muy difícil definir los productos concretos de esos productos abstractos.
- Variación del patrón: la principal variación viene del hecho de que según los requerimientos, se puede definir el producto como una clase abstracta[7] ó como una *interfase*[8].

---

[6] Aquella que puede ser instanciada, es decir, se crea un objeto de ella.
[7] puede tener métodos con solo su firma ó también puede tener métodos implementados.
[8] Solo contiene la firma de los métodos, ninguno debe de ser implementado.

- Patrones relacionados: cada fábrica abstracta debe de contener métodos de fabricación (patrón *Factory Method*), es decir, aquellos métodos que forman parte de la fábrica. Otro de los patrones que debe de interactuar con la fábrica abstracta es el que se conoce como *Singleton*, el cual se utiliza frecuentemente en la fábrica concreta.
- Código de ejemplo: el código de ejemplo está disponible en la sección de anexos.

La utilización de este patrón podrá auxiliar a la optimización de HIM en versiones futuras debido a que la forma de gestionar los productos de MoProSoft en esencia siempre será la misma (crear, consultar y modificar). Con ello, en futuras versiones pueden agregarse nuevas características o recursos a estas tres operaciones sin modificar la estructura original. De aquí, se tendrá una fábrica de HIM (ver Figura 1.1) que contenga tres métodos correspondientes a las tres operaciones. Cada método se apega a las características descritas por el patrón *Factory Method* que se presenta en la siguiente sección.

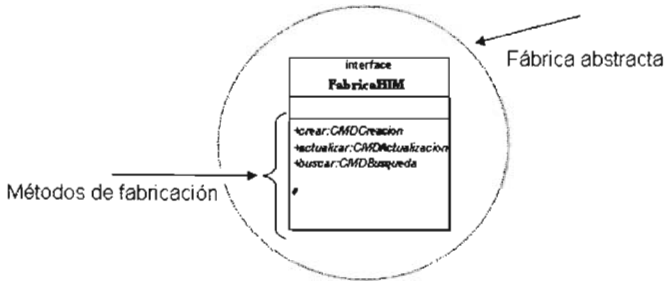


Figura 1.1. Fábrica abstracta de HIM.

1.4.2 Patrón *Factory Method* (Método de Fabricación).

- También conocido como: *Virtual Builder* (constructor virtual).
- Propiedades: patrón del tipo creación-clase.
- Propósito: define un método estándar para crear un objeto, es decir, adentro del método de fabricación se crea y retorna el objeto concreto.
- Motivación: retomando el ejemplo del patrón anterior de la gestión de datos personales, se requiere que las tareas asociadas a los datos tengan la capacidad de poder controlar sus propios editores para administrar las adiciones y cambios de dicha información. Por lo tanto se puede definir un método de fabricación en la fábrica abstracta, que retorne un editor genérico y así pueda ser utilizado de una manera concreta por la clase que representa alguna información específica, además de controlar sus propios cambios.
- Aplicabilidad: **este patrón puede ser utilizado cuando se requiera proporcionar cierta libertad en el tipo específico de objeto que se pueda crear en momentos posteriores.**

- Descripción: cuando se quiere crear una aplicación, en ciertos momentos se tiene claro que tipo de componentes se van a utilizar, pero en ocasiones la implementación se realiza en momentos posteriores, y allí es donde surgen requerimientos, que no se tomaron en cuenta. Para combatir estos contratiempos se pueden utilizar *interfases* para implementar dichos componentes, sin embargo, como a partir de las *interfases* no se pueden generar objetos, se necesita una clase que las implemente introduciendo el código necesario en sus métodos, dichos métodos son los métodos de fabricación.
- Implementación: por simplicidad la implementación de este patrón se realizará en la parte de análisis y diseño de la capa de control, por lo que se mostrará en el capitulo 4.
- Ventajas e inconvenientes: la principal ventaja retomando el ejemplo, es que en la aplicación solo se necesita saber como se solicitará el editor de un elemento y dicho editor se encargará de realizar los cambios al elemento. La desventaja de este patrón surge del hecho de que cada que se quiera agregar un nuevo elemento, se tiene que crear una clase que implemente la *interfase* que contiene el método de fabricación y en dicha clase introducir la lógica para la edición del elemento.
- Variaciones: el método de fabricación puede recibir algún parámetro.
- Patrones relacionados: como el patrón anterior (fábrica abstracta) requiere de los servicios del patrón en cuestión, entonces, ambos patrones se relacionan del hecho de que se pueden utilizar uno ó más métodos de fabricación definidos en la fábrica abstracta.
- Código de ejemplo: el código de ejemplo está disponible en la sección de anexos.

Desde el contexto de HIM, cada método de fabricación es representado por las tres operaciones básicas de un producto en MoProSoft, es decir, se tendría un método de fabricación para la creación de un producto, se tendría un método de fabricación para la consulta de un producto y se tendría un método de fabricación para la modificación de un producto. Cada método formaría parte de la fábrica tal como se plasmó en la Figura 1.1.

1.4.3 Patrón *Command* (Comando).

- También conocido como: *Action* (acción).
- Propiedades: patrón del tipo comportamiento-objeto.
- Propósito: poder encapsular un comando (método) en un objeto de tal forma que adquiera las propiedades de los objetos, es decir, dicho comando pueda ser pasado como parámetros de un método, ó en su caso, dicho comando pueda ser proporcionado como valor de retorno de un método.
- Motivación: cuando en una aplicación un usuario ejecuta una acción, dicha aplicación debe de tener la capacidad de poder revertir la acción recientemente ejecutada. Para ello se pueden plasmar las acciones que un usuario puede ejecutar en un objeto (el objeto comando). Con esto solo se necesita que una aplicación invoque al método conocido como "ejecutar" del objeto comando y acto seguido se ejecuta dicha acción. Para el caso de que se requiera revertir la acción recién ejecutada, basta que la aplicación invoque al método conocido como "deshacer" del objeto comando y así se regresa a la situación original.

- Aplicabilidad: **este patrón puede ser utilizado cuando se quiera dar soporte para deshacer comandos recientemente ejecutados.**
- Descripción: el patrón comando tiene la capacidad de encapsular tanto su funcionalidad como sus datos para que posteriormente las acciones de un usuario puedan ser ejecutadas. Por ello una aplicación que pretenda utilizar dicho patrón, debe de contener una fuente donde el usuario puede disparar cualquier acción (podría ser una interfaz gráfica), también debe de contener un receptor que atienda la acción disparada por el usuario (un servlet) y por último debe de contar con el comando en sí que será el que ejecute la acción.
- Implementación: por simplicidad la implementación de este patrón se realizará en la parte de análisis y diseño de la capa de control, por lo que se mostrará en el capítulo 4.
- Ventajas e inconvenientes: como el comando se encapsula en un objeto, por lo tanto, dicho comando puede ser utilizado como un objeto normal adquiriendo todas las propiedades de los objetos. A demás si se quiere incorporar nuevos comandos solo se necesita realizar la implementación de la *interfase* que lo contiene.
- Variaciones: como ya se menciono brevemente, dicho patrón puede contener un método deshacer que sea capaz de poder revertir la última acción ejecutada por el usuario, para ello, se necesita mantener una referencia a el último comando.
- Patrones relacionados: se puede utilizar el patrón conocido como *Memento* para poder guardar el estado del receptor y así deshacer el comando.
- Código de ejemplo: el código de ejemplo está disponible en la sección de anexos.

Cada método de fabricación de HIM entonces se convierte en un comando, el cual puede ser revertido si así se requiere (ver Figura 1.2).

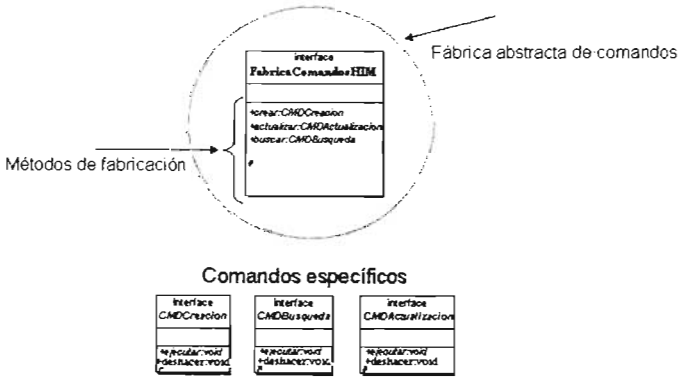


Figura 1.2. Fábrica abstracta de comandos de HIM.

Estos son los patrones GoF que se utilizaron en la implementación de la capa de control de la herramienta. Dichos patrones sirven de ayuda ó como complemento a los patrones de diseño de Java 2 *Enterprise Edition* (J2EE).

1.5 Introducción a los patrones de diseño J2EE.

J2EE es un grupo de especificaciones (también conocido como plataforma) diseñadas por SUN[9] que permiten la creación de aplicaciones empresariales. Es importante hacer notar que J2EE es sólo una especificación, lo cual permite que diversos productos sean diseñados alrededor de estas especificaciones. Dentro de los componentes, que forman parte de ésta plataforma, se encuentran: Java Database Conectivity (JDBC[10]) 2.0, JavaMail[11] 1.1, JavaServer Pages (JSP[12]) 1.2, Servlet[13] (2.3) entre otros.

Los patrones de diseño J2EE fueron documentados para ayudar a diseñar aplicaciones que son ejecutadas en entornos Web. Cabe señalar que la herramienta HIM en su conjunto está diseñada para ser ejecutada en navegadores Web, por lo tanto los patrones de diseño J2EE también fueron analizados. Tales patrones son agrupados en 3 categorías: capa de presentación, capa del negocio y capa de datos (ver Figura 1.3).

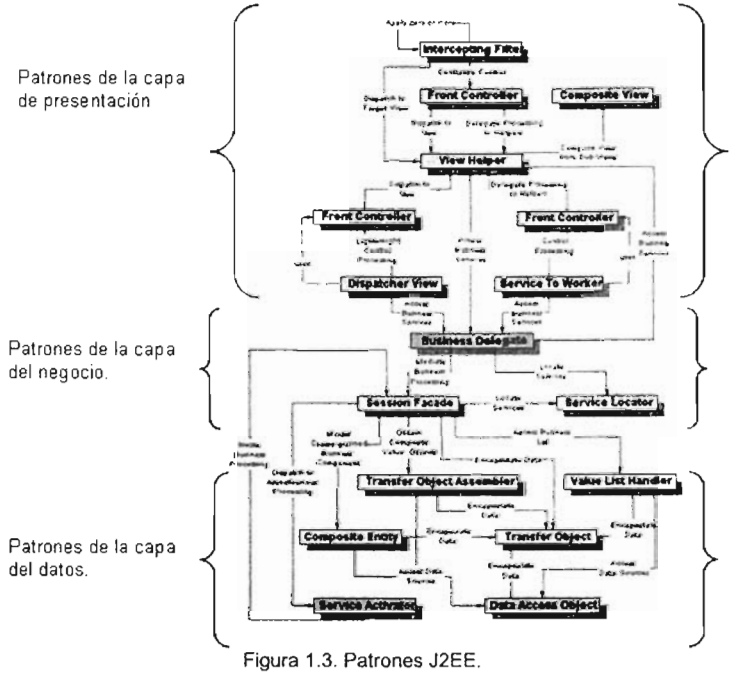


Figura 1.3. Patrones J2EE.

[9] Empresa de software.
[10] Conjunto de clases diseñadas para la conexión a bases de datos.
[11] Conjunto de clases diseñadas para la manipulación de correo electrónico.
[12] Plantilla para una pagina Web que emplea código Java para generar un documento HTML dinámicamente
[13] Son clases de java que amplian la funcionalidad de un servidor Web mediante la generación dinámica de páginas Web.

Estas capas son similares a las del patrón de arquitectura MVC (capa de vista, capa de control y capa de datos). La diferencia radica en que el patrón MVC es más abstracto, puede utilizarse tanto en aplicaciones Web como en aplicaciones de escritorio sencillas, mientras que los patrones J2EE están diseñados para utilizarse en entornos Web un poco más complejos y robustos que necesitan una mejor gestión en lo que respecta a los recursos que emplean para cumplir con la funcionalidad de la aplicación en cuestión.

Los patrones J2EE presentados en la figura anterior fueron analizados y seleccionados de acuerdo a los requerimientos de HIM. En la sección 2.4 del siguiente capítulo se describen los patrones J2EE que fueron utilizados en la implementación de la herramienta.

Existe un lenguaje de patrones para J2EE que se encarga de realizar la agrupación de algunos patrones de arquitectura, patrones de diseño GoF y patrones J2EE de tal forma que se pueda obtener una arquitectura robusta y bien diseñada, que proporciona las bases para el diseño de aplicaciones Web. Este lenguaje es conocido como E++ [DE1]. En la sección 2.2 del siguiente capítulo se describen los elementos más importantes de este lenguaje.

CAPÍTULO 2. ARQUITECTURA DE HIM BASADA EN LOS PATRONES J2EE

Este capítulo describe la arquitectura utilizada por la herramienta HIM, de allí se parte para analizar los patrones de diseño J2EE que forman parte de la capa de control de HIM.

En el título de éste capítulo se menciona la palabra conocida como **arquitectura**, desde el punto de vista de aplicaciones de software, ¿qué es una arquitectura?.

La arquitectura es la estructura organizativa de un sistema, que incluye su descomposición en partes, mecanismos de interacción y conectividad. La arquitectura también sirve de guía para proporcionar información sobre el diseño del sistema.

2.1 Clasificación de las arquitecturas.

Como se mencionó en el capitulo anterior, HIM es una aplicación que se ejecuta en entornos WEB, por ello, es necesario instalarla en un servidor para que de esta forma los clientes puedan acceder a ella desde un navegador WEB. Actualmente existen tres tipos de arquitectura que pueden ser utilizadas para estas aplicaciones:

- Arquitectura de dos capas.
- Arquitectura de tres capas.
- Arquitectura de n capas.

A continuación se describe brevemente estos tres tipos de arquitectura.

2.1.1 Arquitectura de dos capas.

La mayoría de las aplicaciones Cliente-Servidor funcionan bajo este tipo de arquitectura. Dicha arquitectura se divide en dos capas: el *front-end, y el back-end* (modelo). En la primera capa se sitúa la parte de la interfaz de usuario, llamadas a sentencias SQL y aplicaciones de escritorio. En la segunda capa se puede situar ya sea un servidor de base de datos ó algún otro tipo de almacenamiento como puede ser archivos XML.

El principal inconveniente que tiene esta arquitectura, se deriva del hecho de que, la interfaz de usuario, la lógica de aplicación y el modelo de datos se encuentran en el cliente. En la figura 2.1 se puede observar este hecho.

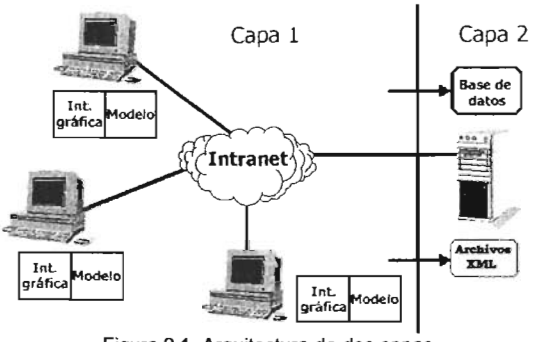


Figura 2.1. Arquitectura de dos capas.

Como se puede observar en la figura, si se hace un cambio en la implementación del modelo de datos se tendría que realizar una recompilación en cada uno de los clientes lo que realmente representa un problema. En resumen esta arquitectura presenta las siguientes características:

- Mucha carga en el cliente.
- Poca carga en el servidor.
- Mucho tráfico en la red.
- Mantenimiento costoso en cada cliente.

2.1.2 Arquitectura de tres capas.

La adición de una capa extra es una solución al problema que presenta el modelo anterior. La capa adicional se sitúa entre el *front-end y back-end.* Esta capa intermedia encapsula la lógica de la aplicación (o reglas de negocios) asociado con el sistema y la separa tanto de la interfaz de usuario como de la bases de datos. Ahora un cambio en la implementación del modelo solo afecta al servidor. La figura 2.2 muestra esta arquitectura.

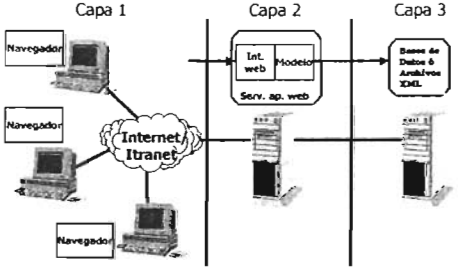


Figura 2.2. Arquitectura de tres capas.

Las principales característica de la arquitectura de 3 capas son:

- Se disminuye la carga en el cliente.
- La mayoría de la lógica del negocio se ejecuta del lado del servidor.

- Se pueden crear diferentes interfaces de usuario para la misma lógica del negocio.

2.1.3 Arquitectura de n capas (multicapa).

La arquitectura de n capas es muy similar a la arquitectura anterior. Sin embargo en esta arquitectura la lógica de la aplicación está dividida en componentes[14] según sus funciones, por lo que el número de capas de una aplicación dependerá de los componentes que se decida utilizar. Cabe señalar que la plataforma J2EE utiliza esta arquitectura. Debido a que HIM utiliza la plataforma J2EE para su desarrollo, se puede concluir que la arquitectura de HIM es una arquitectura multicapa. En la figura 2.3 se muestra un ejemplo de esta arquitectura.

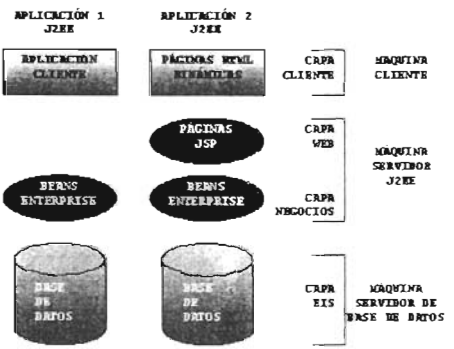


Figura 2.3. Arquitectura multicapa.

En la figura anterior, se observa que los componentes de la capa del cliente se ejecutan en la computadora del cliente (para el caso de HIM, en el navegador del cliente). Tanto los componentes de la capa WEB como los componentes de la capa del negocio, se ejecutan en el servidor J2EE. Los componentes de la capa EIS (*Enterprise Information Server*), que controlan el acceso a la base de datos se pueden ejecutar desde otro servidor ó en su caso desde el mismo servidor en donde se encuentra la capa WEB y la capa del negocio.

Las características que proporciona esta arquitectura son las siguientes:

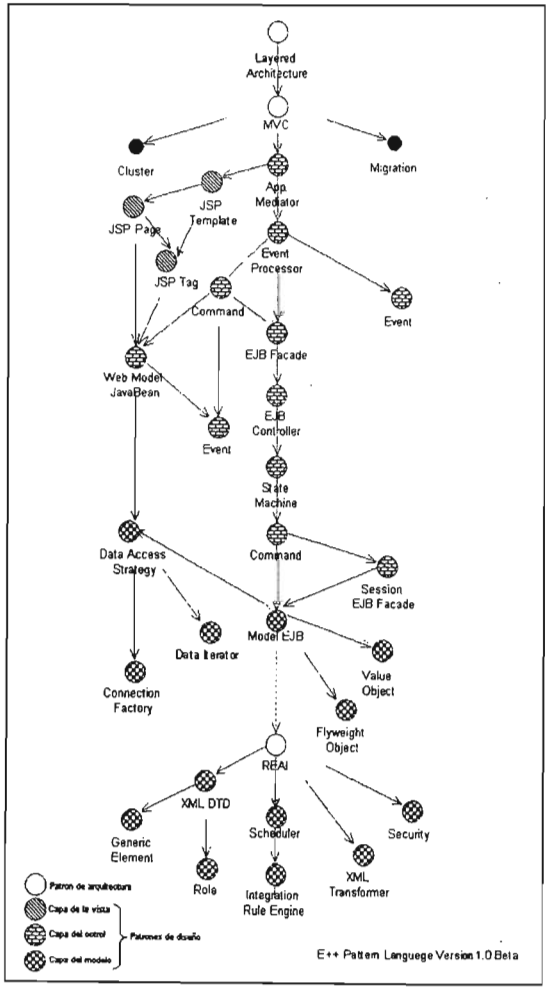
- Sólo los datos necesarios son transferidos al cliente (menor carga en la red).
- Al aumentar el número de niveles aumenta el número de comunicaciones entre los componentes, lo que acarrea un mayor tiempo de respuesta.
- Facilidad en el cambio de base de datos.
- Aislamiento frente a cambios.
- Seguridad.
- Administración central de recursos y localización de fallos.

[14] Es una unidad de software funcional auto-contenido, que se ensambla dentro de una aplicación y tiene la posibilidad de comunicarse con otros componentes.

2.2 Lenguaje de Patrones E++.

En el capitulo 1 se mencionó, que el lenguaje de patrones E++ no solo contiene patrones de diseño, también contiene **patrones de arquitectura, que proporcionan un marco de trabajo para los patrones de diseño**, es decir, proporcionan una forma en la cual los patrones de diseño puedan trabajar conjuntamente para construir un marco de trabajo J2EE de alta calidad. La forma en que se agrupan todos estos patrones, es mediante una estructura de árbol, en la cual cada nodo representa ya sea un patrón de arquitectura (nodo hueco) ó un patrón de diseño (nodo rellenado). En la figura 2.4, se muestra dicho árbol.

Figura 2.4. Árbol E++



De una manera sintetizada se puede visualizar, que el nodo raíz del árbol es el patrón de arquitectura conocido como **patrón de capas.** En dicho patrón, se plantea que todo sistema debe de ser particionado por funcionalidades para su correcto desarrollo. Éste patrón no indica el número de particiones que debe tener el sistema. Esta tarea la decide el siguiente nodo, que es el patrón conocido como **patrón MVC** ó también conocido como patrón Modelo-Vista-Controlador.
Éste patrón indica que toda aplicación debe de ser dividida para su correcto desarrollo en tres capas: la capa de la vista, la capa del modelo y **la capa del control.**

La capa de la vista (no cubierta en esta tesis) es la encargada de presentarle al usuario final la funcionalidad del sistema mediante interfaces gráficas. Para ello en el árbol se plantea el uso páginas JSP, plantillas JSP y etiquetas JSP.

La capa de control es la encargada del manejo, ejecución y coordinación de la lógica del negocio, es decir, los

requerimientos de la herramienta plasmados en código. Esta capa de acuerdo a los requerimientos de HIM, hace uso de algunos de los patrones del árbol, como son, el *Application Mediator* y el *Command*. Cada uno de estos patrones serán analizados más adelante.

La capa del modelo (no cubierta en esta tesis) es la encargada de almacenar el estado del sistema. Para ello hace uso del patrón de arquitectura conocido como REAI (*Rule Engine-Based Entrerprise Integration*). Típicamente el estado del sistema es almacenado en una Base de Datos ó en archivos XML.

El análisis del lenguaje E++ dio la pauta para poder seleccionar la arquitectura completa de HIM, para que posteriormente se realice un enfoque individual por capas basándose en el patrón MVC, correspondiente a la capa de control.

Existe además del lenguaje E++, otro marco de trabajo J2EE desarrollado por la comunidad de SUN, que ayuda de la misma forma al desarrollo de aplicaciones WEB.

2.3 Marco de Trabajo J2EE.

Actualmente existe un gran número de patrones que han sido publicados por la comunidad que desarrolla aplicaciones en J2EE. En el proceso de desarrollo e interacción de los patrones, los desarrolladores pueden crear un marco de trabajo que mejore la estabilidad, desempeño y escalabilidad de sus aplicaciones. Sin embargo, debido a que el número de patrones continúa creciendo, resulta difícil para los desarrolladores seleccionar la mejor combinación de patrones que les permitan crear un buen marco de trabajo. Para solucionar este problema, se han publicado algunos marcos de trabajo que se adecuan a las aplicaciones dependiendo de sus requerimientos.

La comunidad J2EE decidió también realizar, una clasificación de sus patrones. Tales patrones como anteriormente se mencionó son agrupados en 3 categorías (ver también la figura 1.3 del capítulo anterior): capa de presentación, capa del negocio y capa de datos.

La tabla 2.1 muestra estos patrones agrupados por categoría.

Capa de Presentación	Capa del Negocio	Capa de Datos
Intercepting Filter	***Business Delegate***	***Data Access Object***
Front Controller	***Business Object***	***Composite Entity***
View Helper	***Command***	
Validator	***Service Locator***	
Composite View	***Session Facade***	
	Transfer Object	
	Transfer Object Assembler	
	Value List Handler	

Tabla 2.1. Clasificaciones de patrones J2EE.

Los patrones de la capa de presentación son los encargados de interactuar con el cliente. Interceptan las peticiones del cliente, llevan a cabo el control de acceso y autenticación, dirigen y despachan las peticiones del cliente y construyen la respuesta al cliente, es decir, el menú de opciones que se visualizará en la interfaz de usuario.

Los patrones de la capa del negocio, se encargan de implementar y gestionar toda la lógica correspondiente a la aplicación, aquí es donde se utilizan de forma auxiliar los patrones de diseño GoF analizados en el capítulo 1.

Los patrones de la capa de datos se encargan de encapsular la comunicación con el modelo de datos (bases de datos ó archivos XML) e implementar los mejores mecanismos para la obtención de datos que serán visualizados en la interfaz del usuario.

A continuación se muestra en la figura 2.5 uno de los *Frameworks* J2EE propuestos por la comunidad de desarrolladores [IBM]. A través del análisis de diversas aplicaciones Web, la comunidad de desarrolladores diseñaron este "*Framework*" debido a que la gran mayoría de dichas aplicaciones involucraban el mismo proceso, es decir, necesitaban de la autenticación previa del usuario, el control centralizado de peticiones, la construcción de interfaces de usuario según su Rol y el acceso a bases de datos. Con éstos antecedentes la comunidad de desarrolladores recopilo los patrones que mejor se adecuarán a éstas características.

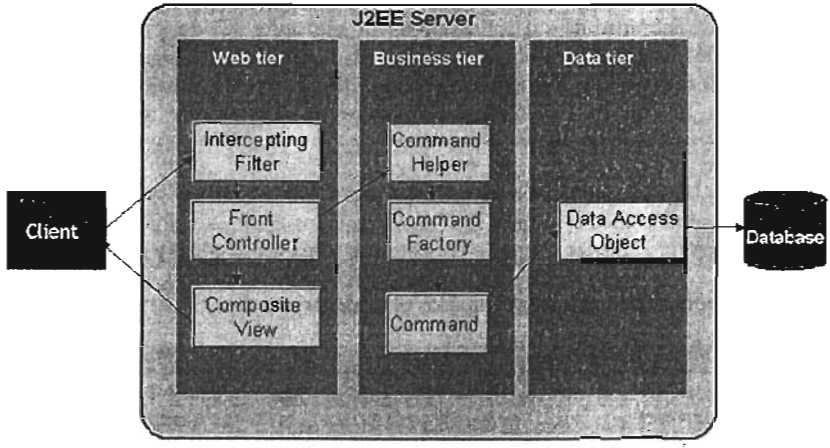


Figura 2.5. Marco de Trabajo J2EE.

HIM al ser una aplicación Web, requiere prácticamente de las mismas características que el *Framework* de la Figura anterior plantea, es decir, necesita que todos y cada uno de los usuarios que intenten ingresar a la herramienta sean previamente autenticados. Cada usuario debe de contar con uno ó más Roles los cuales son la pauta para poder construir las pantallas donde navegará el usuario. Se debe de contar con un control centralizado de

peticiones lo que ayuda a un mejor control de la funcionalidad de la herramienta y una mejor detección de errores en caso de que se presenten. A demás de tener un mecanismo flexible para poder obtener y visualizar los datos de forma simple al usuario.

Este marco de trabajo fue el que mejor se adecuó a los requerimientos de HIM. Por tal motivo de este modelo se partió para desarrollar la arquitectura propia de HIM (ver Figura 2.6) que es muy similar al "*Framework*" de IBM.

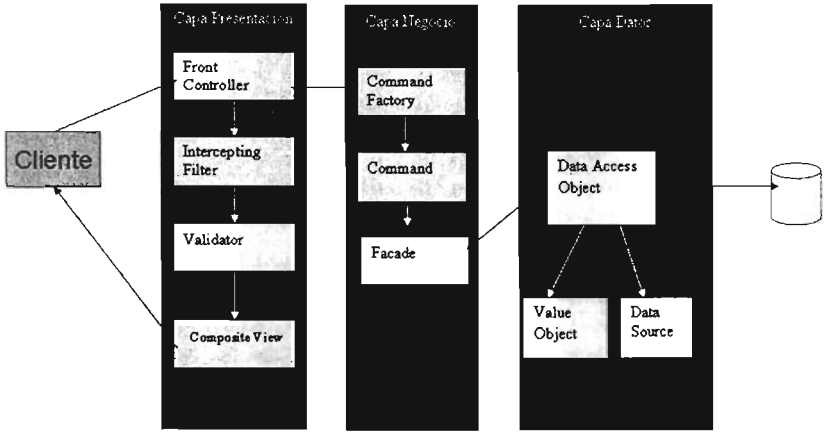


Figura 2.6. Arquitectura de HIM

De la capa de presentación:

- El patrón *Front Controller* es el encargado de procesar y dirigir todas las peticiones del usuario. Tal funcionalidad fue implementada por la capa de control (cubierta en esta tesis).

- El patrón *Intercepting Filter* es el encargado de llevar a cabo el control de acceso y autentificación. Dicha funcionalidad fue implementada por la capa de seguridad (no cubierta en esta tesis).

- El patrón *Validator* es el encargado de validar cada dato que proporcione el usuario en las interfaces. Tal funcionalidad fue implementada por la capa de control (cubierta en esta tesis).

- El patrón *Composite View* se encarga de construir las interfaces ó pantallas según el Rol del usuario. Tal funcionalidad fue implementada por la capa de vista (no cubierta en esta tesis).

De la capa del negocio:

- Los patrones *Command* y *Command Factory* ya fueron descritos en el capitulo 1. Tal funcionalidad fue implementada por la capa de control (cubierta en esta tesis).
- El patrón *Facade* es el encargado de proporcionar los métodos del negocio a nivel de datos ocultando la complejidad de su implementación. Tal funcionalidad fue implementada por la capa del modelo (no cubierta en esta tesis).

De la capa de datos:

- Los patrones *Data Access Object, Value Object y Data Sources* son los encargados de implementar la lógica necesaria para la creación, consulta y modificación de los productos de MoProsoft en el repositorio. Tal funcionalidad fue implementada por la capa del modelo (no cubierta en esta tesis).

De lo anterior, se concluye que la capa de control de HIM hace uso de algunos patrones tanto de la capa de presentación como de la capa del negocio. De la capa de presentación implementa el patrón conocido como *Front Controller* (en E++ llamado *Application Mediator*) y el patrón conocido como *Validator*. De la capa del negocio implementa el patrón conocido como *Command* y *Command Factory* mientras que hace uso de la funcionalidad proporcionada por el patrón *Facade*.

A continuación será descrito tanto el patrón *Front Controller* como el patrón *Validator*.

2.4 Patrones J2EE utilizados en la capa de control de HIM.

Para la descripción de cada uno de los patrones de diseño J2EE utilizados en la capa de control de HIM, la comunidad de desarrolladores decidió utilizar el formato más simple de la plantilla desarrollada por Alexander, agregándole una quinta área, la cual, es conocida como "contexto". Aquí se especifica los entornos bajo los cuales existe el patrón.

Front Controller.

- Contexto. El mecanismo de manejo de peticiones de la capa de presentación (las solicitudes que un cliente puede realizar desde un navegador) debe controlar y coordinar el procesamiento de todos los usuarios mediante las peticiones que lleguen a realizar. Dichos mecanismos de control se pueden manejar de una forma centralizada (ver Figura 2.7).

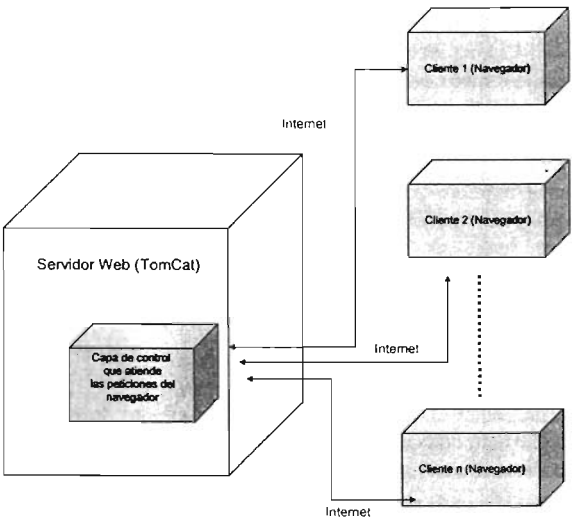


Figura 2.7. Mecanismo de peticiones.

- Problema. El sistema requiere un punto de acceso centralizado para que el manejo de peticiones de la capa de presentación soporte la integración de los servicios del sistema, recuperación de contenidos, control de vistas y navegación

- Causas. Dentro de las causas más importantes surgen:
  a. La lógica se maneja mejor en una localización central en lugar de estar replicada dentro de varias vistas.
  b. Existen puntos de decisión con respecto a la recuperación y manipulación de los datos.
  c. Se utilizan varias vistas para responder a peticiones de negocio similares.

- Solución. Usar un controlador como el punto de contacto para manejar las peticiones. El controlador maneja el control de peticiones, incluyendo la invocación de los servicios de seguridad como la autentificación, delegar el procesamiento del negocio, controlar la elección de una vista apropiada, el manejo de errores, y el control de la selección de estrategias de creación de contenido.

- Consecuencias. El acceso centralizado a una aplicación significa que las peticiones se pueden seguir y guardar muy fácilmente. Se debe de tener en mente que con un control centralizado es posible que se presente un solo punto de fallo del sistema.

Las tareas que controla el patrón *Front Controller* pueden ser implementadas por un marco de trabajo que forma parte del proyecto *Jakarta*, de la fundación *Apache Software* [DE2]. Este marco es conocido con el nombre de *Struts*, el cual se analizará ampliamente en el siguiente capitulo.

Validator.

- Contexto. El sistema recopila los datos proporcionados por el cliente (mediante una interfaz), dichos datos deben tener un formato válido. Éste formato depende del tipo de dato en cuestión, es decir, pueden ser datos de tipo fecha, hora, cadena, entero, compuestos (direcciones de correo), etc.

- Problema. Se requiere que los datos antes de ser procesados, sean validados. Si los datos tienen un formato incorrecto, esto conlleva a que se desperdicien recursos del sistema, es decir, el procesamiento que fue solicitado para cumplir algún requerimiento nunca se ejecuta ó en peor de los casos se ejecuta con datos que pueden dañar más adelante la integridad del sistema.

- Causas. Dentro de las más importantes se tienen:

    a. Se presentan ocasiones en las cuales el cliente no conoce con exactitud el tipo de dato que debe de proporcionar.
    b. Se desea evitar que los procesamientos que llevan a cabo la lógica del negocio no sean interrumpidos por errores en los datos que éstos procesan.
    c. Se desea tener un lugar centralizado para el control de cada uno de los errores que se puedan presentar por formatos incorrectos en los datos.

- Solución. Una vez que se cuenta con el controlador central proporcionado por el patrón anterior (*Front Controller*), se necesita que dicho controlador dirija la petición del cliente a un proceso auxiliar en el cuál se verifique la integridad de los datos. Si los datos son correctos la petición es atendida. Si los datos tienen un formato incorrecto se debe de contar con un mecanismo que se lo haga saber al cliente redirigiendo de nuevo la petición. La complejidad en la implementación del proceso validador dependerá exclusivamente del tipo de dato que se espera proporcione el cliente.

- Consecuencias.

    a. Validar el formato de los datos antes de procesarlos, ayuda al buen funcionamiento y a la consistencia del sistema.
    b. Centralizar el control de errores ocasionados por el formato de los datos, facilita la depuración de los mismos.

Una vez que se definió la arquitectura de HIM, y los patrones de diseño que se van a utilizar en la capa de control, se necesita profundizar un poco más en la forma en que funciona el patrón *Front Controller,* ya que se puede considerar como una de las partes centrales de la herramienta HIM. Por este motivo, el marco de trabajo conocido como *Struts* será motivo de análisis en el siguiente capítulo.

CAPÍTULO 3. Struts.

Este capitulo describe el funcionamiento del marco de trabajo conocido como Struts, es decir, se analiza el funcionamiento e interacción de los componentes que lo conforman.

En el capítulo anterior se comentó que el patrón conocido como *Front Controller* es el encargado de gestionar todas y cada una de las peticiones que pueda realizar un usuario al sistema. Se comentó además que existe un marco de trabajo que puede implementar dicho patrón y se conoce como *Struts*.

Struts además de realizar labores de control de peticiones, cuenta con una amplia gama de funcionalidades, de hecho, fue creado para implementar el patrón de arquitectura conocido como MVC (Modelo-Vista-Control).

3.1 Introducción a *Struts*.

Los componentes que forman una aplicación Web pueden ser clasificados dentro de dos grupos: los componentes del negocio y los servicios de aplicaciones. En la mayoría de los casos resulta complicado poder reutilizar los componentes del negocio en otras aplicaciones dado que los requerimientos cambian. Sin embargo los servicios se pueden volver a utilizar en otras aplicaciones. Tales servicios pueden incluir el enrutamiento de solicitudes, resolución de errores, control de vistas y diseño de etiquetas personalizadas.

Cualquier programa que tenga la capacidad de ofrecer este tipo de servicios y que se pueda utilizar en el diseño de una aplicación incorporando sus componentes del negocio, se puede denominar como **marco** ó **estructura de aplicación Web.**

Struts es una estructura de aplicación Web de código abierto, basada en el patrón de diseño MVC y diseñada utilizando el servlet y las API's de Java para construir complejas aplicaciones Web [FGI+01].

Struts permite dividir la lógica del negocio, la lógica de control y el código de presentación de una aplicación, lo que mejora su posterior utilización y mantenimiento. El marco de trabajo *Struts* forma parte del proyecto *Jakarta*, gestionado por la fundación Apache Software.

El marco de trabajo *Struts* ofrece los siguientes servicios, de acuerdo, a los requerimientos de una aplicación Web:

- Un servlet que actúa como controlador central.
- Bibliotecas de etiquetas JSP (se explicarán más adelante) para la administración de *JavaBeans*[15], generación de HTML, manejo de plantillas y control de flujo en JSP.
- Una estructura de internacionalización de mensajes, lo que significa que cualquier mensaje que aparezca al utilizar la aplicación se encuentra en el idioma del usuario. Para ello es necesario crear un archivo de recursos de aplicación en el que se incluyan los mensajes adecuados para cada idioma.
- Implementación de JDBC para definir las fuentes de datos y una agrupación de conexiones a bases de datos.
- Un mecanismo general de resolución de errores y excepciones, lo que implica la recuperación de mensajes de error desde un archivo de recursos de aplicación.
- Análisis de sintaxis (parseo) XML.
- Utilerías para cargar archivos.
- Utilerías de conexión.

En el transcurso de este capítulo se explicará de una manera simple, como se pueden utilizar algunas de estas características.

3.2 Arquitectura de *Struts*.

A continuación se muestra la arquitectura de *Struts*, en la cual, se utiliza el paradigma MVC para dividir sus componentes. Ver figura 3.1.

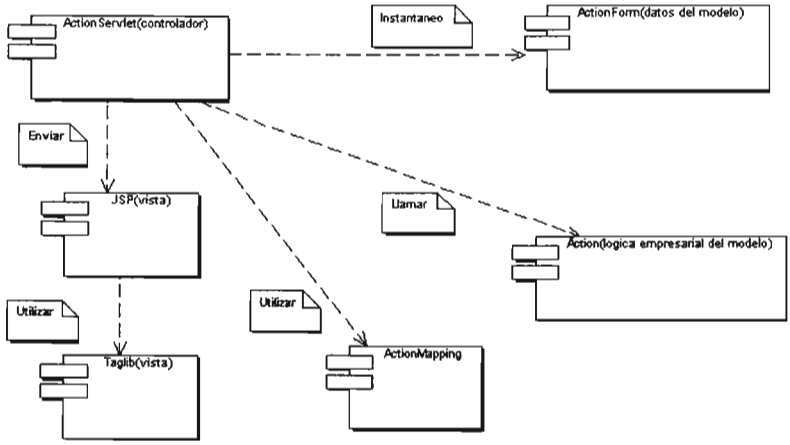


Figura 3.1. Arquitectura de Struts.

[15] clases donde se almacenan los datos que proporciona un usuario desde la interfaz

Para desarrollar el nivel de presentación (vista) de una aplicación basada en *Struts*, se utiliza las bibliotecas de etiquetas de *Struts* (*taglib*). Todas las solicitudes del cliente se transmiten a un servlet denominado ***ActionServlet*** que actúa como controlador central (éste controlador es la implementación del patrón *Front Controller*). ActionServlet a su vez, pasa los datos de la solicitud a un *JavaBean* conocido como *ActionForm*.

Un ***ActionForm*** es un *JavaBean* que representa los datos que son recolectados desde un formulario (datos que envía el usuario desde una interfaz). Estos formularios son generados por las páginas JSP con ayuda de la biblioteca de etiquetas *html* (se analizan mas adelante) de *Struts*. El *ActionForm* tiene la posibilidad (de manera opcional) de validar los datos (implementación del patrón *Validator*) antes de pasarlos al *ActionServlet* para su posterior procesamiento.

Un ***ActionMapping*** es un objeto que permite redireccionar la solicitud del cliente, es decir, si la solicitud es exitosa, la petición se continua procesando hasta ejecutar cierta funcionalidad, si la solicitud es errónea la petición se procesa de nuevo solicitando una vez más los datos.

Las clases ***Action*** son los componentes específicos de la aplicación utilizados para ejecutar todas las solicitudes del cliente.

3.3. Configuración de *Struts*.

Como ya se mencionó, *ActionServlet* es el principal componente del elemento controlador, es el responsable de la delegación de las solicitudes. Para lograr esto, se necesita un tipo de guía que dirija las solicitudes a los componentes correspondientes, es decir, que comprenda desde la asignación de una determinada solicitud (basada en su dirección *URL*), hasta el componente que procesará dicha solicitud. Toda esta información acerca de la asignación se guarda en objetos *ActionMapping*. Estos objetos se configuran mediante un archivo XML, los cuales son creados por el programador de la aplicación.

Este archivo XML comúnmente conocido como *struts-config.xml*, contiene la siguiente información:

- Se definen los elementos *JavaBeans* que almacenarán los datos recogidos desde la interfaz. Cada elemento que representa un *JavaBean* contiene:

    a) Un nombre.
    b) La ruta de la clase que almacenará los datos.

- Se definen los elementos *ActionMapping* que sirven para configurar los objetos *ActionMapping*. Cada elemento *ActionMapping* contiene:

    1. El nombre del servlet que atenderá la petición.

2. La ruta de la clase que implementa al servlet. Esta clase es una clase *Action* que implementa la lógica de la aplicación.
3. El nombre del *JavaBean* que utilizará la clase *Action* para procesar los datos que proporcione un usuario.
4. El alcance del *JavaBean*, es decir, si el valor del JavaBean permanece vigente por toda la sesión del usuario.
5. La ruta del JSP en el caso que se presente un error en la información que proporciona el usuario desde la interfaz.
6. El valor de validación del *JavaBean*. Esto se aplica para cuando se quiere validar la información del *JavaBean* antes de que sea procesada por la clase *Action*.
7. La ruta del recurso al que se dirigirá la respuesta, en el caso de que el procesamiento haya sido exitoso.

- Se definen fuentes de datos JDBC para el acceso a bases de datos. Aquí se define el nombre del controlador de dicha fuente de datos, la ruta de la fuente de datos, el nombre y contraseña de acceso a la fuente de datos y el número de conexiones por si la concurrencia de usuarios conectados al sistema aumenta.

A continuación se muestra un fragmento de código para su mejor comprensión. En él se muestra el inciso al cual hace referencia en los puntos anteriores.

```
<struts-config>
 <form-beans>
  <form-bean
       name="ingresarSistemaActionForm"                    .....a)
       type="moprosoft.him.ingresarSistemaActionForm" />  .....b)
 </form-beans>
 <action-mappings>
  <action
       path="/ingresarSistemaAction"                       .....1)
       type="moprosoft.him.ingresarSistemaAction"  .....2)
       name="ingresarSistemaActionForm"            .....3)
       scope="request"                  .....4)
       input = "/ingresarSistema.jsp"  .....5)
       validate = "true"                .....6)
        <forward name = "accesoExitoso" path="/MenuPrincipal.jsp"/>  .....7)
  </action>
 </action-mappings>
 <data-sources>
  <data-source key="dataSource">
   <set-property property="driverClassName" />
   <set-property property="url" />
   <set-property property="username" />
   <set-property property="password" />
  </data-source>
 </data-sources>
```

3.4. Componentes de *Struts*.

Para comprender de una manera más detallada el funcionamiento de *Struts*, a continuación se analiza de forma resumida cada uno de sus componentes.

3.4.1 El controlador *ActionServlet*.

Este componente es el más importante dentro de *Struts* y se implementa mediante la clase *org.apache.struts.action.ActionServlet*. Cuando se presenta una solicitud, el controlador realiza las siguientes operaciones:

- Localiza el identificador URI (identificador universal/uniforme de recursos) de la solicitud entrante.
- Hace coincidir el identificador URI con el objeto *ActionMapping* correspondiente.
- Crea una instancia de bean *ActionForm*, en caso de que se haya declarado uno. Recupera también las propiedades de alcance del bean.
- Invoca al método *perform*() sobre la instancia de la clase *Action*. En este método es donde se implementa la lógica del negocio.
- Redirige la respuesta al usuario.

3.4.2 La clase *Action*.

Para que el método *perform*() de la clase *Action* pueda implementar la lógica del negocio, necesita tener acceso a cuatro objetos que son proporcionados como parámetros. El primer parámetro es el objeto *ActionMapping* que proporciona información del recurso al cual se debe dirigir la solicitud en caso de éxito ó fracaso. El segundo parámetro es el objeto *ActionForm* que contiene los datos que son recolectados desde la interfaz. El tercer y cuarto parámetro son los objetos *ServletRequest* y *ServletResponse* que proporcionan un mejor control de la información que se puede enviar en una petición.

La clase *Action* genera una respuesta a la solicitud del usuario y posteriormente la dirige hasta un recurso determinado por medio de la clase *ActionForward* que se define en el archivo de configuración. Cuando el método *perform*() es ejecutado, retorna un objeto de tipo *ActionForward* que contiene el recurso al cual se dirige el siguiente procesamiento. Este recurso puede ser otro servlet ó un JSP donde se pueden proporcionar más datos.

En la clase *Action* existe otro método conocido como *saveErrors*() que se utiliza para guardar mensajes de error. Estos mensajes una vez almacenados, pueden ser mostrados en las paginas JSP mediante las etiquetas HTML de *Struts*.

3.4.3 El *bean ActionForm.*

Cuando *ActionServlet* invoca la clase *Action*, crea la correspondiente instancia del *bean ActionForm* y la pasa a la clase *Action* como parámetro en el método *perform*(). La clase *ActionForm* cuenta con un método conocido como *validate*() cuya funcionalidad es validar los datos antes de que el *bean ActionForm* sea pasado a la clase *Action*. En este método es donde se puede implementar el patrón *Validator* y su complejidad dependerá del tipo de dato que se haya almacenado en el bean.

3.4.5 El objeto *ActionMapping.*

Como ya se comento, el *ActionServlet* pasa una instancia de la clase *ActionMapping* como parámetro del método *perform*(). Este objeto contiene toda la información definida en el archivo de configuración XML. Con esta información se puede recuperar el nombre del recurso al cual se dirige la petición por medio del método *findForward*() del objeto *ActionMapping*. Este método retorna un objeto del tipo *ActionForward* que a su vez es utilizado como valor de retorno en el método *perform*() de la clase *Action.*

A grandes rasgos esta es la funcionalidad de los 4 componentes de *Struts*. Cuando dichos componentes interaccionan en conjunto, se puede procesar en su totalidad una petición de usuario. La figura 3.2 muestra en un diagrama de colaboración un caso práctico en donde se visualiza la interacción de estos 4 componentes.

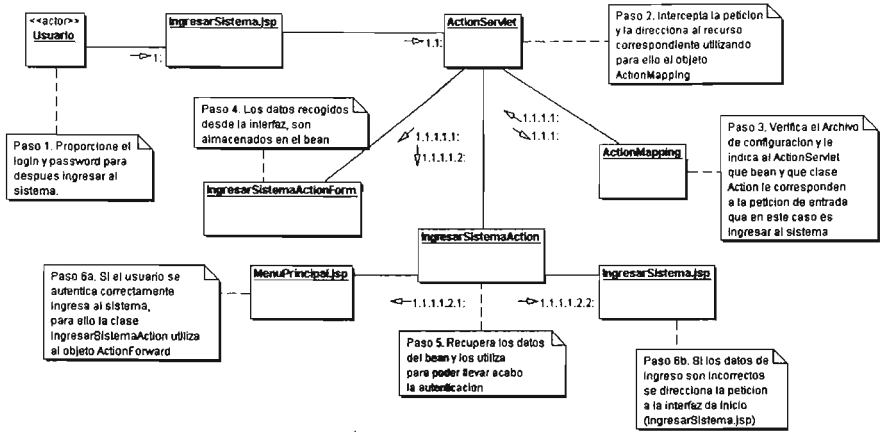


Figura 3.2. Diagrama de colaboración para el caso de uso Ingresar al Sistema de HIM.

3.5 La biblioteca de etiquetas de *Struts*.

El objetivo de la biblioteca de etiquetas de *Struts* es poder manipular información en los componentes JSP (páginas). Esta biblioteca esta constituida por cuatro series de etiquetas que son:

- *bean*. Su objetivo es manipular JavaBeans dentro del JSP
- *logic*. Permite manipular el flujo de control dentro del JSP.
- *html*. Genera contenido html dentro del JSP.
- *template*. Permite construir páginas que utilizan formato común, utilizando para ello plantillas dinámicas.

Aunado a esta biblioteca, el desarrollador de la aplicación puede crear sus propias etiquetas de acuerdo a los requerimientos que se vayan presentando en el transcurso de la aplicación. Sin profundizar a detalle, estas etiquetas personalizadas requieren de una clase que sea capaz de manipular los datos que se desean exponer en el JSP y de un archivo de configuración que proporcione la información necesaria para usar la etiqueta.

3.5.1 Etiquetas *bean*.

El marco de trabajo de Struts proporciona una amplia variedad de etiquetas para trabajar con JavaBeans desde páginas JSP. Estas etiquetas se pueden dividir por funcionalidad en cuatro grupos:

- Etiquetas para copiar *JavaBeans*, es decir, se puede copiar la información de un *JavaBean* a otro *JavaBean*.
- Etiquetas para definir variables de *script*. Se crean variables que se puedan utilizar dentro de los JSP.
- Etiquetas para la reconstrucción de *beans*, es decir, se puede mostrar la información que almacena un *JavaBean* en la interfaz que visualiza el usuario.
- Etiquetas para la internacionalización de mensajes. Los mensajes que se muestren en la interfaz de usuario aparecerán en el idioma del usuario.

3.5.2 Etiquetas lógicas.

Estas etiquetas tienen la capacidad de manejar el control de flujo dentro de una página JSP, por ejemplo, la iteración o la evaluación del cuerpo de la etiqueta. Según su funcionalidad estas etiquetas se categorizar en tres grupos:

- Etiquetas para la lógica condicional. Una analogía de este tipo de etiquetas son las estructuras de control de Java "if" y "else" junto con sus operadores de comparación (igual que, menor que, mayor que, etc.). Dichas etiquetas tienen la capacidad de poder seleccionar el camino de la ejecución de código dependiendo de la condición.

- Etiquetas de iteración. Estas etiquetas son similares en funcionalidad a los ciclos "for" y "while" de Java. Se itera sobre los datos que almacena la propia etiqueta.
- Etiquetas de reenvío de respuestas. Simplemente sirven para poder reenviar la petición a otro recurso, puede ser otro JSP, un servlet ó en su caso una dirección URL.

3.5.3 Etiquetas *html*.

Estas etiquetas son las más comunes y utilizadas, debido a que tiene la capacidad de poder generar elementos *html* que sirven para la captura de datos proporcionados por el usuario, estos elementos *html* pueden ser: campos de texto, áreas de texto, listas de selección, casillas de verificación, etc.

También estas etiquetas son utilizadas para mostrar en la interfaz del usuario los mensajes de error que se puedan generar ya sea en la misma captura de los datos por parte del usuario ó en su caso por algún error que se presente por parte del funcionamiento de la aplicación.

3.5.4 Etiquetas *template*.

Siempre se debe de tomar en cuenta que una aplicación Web puede estar formada por cientos de páginas que utilizan el mismo diseño, es decir, cada página cuenta con una sección de encabezado, una sección de contenido (en constante variación) y una sección de pie de página. El uso de plantillas es la mejor solución para tal diseño, por el motivo de que si llegará a cambiar el contenido de alguna de las tres partes no se tendría que realizar dicha modificación en cada una de las páginas. La modificación sólo se realizaría en la plantilla y automáticamente se actualizarían todas las páginas con la nueva información.

Las etiquetas *template* permiten poder hacer uso de plantillas dinámicas que son una de las formas más completas de modular el diseño de una página Web.

Una vez que se cuenta con la arquitectura de la herramienta HIM y además de que ya se explicó el funcionamiento básico de *Struts*, se procederá a explicar como se realizó el análisis y diseño de la capa de control de HIM.

CAPÍTULO 4. Análisis y diseño de la capa de control de HIM.

Este capítulo presenta la manera en que se realizó el análisis y diseño de la capa de control de HIM, tomando como base sus requerimientos.

4.1 Introducción.

En el capítulo dos se mostraron los motivos por los cuales en HIM se utilizó una arquitectura que dividía el sistema en tres capas: modelo, vista y control. Se plasmó también la arquitectura de diseño que se utilizó en HIM, usando para ello el marco de trabajo propuesto por la comunidad J2EE para aplicaciones en Web.

Para el análisis y diseño de la capa de control, se tomó en cuenta la forma en que son organizados los componentes de *Struts* (ver capitulo 3.4), esto con el fin de poder relacionar los requerimientos de HIM con algunos de los componentes de *Struts* y de esta forma poder agruparlos.

A continuación a lo largo de este capítulo se muestra la forma en que se definió la arquitectura de la capa de control de HIM.

4.2 Descripción del diagrama general de casos de uso de HIM.

El diagrama general de casos de uso de HIM y su breve descripción, será la pauta para poder entender la forma en que se realizó el análisis y diseño de la capa de control de HIM.

Cuando se realizó el análisis de requerimientos de HIM se concluyó que la herramienta podía gestionar dos tipos de productos generados por MoProSoft.

Estos productos se pueden dividir en dos categorías:

- Los productos que para ser creados, consultados ó modificados, necesitan de la captura de información desde la interfaz de usuario, como por ejemplo, los datos que se proporcionan para crear un nuevo registro de recursos humanos (nombre, dirección, teléfono, etc.). A este tipo de productos se les dio el nombre de productos con información especializada (**PIE**).

- Los productos que por su contenido son almacenados ó consultados como archivos, como por ejemplo, el plan del proyecto, en donde se lleva a cabo un calendario de actividades que pudiera ser realizado en una herramienta como Microsoft Project. A este tipo de productos se les dio el nombre de **cajas negras**.

4.3 Estructura de paquetes de la capa de control de HIM.

Con los antecedentes previos, se llegó a organizar la capa de control con la siguiente estructura de paquetes (ver Figura 4.2).

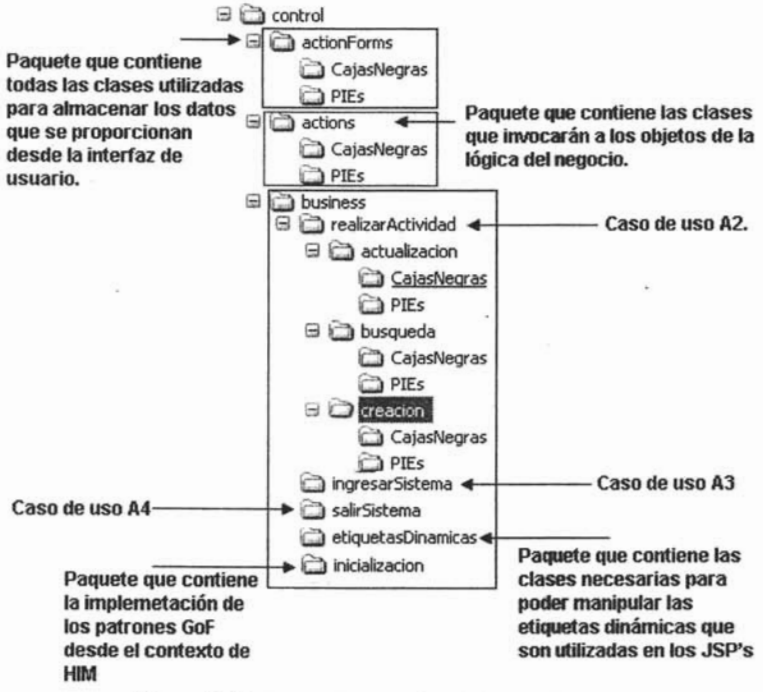


Figura 4.2. Estructura de paquetes de la capa de control

De la Figura 4.2, se puede apreciar que la capa de control está dividida en tres paquetes:

- actionForms. Este paquete contiene todas las clases utilizadas para almacenar los datos que se proporcionan desde la interfaz de usuario (ver capítulo 3.4.3).

- actions. Este paquete contiene las clases que invocarán a los objetos que implementan la lógica del negocio (ver capítulo 3.4.2).

- business. Este paquete contiene cinco subpaquetes. Cada subpaquete dependiendo de su funcionalidad contiene todas las clases encargadas de implementar la lógica del negocio y son:

  - ✓ realizarActividad. Este paquete contiene las clases que llevarán a cabo las operaciones fundamentales sobre cada producto, es decir, la creación, consulta y modificación de un cierto producto.

  - ✓ ingresarSistema. Este paquete contiene todas las clases necesarias para poder cumplir con el proceso de ingreso al sistema.

  - ✓ salirSistema. Este paquete contiene todas las clases necesarias para poder cumplir con el proceso de salida del sistema.

  - ✓ etiquetasDinamicas. Este paquete contiene las clases necesarias para poder manipular las etiquetas dinámicas que son utilizadas en los JSP's.

  - ✓ inicializacion. Este paquete contiene las interfases necesarias para la creación de la fábrica de comandos, es decir, la forma en que se utilizaron los patrones *Abstract Factory, Factory Method y Command* aplicados a HIM.

También se puede apreciar que en cada uno de los paquetes se realizó una división que separa los PIE´s de las cajas negras y así tener una mejor agrupación.

En lo que respecta al paquete de "inicializacion", es importante señalar la forma en que los patrones de diseño fueron implementados desde el punto de vista de la herramienta, es decir, la forma en que éstos fueron mapeados al contexto de HIM. Para lograr esta correspondencia, la capa de control buscó tener centralizada la gestión de cada uno de los objetos utilizados en la lógica del negocio, para ello, se utilizó la fábrica abstracta de comandos para que cumpliera este objetivo. En ella cada producto de HIM tiene la facultad de poder ser creado, consultado ó modificado (ver la Figura 1.2 del capítulo 1). Cuando se quiera crear, consultar ó modificar un producto de HIM, basta con hacer referencia a la fábrica abstracta de comandos y utilizar el comando específico, dicho comando se ejecuta y la funcionalidad para el producto en cuestión se lleva a cabo. La estructura organizativa de la fábrica de comandos es aplicable tanto en los productos PIE´s como en las cajas negras, es decir, tanto las cajas negras como los PIE´s pueden ser creados, consultados y modificados. La única variación se presenta en la implementación a nivel de código del método "ejecutar" de cada comando. Esto es debido a que como ya se comento, los PIE´s contienen información que se crea ó consulta directo en la interfaz, mientras que las cajas negras son archivos que se crean, consultan ó modifican de acuerdo a su extensión.

Con esta información ya se puede pasar a la parte del análisis de la capa de control de HIM.

4.4 Análisis de la capa de control de HIM.

Los diagramas de clases del análisis intentan refinar con mayor detalle la funcionalidad de cada caso de uso de HIM, es decir, establecer la asignación inicial de funcionalidad del sistema a un conjunto de clases desde un punto de vista abstracto.

La Figura 4.4 muestra el diagrama de clases del análisis para el caso de uso **"A3.Ingresar al sistema"** (ver Figura 1).

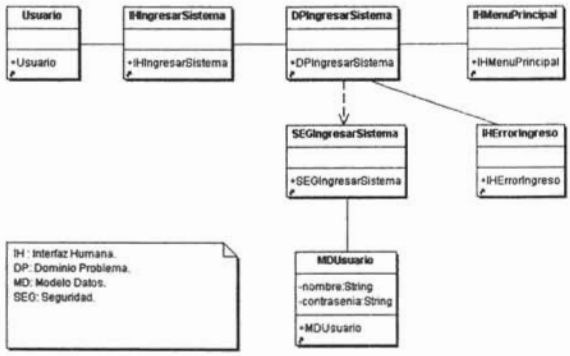


Figura 4.4. Diagrama de clases del análisis para ingresar al sistema.

En el diagrama se aprecia que las clases tienen un prefijo, el cual, plasma la funcionalidad de cada clase, es decir:

- El prefijo "IH" se refiere a todas las interfaces con las que interactuará el usuario (más adelante en los diagramas de clases del diseño se plasman como páginas JSP).

- El prefijo "DP" indica que dicha clase (DPIngresarSistema) se encargará de procesar toda la lógica del negocio correspondiente al ingreso al sistema (a un mayor detalle se implementa la funcionalidad de la clase con el uso de "Struts" y con los patrones de diseño GOF).

- El prefijo "MD" indica que dicha clase (MDUsuario) lleva a cabo la labor de recuperar los datos relacionados con el acceso al sistema (persistencia de los datos).

- El prefijo "SEG" se refiere a la clase (SEGIngresarSistema) que proporciona la seguridad necesaria en el proceso de ingreso al sistema.

El diagrama de clases del análisis para el caso de uso **"A2.Realizar actividad"** se muestra en la Figura 4.5.

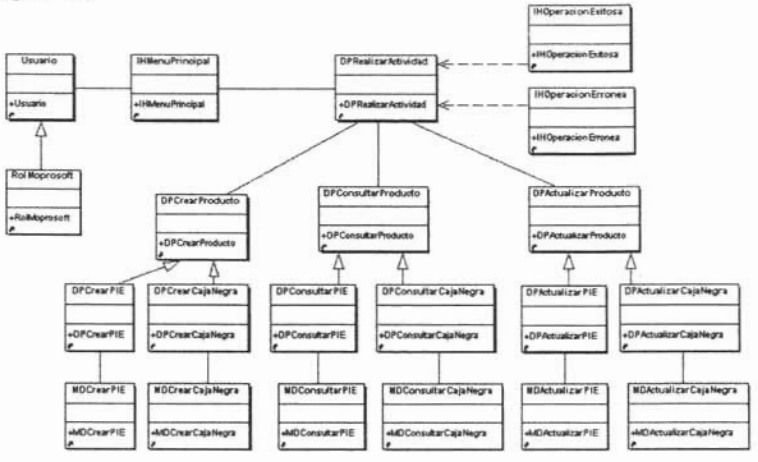


Figura 4.5. Diagrama de clases del análisis para realizar actividad.

En el diagrama se plasma la forma en que se puede llevar a cabo una actividad de MoProSoft. Cuando el usuario accede al sistema con un rol especifico, la interfaz "IHMenuPrincipal" le presenta todas y cada una de las actividades que puede ejecutar dicho usuario según su rol. Cada actividad por su parte puede crear, consultar o modificar un producto, que como ya se mencionó anteriormente puede ser un PIE o una caja negra.

En la Figura 4.6 se muestra el diagrama de clases del análisis para el caso de uso **"A4.Salir del sistema"**.

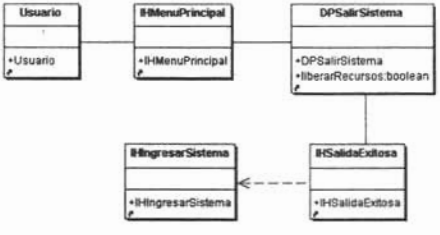


Figura 4.6. Diagrama de clases del análisis para salir del sistema.

El diagrama anterior tiene la funcionalidad de que cuando el usuario termina su sesión se debe de liberar todos los recursos que se hayan utilizado, posteriormente se debe de direccionar una vez más a la interfaz de entrada.

4.5 Diseño de la capa de control de HIM.

Los diagramas de clases del diseño que se presentarán a continuación toman en consideración detalles de implementación tales como el lenguaje en que se codificarán las clases, la tecnología utilizada para poder cumplir la funcionalidad de cada caso de uso (*Struts*) y las buenas prácticas de programación que ayudan también al cumplimiento de los casos de uso (patrones de diseño). Por su parte, los diagramas de secuencia son un complemento a los diagramas de clases del diseño ayudando a plasmar de una forma ordenada los pasos que se deben de seguir para cumplir con la funcionalidad de cada caso de uso.

4.5.1 Diagrama de clases y secuencia para ingresar al sistema.

En la Figura 4.7 se muestra el diagrama de clases del diseño para el caso de uso **"A3.Ingresar al sistema".** En este diagrama se puede apreciar que existe una correspondencia con el diagrama de clases del análisis de la Figura 4.4.

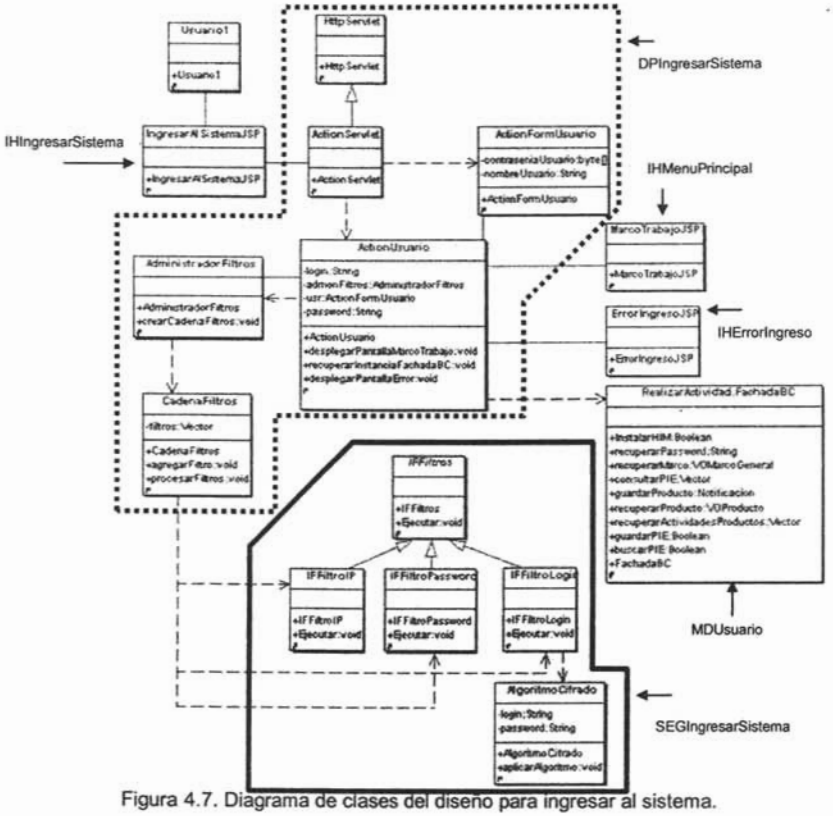


Figura 4.7. Diagrama de clases del diseño para ingresar al sistema.

En el diagrama anterior se aprecia que la clase del análisis con nombre "IHIngresarSistema" se mapea a su correspondiente clase del diseño como una página JSP conocida como "IngresarSistema.jsp". Se aprecia también que la clase "DPIngresarSistema" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para ingresar al sistema. Este conjunto esta conformado por las clases que implementan la funcionalidad del marco de trabajo "*Struts*"(ver capítulo 3.2) que son "ActionServlet", "ActionFormUsuario" y "ActionUsuario" y de dos clases que gestionan el proceso de filtrado "AdministradorFiltros" y "CadenaFiltros", esta última clase, instancia las 3 clases que llevan a cabo la tarea de proporcionar la seguridad en el ingreso a HIM (verificar y validar los datos del usuario) y son: "IFFiltroIP", "IFFiltroLogin" y "IFFiltroPassword" (agrupadas con una línea continua). La función de estas 3 clases es filtrar datos (desde el punto de vista abstracto), por lo tanto, se requiere que dichas clases tengan la libertad de definir su tipo de Filtrado en tiempo de ejecución (ver patrón *Factory Method*, capítulo 1.4.2) implementando para ello la interfase "IFFiltros". La clase "AlgoritmoCifrado" por su parte, lleva a cabo la función de descifrar y cifrar los datos (por motivos de seguridad), para ello, necesita recuperar dichos datos del objeto de la clase "FachadaBC" que contiene el método que lleva a cabo la funcionalidad de la clase "MDUsuario" perteneciente al diagrama de clases del análisis de la Figura 4.4. Por último, tanto la clase "IHMenuPrincipal" como la clase "IHErrorIngreso" se mapean a su correspondiente página JSP ("MenuPrincipal.jsp" y "ErrorIngreso.jsp" respectivamente).

A continuación en la Figura 4.8 se muestra el diagrama de secuencia para el caso de uso **"A3.Ingresar al sistema"**.

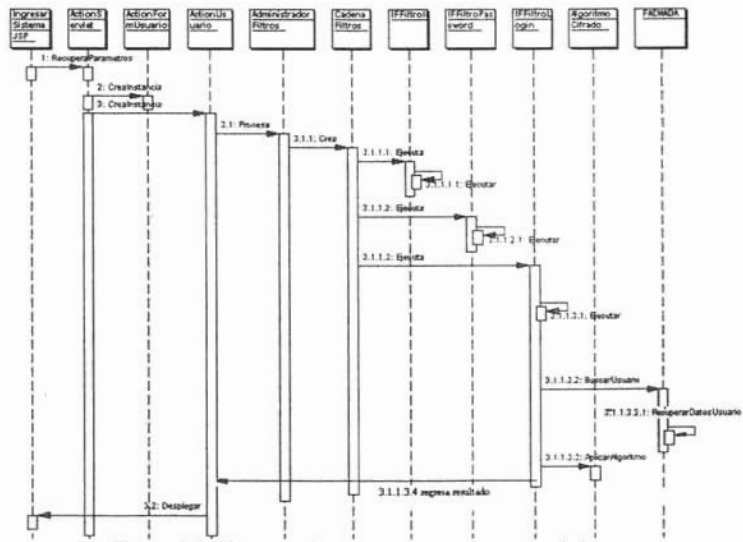


Figura 4.8. Diagrama de secuencia para ingresar al sistema.

En el diagrama anterior cuando el usuario accede al sistema realiza una petición al "ActionServlet" (patrón *Front Controller* implementado por Struts), dicho servlet crea un objeto de la clase "ActionFormUsuario" y almacena allí los datos que recoge de la interfaz. Estos datos son pasados al objeto de la clase "ActionUsuario" para que sean procesados. "ActionUsuario" crea un objeto de la clase "AdministradorFiltros" el cual controlará todo el proceso de filtrado. Este proceso consiste en crear una cadena de filtros en donde se verifica la dirección IP del usuario, su *login* y su *password*. Para recuperar el *login* y el *password*, el objeto de la clase "IFFiltroLogin" y el objeto de la clase "IFFiltroPassword" hacen uso de la fachada, la cual obtiene esos dos datos almacenados en un archivo en XML. Con los datos recuperados, se aplica el algoritmo de cifrado, si la cadena de filtros se ejecuta completamente, el objeto de la clase "ActionUsuario" dirige la petición del usuario al menú principal de la herramienta, en caso contrario se presenta un mensaje al usuario indicando la negativa de su acceso a HIM.

Dentro de las ventajas que proporciona este diseño se presentan las siguientes:

- Uso de interfases que permiten extender (si así se requiere) la funcionalidad del sistema, con esto, no se necesita realizar mayores cambios al código ya existente.
- Uso de filtros que permiten ahorrar tiempo de ejecución en caso de que la funcionalidad de alguno de ellos no se llegue a concretar.
- Centralización de la creación de comandos lo que conlleva a un mejor control.
- Centralización de todas y cada una de las peticiones (a través de Internet) que entran al sistema.
- Se obtiene, una arquitectura estable lo que proporciona una mejor gestión en lo que respecta a la funcionalidad de la herramienta.

Las desventajas de utilizar este diseño son las siguientes:

- Un incremento en el tiempo de ejecución, debido a que se necesita crear un mayor número de instancias de todos aquellos objetos encargados de llevar a cabo alguna de las funcionalidades que el sistema proporciona.
- Mayor tiempo de desarrollo en lo que respecta al flujo de trabajo de diseño, por el hecho de que todos los integrantes del equipo de trabajo que implementarán los casos de uso requieran de mayor capacitación en lo que respecta a "*marcos de trabajo*", patrones de arquitectura y patrones de diseño.

Como la creación, consulta y modificación de un producto PIE utiliza el mismo procedimiento para llevar a cabo dichas operaciones, las ventajas y desventajas que se presentan para el diseño del caso de uso **A3.Ingresar al sistema** son aplicables a los casos de uso restantes.

4.5.2 Diagrama de clases y secuencia para realizar actividad.

El caso de uso **"A2.Realizar actividad"** contempla las operaciones de creación, consulta y actualización de cualquier producto PIE que pueda ser generado por una actividad de MoProSoft. De esta manera a continuación se presentarán los diagramas de clases de diseño y de secuencia para estas tres operaciones para un producto PIE. Más adelante se realizará el mismo análisis para las cajas negras.

4.5.2.1 Diagrama de clases y secuencia para creación de un PIE.

En la Figura 4.9 se muestra el diagrama de clases del diseño para el caso de uso "creación PIE". En este diagrama se puede apreciar que existe una correspondencia con el diagrama de clases del análisis de la Figura 4.5 en lo que respecta a la creación de un producto PIE.

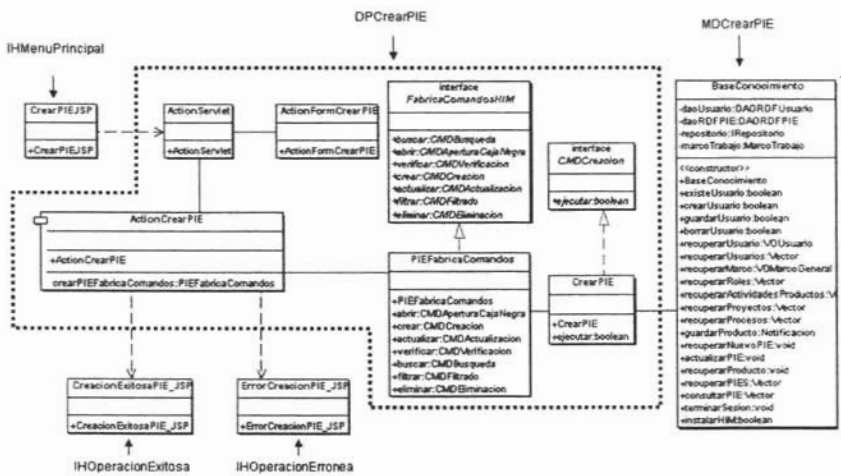


Figura 4.9. Diagrama de clases del diseño para la creación de un PIE.

En el diagrama anterior se muestra que la clase del análisis con nombre "IHMenuPrincipal" se mapea a su correspondiente clase del diseño como una página JSP conocida como "CrearPIE.jsp". Por otra parte, la clase "DPCrearPIE" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para la creación de un PIE. Este conjunto está conformado por las clases que implementan la funcionalidad del marco de trabajo "*Struts*" que son "ActionServlet", "ActionFormCrearPIE" y "ActionCrearPIE" y de dos clases que gestionan el proceso de creación "PIEFabricaComandos" y "CrearPIE". La clase "PIEFabricaComandos" al implementar la interfase "FabricaComandosHIM" hace uso del patrón de diseño GoF conocido como *Abstract Factory* (ver capítulo 1.4.1).

El hecho de utilizar este patrón viene de la situación de que en HIM se utilizan tres operaciones (creación, consulta y modificación) que plantean funcionalidades independientes (pueden estar consideradas como familias de objetos separadas) pero orientadas a un mismo producto PIE. Cada operación es considerada como un comando que se ejecuta al ser invocado por otro objeto (en este caso es el objeto de la clase "ActionCrearPIE"). Por este motivo cada método que se encuentra en la fábrica de comandos implementa el patrón "*Command*" (ver capítulo 1.4.3). Cuando se quiere almacenar un nuevo PIE, "ActionCrearPIE" crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "CrearPIE" (el cual implementa la interfase "CMDCreacion"), como dicho objeto es considerado un comando, "ActionCrearPIE" invoca al método "ejecutar" de "CrearPIE". El método "ejecutar" (que contiene la lógica necesaria para almacenar el PIE), invoca al método de la fachada, el cual está capacitado para realizar la creación (de aquí la clase "MDCrearPIE" es mapeada a dicho método). A continuación, en la Figura 4.10 se muestra el diagrama de secuencia para el caso de uso "creación PIE".

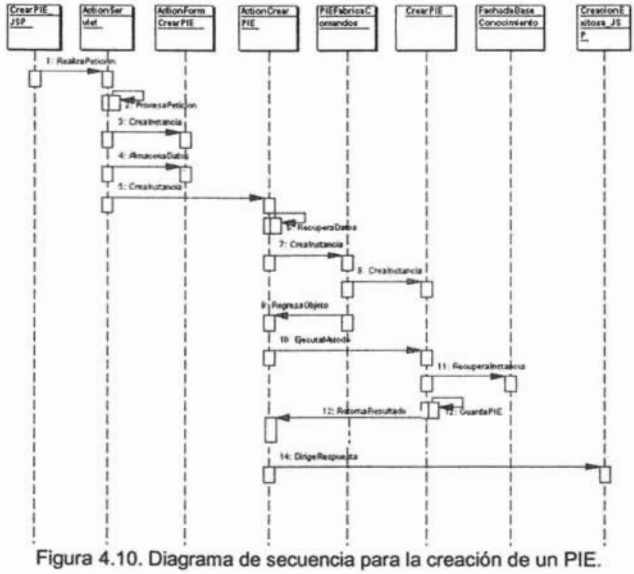


Figura 4.10. Diagrama de secuencia para la creación de un PIE.

En el diagrama anterior, cuando el usuario lleva a cabo la creación de un PIE, realiza una petición a la clase "ActionServlet" el cual crea un objeto de la clase "ActionFormCrearPIE" y almacena los datos que recoge de la interfaz para ese PIE en específico. Estos datos son pasados a la clase "ActionCrearPIE" para que sean procesados. "ActionCrearPIE" para almacenar el nuevo PIE, crea un objeto de la clase "PIEFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "CrearPIE". Dicho objeto es devuelto al objeto de la clase "ActionCrearPIE". Acto seguido se invoca al método "ejecutar" del objeto "CrearPIE". El método "ejecutar" invoca al método conocido como "guardarProducto" de la clase "BaseConocimiento" y se almacena el nuevo PIE en el repositorio.

4.5.2.2 Diagrama de clases y secuencia para consulta de un PIE.

En la Figura 4.11 se muestra el diagrama de clases del diseño para el caso de uso "consulta PIE". Una vez más se aprecia que en este diagrama existe una correspondencia con el diagrama de clases del análisis de la Figura 4.5 en la parte de consultar los datos de un PIE.

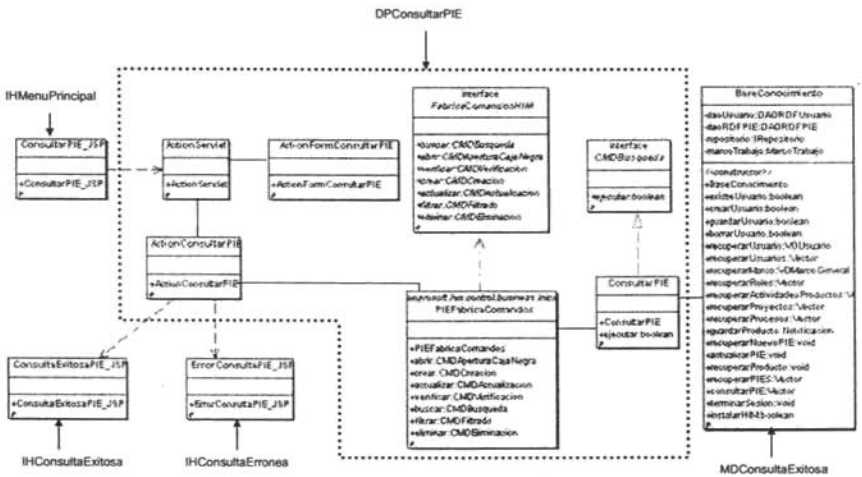


Figura 4.11. Diagrama de clases del diseño para la consulta de un PIE.

En el diagrama se muestra que la clase del análisis con nombre "IHMenuPrincipal" se mapea a una página JSP conocida como "ConsultarPIE.jsp". La clase "DPConsultarPIE" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para la consulta de un PIE. Este conjunto está conformado por las clases que implementan la funcionalidad del marco de trabajo "*Struts*" que son "ActionServlet", "ActionFormConsultarPIE" y "ActionConsultarPIE" y de dos clases que realizan la tarea de consultar los datos de un PIE: "PIEFabricaComandos" y "ConsultarPIE".

La clase "PIEFabricaComandos" implementa la interfase "FabricaComandosHIM". Cuando se quiere consultar los datos de un PIE, "ActionConsultarPIE" crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "ConsultarPIE" (el cual implementa la interfase "CMDBusqueda"), como dicho objeto es considerado un comando, "ActionConsultarPIE" invoca al método "ejecutar" de "ConsultarPIE". El método "ejecutar" (que contiene la lógica necesaria para consultar los datos del PIE), invoca al método de la fachada, el cual está capacitado para realizar la consulta de la información (de aquí la clase "MDConsultarPIE" es mapeada a dicho método).

A continuación, en la Figura 4.12 se muestra el diagrama de secuencia para el caso de uso "consultar PIE".

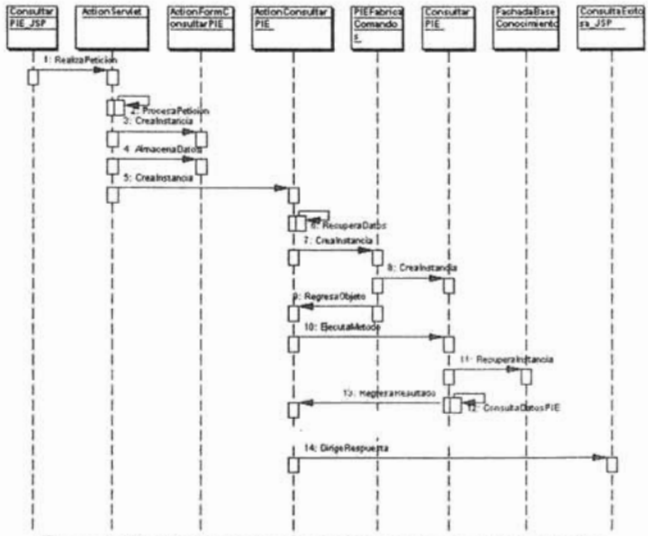


Figura 4.12. Diagrama de secuencia para la consulta de un PIE.

En el diagrama anterior, cuando el usuario lleva a cabo la consulta de un PIE, realiza una petición a la clase "ActionServlet" el cual crea un objeto de la clase "ActionFormConsultarPIE" y almacena allí los datos que recoge de la interfaz. Estos datos son pasados a la clase "ActionConsultarPIE" para que sean procesados. "ActionConsultarPIE" para consultar los datos del PIE, crea un objeto de la clase "PIEFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "ConsultarPIE". Dicho objeto es devuelto al objeto de la clase "ActionConsultarPIE". Acto seguido se invoca al método "ejecutar" del objeto "ConsultarPIE". El método "ejecutar" invoca al método conocido como "recuperarProducto" de la clase "BaseConocimiento" y se recupera los datos del PIE en cuestión. Si la operación se realiza correctamente se plasma dicho mensaje al usuario.

4.5.2.3 Diagrama de clases y secuencia para actualización de un PIE.

En la Figura 4.13 se muestra el diagrama de clases del diseño para el caso de uso "actualizar PIE". En este diagrama se puede apreciar que existe una correspondencia con el diagrama de clases del análisis de la Figura 4.5 en lo que respecta a la actualización de un producto PIE

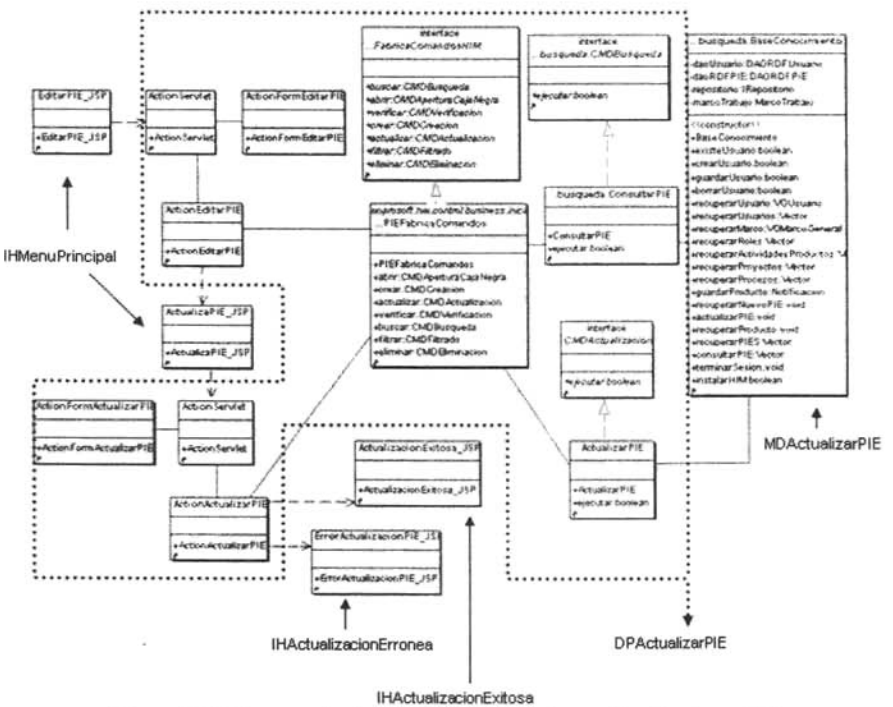


Figura 4.13. Diagrama de clases del diseño para la actualización de un PIE.

Para el proceso de actualizar los datos de un PIE, primero se debe de consultar los datos del PIE y posteriormente se modifican ó actualizan los datos en cuestión. En el diagrama se muestra que la clase del análisis con nombre "IHMenuPrincipal" primero se mapea a una página JSP conocida como "EditarPIE.jsp". La clase "DPActualizarPIE" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para la actualización de un PIE. Dicha actualización como ya se mencionó consta de dos pasos, la consulta del PIE en cuestión y la modificación de sus datos si así requiere.

En la consulta de los datos del PIE las clases involucradas que implementan la funcionalidad del marco de trabajo "*Struts*" son: "ActionServlet", "ActionFormEditarPIE"

y "ActionEditarPIE" y de dos clases que realizan la tarea de consultar los datos de un PIE: "PIEFabricaComandos" y "ConsultarPIE".

La clase "PIEFabricaComandos" implementa la interfase "FabricaComandosHIM". Cuando se quiere consultar los datos de un PIE, "ActionEditarPIE" crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "EditarPIE" (el cual implementa la interfase "CMDBusqueda"), como dicho objeto es considerado un comando, "ActionEditarPIE" invoca al método "ejecutar" de "EditarPIE".

El método "ejecutar" (que contiene la lógica necesaria para consultar los datos del PIE), invoca al método de la fachada, el cual está capacitado para realizar la consulta de la información.

La información del PIE se almacena en el objeto de la clase "ActionFormActulizarPIE" y posteriormente se carga dicha información en la interfaz del usuario (en la clase "ActualizarPIE.jsp" que fue mapeada de la clase "IHMenuPrincipal).

Para la modificación de los datos del PIE, el objeto de la clase "ActionActualizarPIE" crea una instancia de "ActionFormActulizarPIE" para recuperar de allí los datos modificados. "ActionActualizarPIE" también crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "ActualizarPIE" (el cual implementa la interfase "CMDActualizacion"). Como éste objeto es considerado un comando, "ActionActualizarPIE" invoca al método "ejecutar" de "ActualizarPIE". El método "ejecutar" (que contiene la lógica necesaria para actualizar los datos del PIE), invoca al método de la fachada, el cual está capacitado para realizar la actualización de la información.

Si la operación es exitosa se alerta de esta situación al usuario con la clase que representa al JSP "ActualizacionExitosa".

En la Figura 4.14 (siguiente página) se muestra el diagrama de secuencia para el caso de uso "actualizar PIE".

En dicho diagrama, cuando el usuario lleva acabo la actualización de un PIE, realiza una petición a la clase "ActionServlet" (a través de "EditarPIE.jsp") el cual crea un objeto de la clase "ActionFormEditarPIE" y almacena allí el nombre del PIE del cual se editará alguno de sus datos. Estos datos son pasados a la clase "ActionEditarPIE" para que sean procesados. "ActionEditarPIE" para consultar los datos del PIE, crea un objeto de la clase "PIEFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "ConsultarPIE". Dicho objeto es devuelto al objeto de la clase "ActionEditarPIE". Acto seguido se invoca al método "ejecutar" del objeto "ConsultarPIE". El método "ejecutar" invoca al método conocido como "recuperarProducto" de la clase "BaseConocimiento" y se recuperan los datos del PIE en cuestión. Estos datos se almacenan en el objeto de la clase "ActionFormActualizarPIE". Con los datos cargados en el objeto, se procede a desplegarlos en el JSP "ActualizarPIE". De esta manera el usuario puede realizar la modificación de los datos lo que trae consigo su posterior procesamiento a través de "ActionServlet".

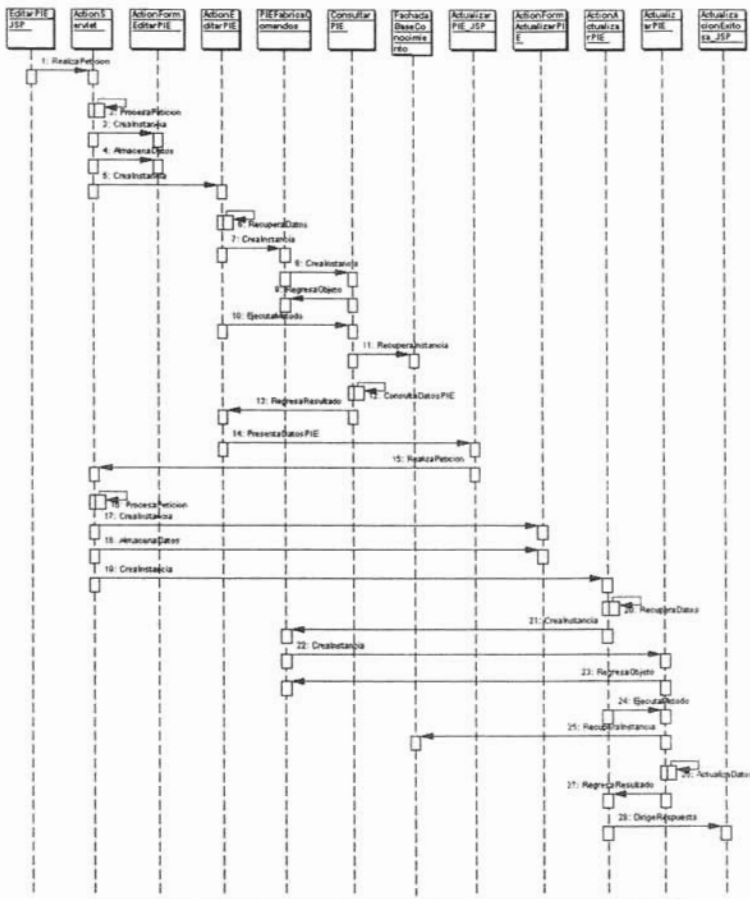


Figura 4.14. Diagrama de secuencia para la actualización de un PIE.

"ActionServlet" crea una instancia de la clase "ActionActualizarPIE" y le pasa los datos (probablemente modificados) del PIE. "ActionActualizarPIE" para actualizar los datos del PIE, crea un objeto de la clase "PIEFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "ActualizarPIE". Dicho objeto es devuelto al objeto de la clase "ActionActualizarPIE". Acto seguido se invoca al método "ejecutar" del objeto "ActualizarPIE". El método "ejecutar" invoca al método conocido como "actualizarPIE" de la clase "BaseConocimiento" y se modifican los datos del PIE en cuestión. Si la operación es exitosa se le indica al usuario a través del JSP "ActualzacionExitosa.jsp".

4.5.2.4 Diagrama de clases y secuencia para la creación (ingreso) de una caja negra.

En la Figura 4.15 se muestra el diagrama de clases del diseño para el caso de uso "creación de una caja negra". Este diagrama muestra una correspondencia con el diagrama de clases del análisis de la Figura 4.5 en lo que respecta a la creación de una caja negra. Por creación de una caja negra se refiere al hecho de poder incorporar al repositorio común un producto (ingresar dicho producto a la Base del Conocimiento). Esta incorporación tiene implícita la autenticación del usuario, lo cual le permitirá ejecutar dicha tarea.

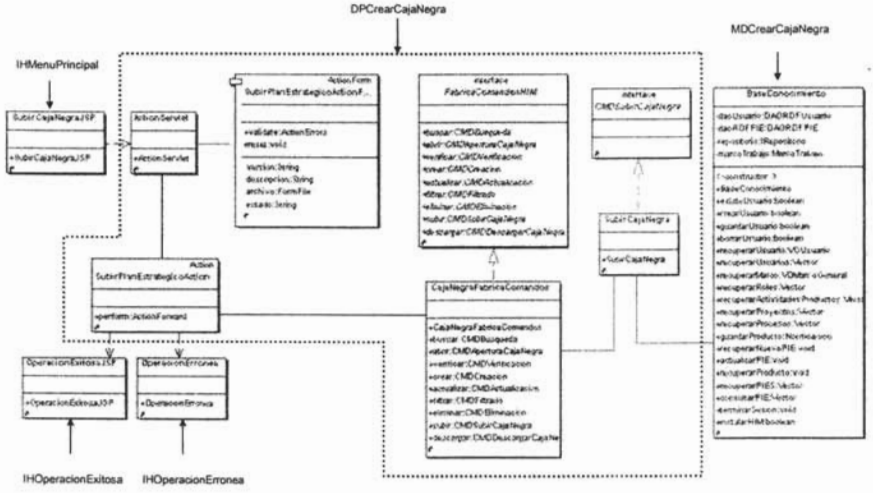


Figura 4.15. Diagrama de clases para la creación de una caja negra.

En el diagrama anterior se muestra que la clase del análisis con nombre "IHMenuPrincipal" se mapea a su correspondiente clase del diseño como una página JSP conocida como "SubirCajaNegra.jsp". Por otro parte, la clase "DPCrearCajaNegra" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para la creación de una caja negra. Este conjunto esta conformado por las clases que implementan la funcionalidad del marco de trabajo "*Struts*" que son "ActionServlet", "SubirCajaNegraActionForm" y "SubirCajaNegraAction" y de dos clases que gestionan el proceso de incorporar el nuevo producto al repositorio: "CajaNegraFabricaComandos" y "SubirCajaNegra". Al igual que en los PIE´s, en las cajas negras también se hace uso del patrón de diseño GoF conocido como *Abstract Factory* (ver capítulo 1.4.1), ya que la clase "CajaNegraFabricaComandos" implementa la interfase "FabricaComandosHIM".

Existen dos operaciones que se pueden llevar a cabo con una caja negra: incorporar dicha caja negra al repositorio ó bien descargarla del repositorio.

Cada operación (al igual que en los PIE´s) es considerada como un comando que se ejecuta al ser invocado por otro objeto (en este caso es el objeto de la clase "SubirCajaNegraAction"). Por ello cada método que forma parte de la fábrica de comandos implementa el patrón "*Command*" (ver capítulo 1.4.3). Cuando se quiere incorporar (subir) una nueva caja negra, "SubirCajaNegraAction" crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "SubirCajaNegra" (el cual implementa la interfase "CMDSubirCajaNegra"), como dicho objeto es considerado un comando, "SubirCajaNegraAction" invoca al método "ejecutar" de "SubirCajaNegra". El método "ejecutar" (que contiene la lógica necesaria para incorporar al repositorio la caja negra), invoca al método de la fachada, el cual es el encargado de realizar el ingreso de la nueva caja negra, dicho método es conocido como "guardarProducto". En la Figura 4.16 se muestra el diagrama de secuencia para el caso de uso que lleva a cabo la creación de una nueva caja negra.

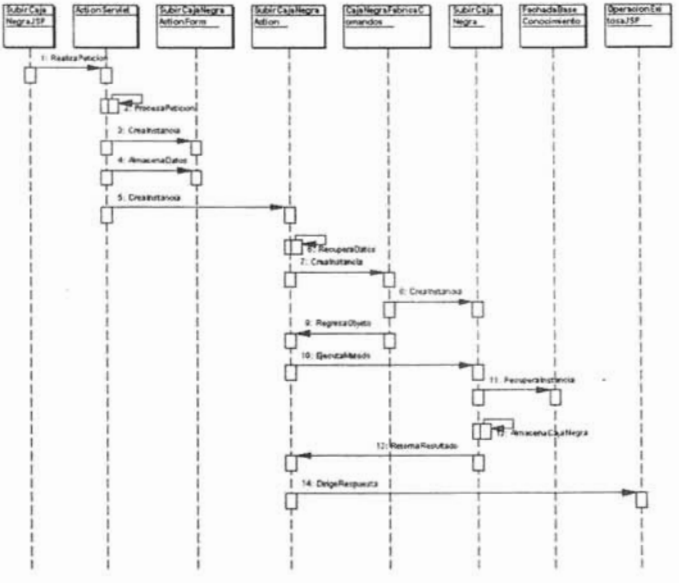


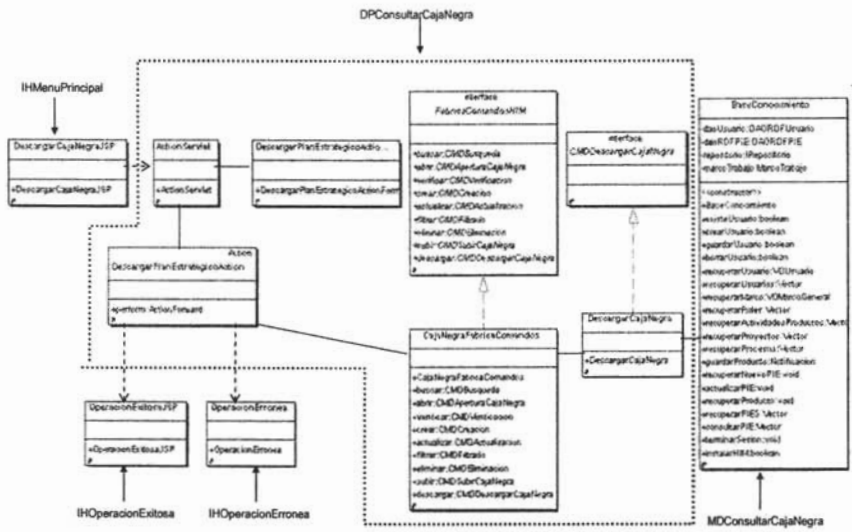
Figura 4.16. Diagrama de secuencia para la creación de una caja negra.

En el diagrama anterior, cuando el usuario lleva a cabo la creación de una caja negra, realiza una petición a la clase "ActionServlet" el cual crea un objeto de la clase "SubirCajaNegraActionForm" y almacena los datos que recoge de la interfaz para esa caja negra en específico. Estos datos son pasados a la clase "SubirCajaNegraAction" para que sean procesados.

"SubirCajaNegraAction" para almacenar la nueva caja negra, crea un objeto de la clase "CajaNegraFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "SubirCajaNegra". Dicho objeto es devuelto al objeto de la clase "SubirCajaNegraAction". Posteriormente se invoca al método "ejecutar" del objeto "SubirCajaNegra". El método "ejecutar" invoca al método conocido como "guardarProducto" de la clase "FachadaBaseConocimiento" y se almacena la nueva caja negra en el repositorio.

4.5.2.5 Diagrama de clases y secuencia para la consulta (descarga) de una caja negra.

En la Figura 4.17 se muestra el diagrama de clases del diseño para el caso de uso "consulta de una caja negra". Este diagrama muestra una correspondencia con el diagrama de clases del análisis de la Figura 4.5 en lo que respecta a la consulta de una caja negra. Por consulta de una caja negra se refiere al hecho de poder descargar desde el repositorio común un producto. Esta descarga tiene implícita la autenticación del usuario, lo cual le permitirá ejecutar dicha tarea.



En el diagrama anterior se muestra que la clase del análisis con nombre "IHMenuPrincipal" se mapea a una página JSP conocida como "DescargarCajaNegra.jsp".

La clase "DPConsultaCajaNegra" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para la consulta (descarga) de una caja negra. Este conjunto está conformado por las clases que implementan la funcionalidad del marco de trabajo "*Struts*" que son "ActionServlet",

"DescargarCajaNegraActionForm" y "DescargarCajaNegraAction" y de dos clases que realizan la tarea de descargar los datos de una caja negra: "CajaNegraFabricaComandos" y "DescargarCajaNegra".

La clase "CajaNegraFabricaComandos" implementa la interfase "FabricaComandosHIM". Cuando se quiere consultar los datos de un PIE, "DescargarCajaNegraAction" crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "DescargarCajaNegra" (el cual implementa la interfase "CMDDescargarCajaNegra"), como dicho objeto es considerado un comando, "DescargarCajaNegraAction" invoca al método "ejecutar" de "DescargarCajaNegra". El método "ejecutar" invoca al método conocido como "recuperarProducto" de la clase "FachadaBaseConocimiento" y se descarga la nueva caja negra desde el repositorio.

En la Figura 4.18 se muestra el diagrama de secuencia para el caso de uso "consultar caja negra".

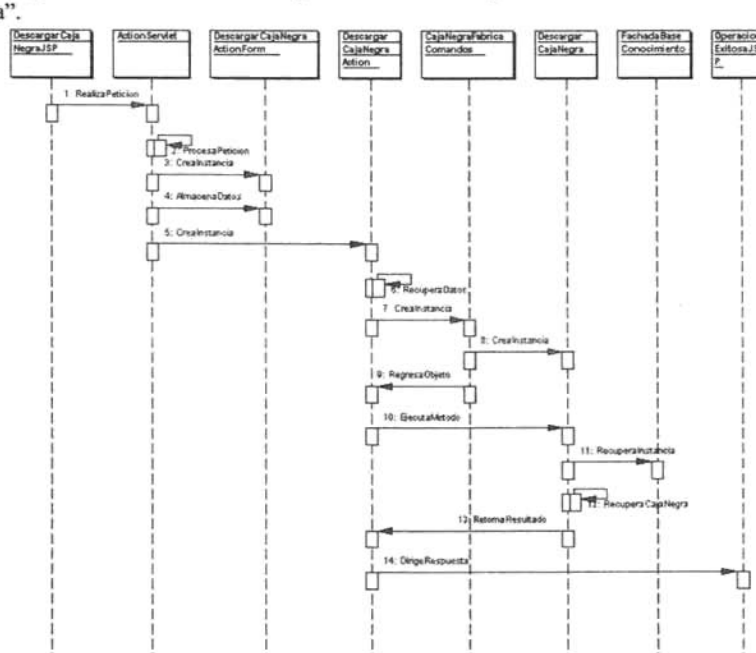


Figura 4.18. Diagrama de secuencia para la consulta de una caja negra.

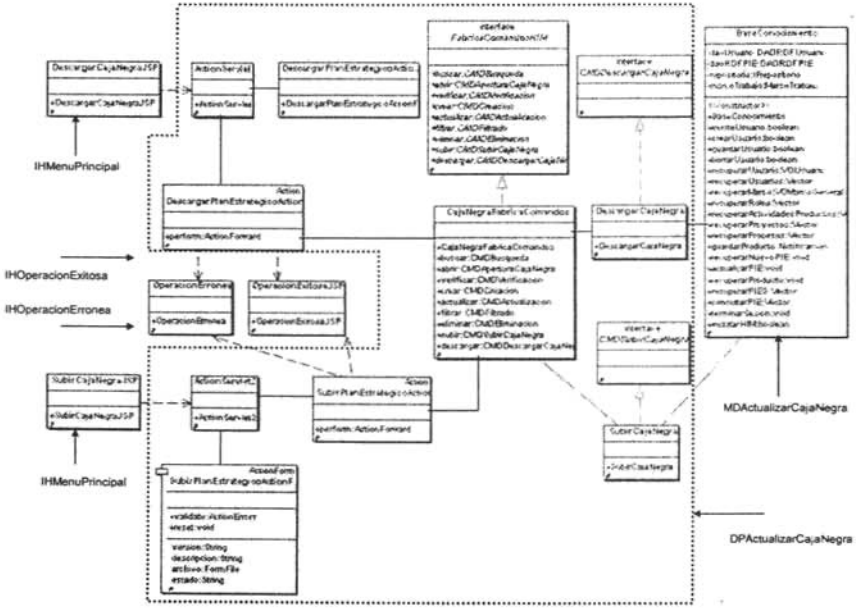
En el diagrama anterior, cuando el usuario lleva a cabo la descarga de una caja negra, realiza una petición a la clase "ActionServlet" el cual crea un objeto de la clase "DescargarCajaNegraActionForm" y almacena allí los datos que recoge de la interfaz.

Estos datos son pasados a la clase "DescargarCajaNegraAction" para que sean procesados. "DescargarCajaNegraAction" para descargar la caja negra, crea un objeto de la clase

"CajaNegraFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "DescargarCajaNegra". Dicho objeto es devuelto al objeto de la clase "DescargarCajaNegraAction". Posteriormente se invoca al método "ejecutar" del objeto "DescargarCajaNegra". El método "ejecutar" invoca al método conocido como "recuperarProducto" de la clase "FachadaBaseConocimiento" y se recupera la caja negra en cuestión. Si la operación se realiza correctamente se plasma dicho mensaje al usuario.

4.5.2.6 Diagrama de clases y secuencia para la actualización de una caja negra.

En la Figura 4.19 se muestra el diagrama de clases del diseño para el caso de uso "actualizar caja negra". En este diagrama se puede apreciar que existe una correspondencia con el diagrama de clases del análisis de la Figura 4.5 en lo que respecta a la actualización de una caja negra.



JSP conocida como "DescargarCajaNegra.jsp". La clase "DPActualizarCajaNegra" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para la actualización de una caja negra. Dicha actualización como ya se mencionó consta de tres pasos, la descarga de la caja negra en cuestión, la modificación de sus datos si así requiere y el ingreso nuevamente de la caja negra al repositorio.

Para la descarga de la caja negra las clases involucradas que implementan la funcionalidad del marco de trabajo "*Struts*" son: "ActionServlet", "DescargarCajaNegraActionForm" y "DescargarCajaNegraAction", y las clases que realizan la tarea de descargar los datos como tales son: "CajaNegraFabricaComandos" y "DescargarCajaNegra".

La clase "CajaNegraFabricaComandos" implementa la interfase "FabricaComandosHIM". Cuando se quiere descargar los datos de una caja negra, "DescargarCajaNegraAction" crea una instancia de la fábrica de comandos que a su vez devuelve una instancia del objeto "DescargarCajaNegra" (el cual implementa la interfase "CMDDescargarCajaNegra"), como dicho objeto es considerado un comando, "DescargarCajaNegraAction" invoca al método "ejecutar" de "DescargarCajaNegra". El método "ejecutar" invoca al método de la fachada, el cual está capacitado para realizar la descarga de la caja negra.

Con la caja negra en poder del usuario (previamente identificado según su Rol), éste cuenta con la libertad de poder realizarle alguna modificación.

Para depositar la caja negra (probablemente modificada), el usuario ejecuta de nuevo el proceso de ingreso (visto en la sección 4.5.2.4) de dicha caja negra al repositorio.

En la Figura 4.20 (siguiente página) se muestra el diagrama de secuencia para el caso de uso "actualizar caja negra".

En dicho diagrama, cuando el usuario lleva a cabo la actualización de una caja negra, realiza una petición a la clase "ActionServlet" (a través de "DescargarCajaNegra.jsp") el cual crea un objeto de la clase "DescargarCajaNegraActionForm" y almacena allí los datos necesarios para identificar la caja negra solicitada. Estos datos son pasados a la clase "DescargarCajaNegraAction" para que sean procesados. "DescargarCajaNegraAction" para descargar la caja negra, crea un objeto de la clase "CajaNegraFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "DescargarCajaNegra". Dicho objeto es devuelto al objeto de la clase "DescargarCajaNegraAction". Posteriormente se invoca al método "ejecutar" del objeto "DescargarCajaNegraAction". El método "ejecutar" invoca al método conocido como "recuperarProducto" de la clase "FachadaBaseConocimiento" y se recupera la información contenida en la caja negra.

De esta manera el usuario puede realizar la modificación de la caja negra descargada. El siguiente paso es ingresar la caja negra de nuevo al repositorio. Para ello el usuario realiza una petición al sistema mediante "SubirCajaNegra.jsp".

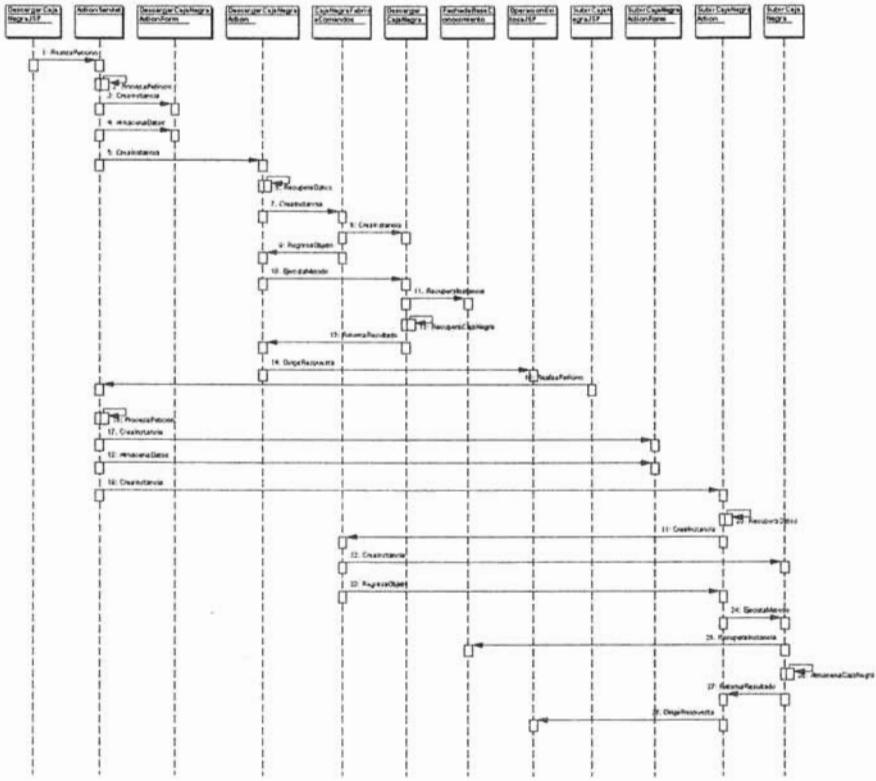


Figura 4.20. Diagrama de secuencia para la actualización de una caja negra.

La petición del usuario es atendida por "ActionServlet", el cual crea una instancia de la clase "SubirCajaNegraActionForm" y guarda los datos de identificación de la caja negra (probablemente modificada). "SubirCajaNegraAction" para ingresar de nueva cuenta la caja negra al repositorio, crea un objeto de la clase "CajaNegraFabricaComandos", éste a su vez realiza una instancia del objeto de la clase "SubirCajaNegra". Dicho objeto es devuelto al objeto de la clase "SubirCajaNegraAction". Acto seguido se invoca al método "ejecutar" del objeto "SubirCajaNegra". El método "ejecutar" invoca al método conocido como "guardarProducto" de la clase "FachadaBaseConocimiento" y se ingresa la caja negra al repositorio.

4.5.3 Diagrama de clases y secuencia para salir del sistema.

En la Figura 4.21 se muestra el diagrama de clases del diseño para el caso de uso **"A4.Salir del sistema".** En este diagrama se puede apreciar que existe una correspondencia con el diagrama de clases del análisis de la Figura 4.6.

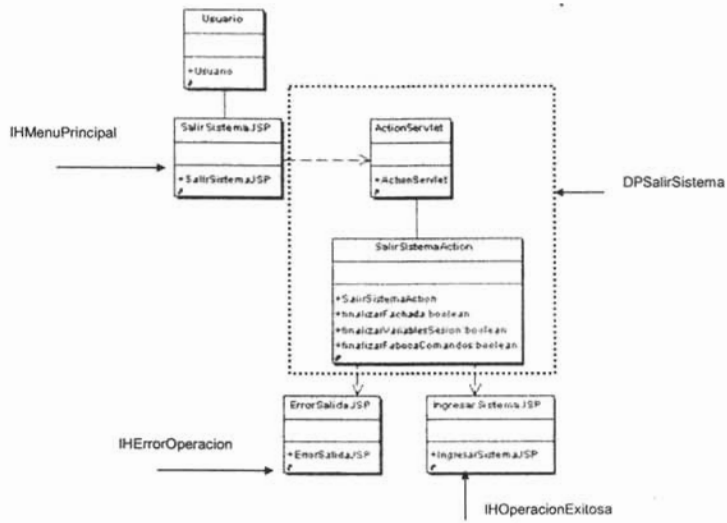


Figura 4.21. Diagrama de diseño para salir del sistema.

En el diagrama anterior se aprecia que la clase del análisis con nombre "IHMenuPrincipal" se mapea a su correspondiente clase del diseño como una página JSP conocida como "SalirSistema.jsp". Se aprecia también que la clase "DPSalirSistema" se mapea en un conjunto de clases (agrupadas con una línea de puntos) que son encargadas de cumplir la lógica del negocio para salir del sistema. Este conjunto esta conformado por las clases que implementan la funcionalidad del marco de trabajo "*Struts*" (ver capítulo 3.2) que son "ActionServlet", y "SalirSistemaAction". En ésta última clase, se lleva a cabo el proceso de finalización de todas las instancias que se pudieron haber creado cuando se ingreso al sistema. El primer objeto que se finaliza es aquel que proporciona todos los métodos de la fachada. Posteriormente se finalizan las variables de sesión que se crearon cuando un usuario en específico ingreso al sistema (por ejemplo las variable encargada de almacenar el nombre de usuario). Por último se finaliza la instancia de la fábrica de comandos que fue creada cuando se realizó alguna operación sobre un producto de HIM.

A continuación en la Figura 4.22 se muestra el diagrama de secuencia para el caso de uso **"A3.Salir al sistema"**

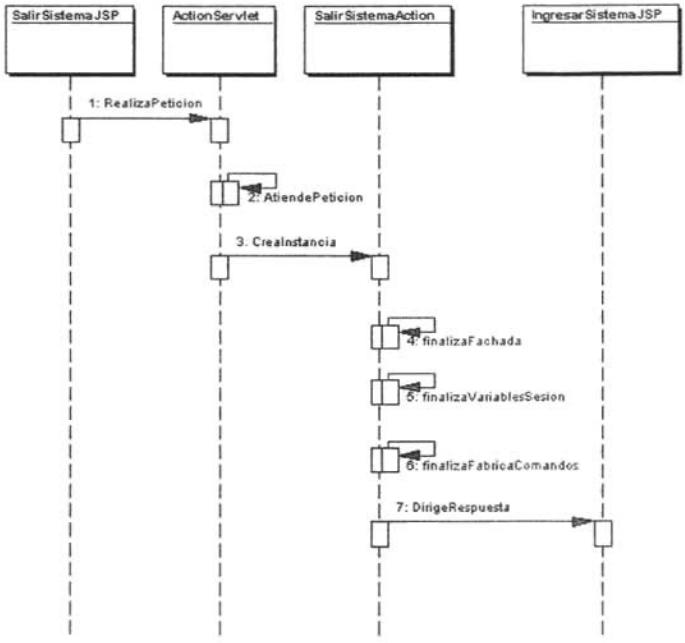


Figura 4.22. Diagrama de secuencia para salir del sistema.

En el diagrama anterior cuando el usuario quiere salir del sistema realiza una petición al "ActionServlet", dicho servlet atiende la petición creando un objeto de la clase "SalirSistemaAction". Este objeto contiene tres métodos encargados de finalizar todas las variables y objetos que fueron creados en la sesión del usuario. Si la operación de salida se concretó, "SalirSistemaAction" dirige la petición del usuario a la pantalla de inicio de la herramienta.

4.6 Diagrama de subsistemas

Con la estructura de paquetes de la capa de control de HIM y las clases principales ya identificadas tanto en el análisis como en el diseño, se presenta a continuación el diagrama de subsistema de la capa de control del HIM (ver Figura 4.23).

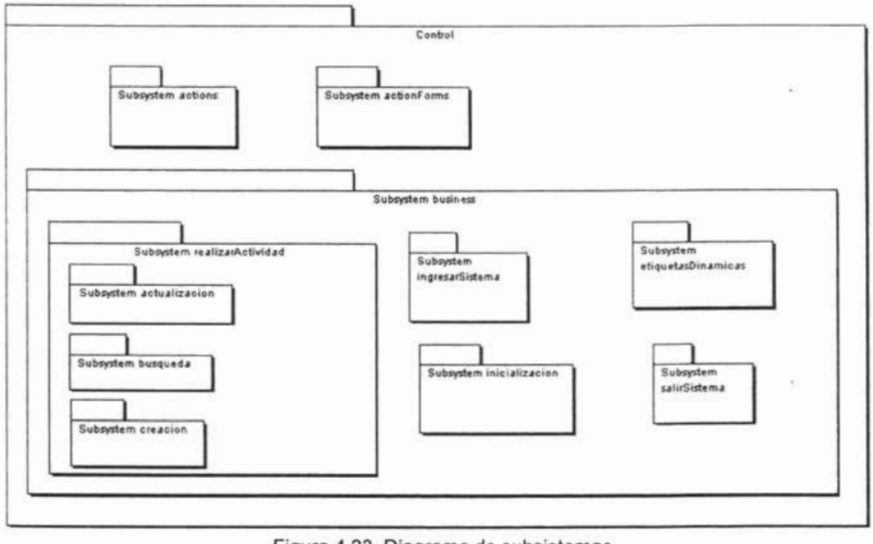


Figura 4.23. Diagrama de subsistemas

En cada subsistema de la figura anterior se agrupan las clases que tienen un significado semánticamente útil para el cumplimiento de la funcionalidad de la capa de control. Cabe mencionar, que el diseño del subsistema se realizó de una forma ascendente, es decir, se propuso el subsistema en base a las clases principales que ya se habían identificado para cumplir con la funcionalidad mínima de la capa de control de HIM. Dichas clases ó componentes se obtuvieron al realizar el análisis del marco de trabajo "*struts*" y también de los casos de uso generales de HIM. En la figura se aprecia que un subsistema a su vez puede contener a otros subsistemas, tal es el caso del subsistema "business", él cual, contiene a cinco subsistemas más: "realizarActividad", "ingresarSistema", "salirSistema", "etiquetasDinamicas", é "inicialización".

Por hacer mención únicamente, en el diseño descendente de los subsistemas se identifican los subsistemas de más alto nivel antes de que se hayan identificado sus clases ó componentes más importantes. Este diagrama concluye la parte de análisis y diseño de la capa de control de HIM.

CAPÍTULO 5. Implementación de la capa de control de HIM.

Este último capítulo abarca la fase de construcción de la capa de control de HIM. Para ello se requiere especificar los aspectos físicos del sistema, es decir, código fuente, bibliotecas, archivos, ejecutables, etc., que intervinieron en la construcción de la capa de control de HIM.

5.1 Introducción.

En la parte de análisis y diseño de la capa de control de HIM se plasmó el comportamiento del sistema desde un punto de vista lógico, es decir, se presentaron los diagramas de casos de uso, los diagramas de clases y los diagramas de secuencia. En la parte final de dicho comportamiento, se necesita plasmar todos los conceptos lógicos del sistema de una manera física, es decir, código que represente las clases e interfases que fueron parte de los diagramas antes mencionados. Cada entidad física generada es representada por un **componente**[16].

Cada componente según la arquitectura que se esté utilizando puede ser ejecutado en diferentes **nodos**[17]. Para el caso de HIM todos los componentes residen del lado del servidor y son ejecutados por las solicitudes de los usuarios (mediante un navegador WEB). Más adelante se presenta el diagrama de despliegue, en el cual, se visualizan los nodos que forman parte de la funcionalidad de HIM.

5.2 Diagrama de componentes.

Cada uno de los diagramas de clases presentados en el capítulo anterior tiene su correspondiente **diagrama de componentes**[18]. Cada componente que interviene en dicho diagrama corresponde con una o más clases ó interfases del diagrama de clases.

[16] Representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases e interfaces.
[17] Elemento físico que existe en tiempo de ejecución y representa un recurso computacional que por lo general dispone de memoria y capacidad de procesamiento.
[18] Conjunto de componentes y sus relaciones.

Hay que recordar que cada diagrama de clases contempla de manera abstracta los productos gestionados por HIM (PIE's y cajas negras -ver capítulo 4.2-). Cada uno de estos productos al ser implementados constituye una parte física. Dicha parte física debe de plasmar cada una de las clases o interfases utilizadas en un PIE ó caja negra en particular.

A continuación en la Tabla 5.1 se listan cada uno de los PIE's y cajas negras que fueron implementados en HIM, esto con el fin de indicarle al lector, que cada diagrama de componentes (más adelante presentados) contemplan tanto a un PIE como a una caja negra de una forma particular.

Proceso de MoProSoft	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: Registro de recursos humanos. Asignación de recursos humanos al proyecto. Lecciones aprendidas.
Bienes Servicios e Infraestructura	PIE: Solicitud de bienes ó servicios. Registro de proveedores. Registro de bienes ó servicios. Lecciones aprendidas.
Gestión de Proyectos	PIE: Descripción del proyecto. Registro del proyecto. Lecciones aprendidas.
Administración de Proyectos Específicos	PIE: Equipo de Trabajo. Lecciones aprendidas.
Desarrollo y Mantenimiento de Software	PIE: Lecciones aprendidas.

Tabla 5.1. Productos implementados en HIM.

En la tabla anterior se muestran todos los productos que fueron implementados por la capa de control de HIM. En la siguiente sección se mostrarán cada uno de los diagramas de componentes que se generaron para la capa de control de HIM. Cada uno de estos diagramas es aplicable a cualquier PIE ó caja negra de manera particular.

En la sección de Anexos se presenta todo el código generado para cada uno de los productos presentados en la Tabla 5.1.

5.2.1 Diagrama de componentes para "ingresar al sistema".

La Figura 5.1 muestra el diagrama de componentes para el caso de uso "ingresar al sistema", dicho diagrama se obtuvo a partir del diagrama de clases de la figura 4.7 (capítulo anterior).

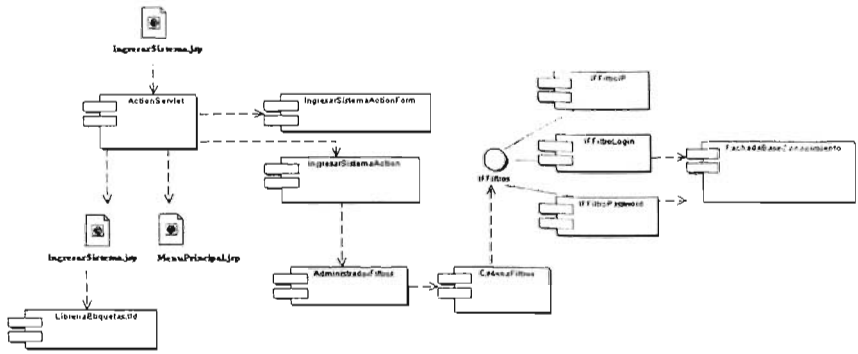


Figura 5.1. Diagrama de componentes para ingresar al sistema.

Al comparar ambas figuras (4.7 y 5.1 respectivamente) se aprecia que cada clase del diagrama de clases se convirtió en un componente. Cuando la clase se convierte en un componente su funcionalidad se mantiene igual. Por su parte cada clase que representa alguna interacción con el usuario se convirtió en un archivo con extensión JSP. Lo que se destaca de este diagrama es el componente que utiliza la librería de etiquetas de *struts* (ver capítulo 3.5.3) conocido como "LibreriaEtiquetas.tld". Dicho componente es invocado a través de "IngresarSistema.jsp" y su función es la de alertar al usuario de la existencia de un error en los datos de autenticación proporcionados.

5.2.2 Diagrama de componentes para "realizar actividad".

Realizar actividad es el caso de uso en el cual se lleva a cabo las operaciones de creación, consulta y actualización para un producto PIE ó en su caso, las operaciones de ingreso, descarga y actualización de una caja negra.

A continuación se presentan los diagramas de componentes para estas operaciones.

5.2.2.1 Diagrama de componentes "crear PIE".

El diagrama de componentes para la creación de un producto PIE se presenta en la Figura 5.2, tal diagrama se obtuvo a partir del diagrama de clases de la Figura 4.9 del capítulo anterior.

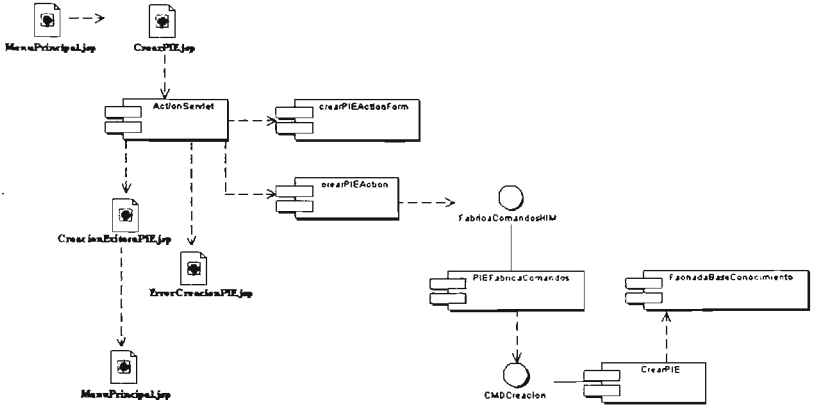


Figura 5.2. Diagrama de componentes para crear PIE.

Como se comentó anteriormente, cada clase del diagrama de clases se convirtió en un componente. Cada archivo con extensión ".jsp" representó la vía de interacción de HIM con el usuario. En este caso el usuario proporciona los datos del nuevo PIE en "CrearPIE.jsp" y se lleva a cabo la tarea.

El diagrama de la Figura 5.2 es aplicable para la creación de cada uno de los PIE's presentados en la Tabla 5.1. De esta manera, en la construcción de cada PIE, solo se sustituyó el prefijo "CrearPIE" (diagrama anterior) por el correspondiente PIE en cuestión, es decir, al implementar la creación del PIE conocido como "Registro del proyecto", cada componente del diagrama anterior fue nombrado como: "CrearRegistroProyecto.jsp", "CrearRegistroProyectoActionForm.java", "CrearRegistroProyectoAction.java", y "CrearRegistroProyecto.java" respectivamente. Esta notación es aplicable para los diagramas de componentes restantes.

5.2.2.2 Diagrama de componentes "consultar PIE".

La Figura 5.3 presenta el diagrama de componentes para la consulta de un producto PIE, dicho diagrama se obtuvo a partir del diagrama de clases de la Figura 4.11 del capítulo anterior.

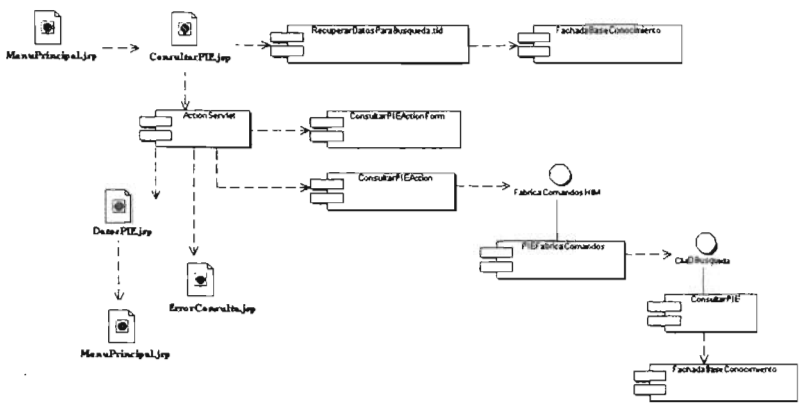


Figura 5.3. Diagrama de componentes para consultar PIE.

Para poder consultar todos los datos de un PIE en particular, primero se debe de mostrar los nombres de los PIE´s existentes y así poder seleccionar el PIE deseado. El componente encargado de realizar esta consulta es conocido como "RecuperarDatosParaBusqueda.tld" del diagrama anterior. Dicho componente es una etiqueta dinámica de "*Struts*" creada a la medida del requerimiento deseado (ver capítulo 3.5). La información entregada por esta etiqueta dinámica es desplegada en "ConsultarPIE.jsp", acto seguido el usuario selecciona el PIE del cual se quiere conocer todos sus datos y se lleva acabo la tarea.

El diagrama de la figura anterior es aplicable para la consulta de los datos de cada uno de los PIE's presentados en la Tabla 5.1.

5.2.2.3 Diagrama de componentes "actualizar PIE".

El diagrama de componentes para la actualización de un producto PIE se presenta en la Figura 5.4, tal diagrama se obtuvo a partir del diagrama de clases de la Figura 4.13 del capítulo anterior

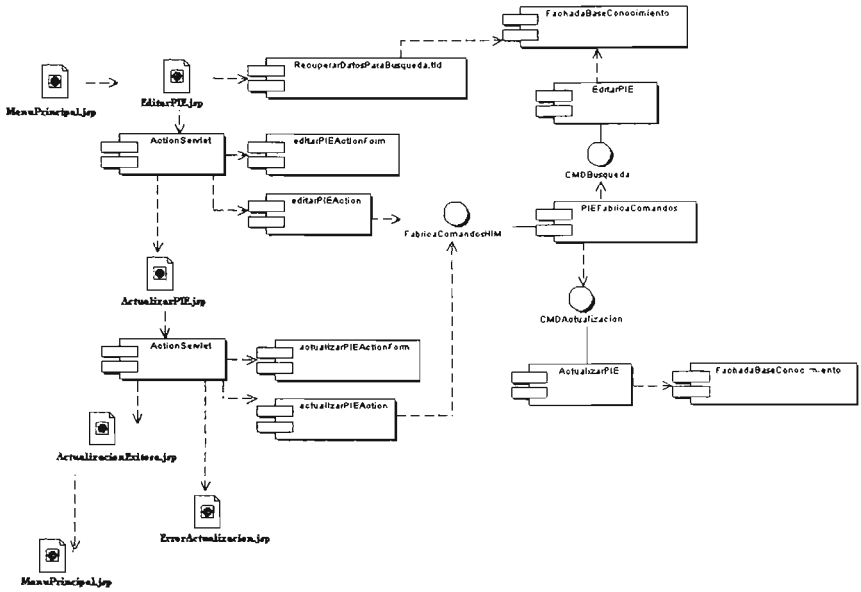


Figura 5.4. Diagrama de componentes para actualizar PIE.

Para poder actualizar los datos de un PIE en particular, primero se debe de mostrar los nombres de los PIE´s existentes y así poder seleccionar el PIE deseado. El componente encargado de realizar esta consulta es conocido como "RecuperarDatosParaBusqueda.tld" del diagrama anterior. Dicho componente es una etiqueta dinámica de "*Struts*" (ver capítulo 3.5). La información entregada por esta etiqueta dinámica es desplegada en "EditarPIE.jsp", posteriormente el usuario selecciona el PIE del cual se quiere editar sus datos y se recuperan. Los datos recuperados se presentan al usuario a través de "ActualizarPIE.jsp". Con los datos del PIE recuperados, el usuario tiene la libertad de poder modificarlos y así se completa dicha tarea.

El diagrama de la figura anterior es aplicable para la actualización de los datos de cada uno de los PIE's presentados en la Tabla 5.1.

5.2.2.4 Diagrama de componentes para la creación (ingreso) de una caja negra.

La Figura 5.5 presenta el diagrama de componentes para la creación de una caja negra, dicho diagrama se obtuvo a partir del diagrama de clases de la Figura 4.15 del capítulo anterior.

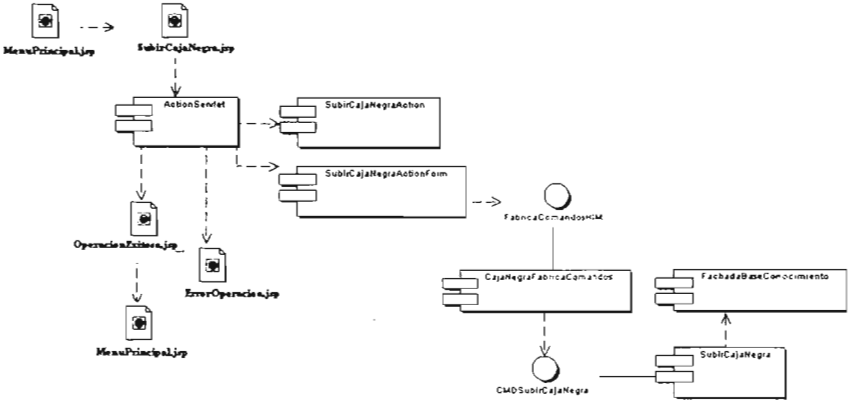


Figura 5.5. Diagrama de componentes para subir una caja negra.

En la creación de una caja negra se incorpora un nuevo producto de MoProsoft al repositorio, allí se almacenan todos los productos generados. Cada que se ingresa una caja negra se debe de indicar su versión y estado (validado, verificado, etc.) de dicho producto. Estos dos atributos los especifica el usuario a través de "SubirCajaNegra.jsp". Hecho lo anterior se ejecuta la tarea.

El diagrama de la figura anterior es aplicable para la creación de cada una de las cajas negras presentadas en la Tabla 5.1 (solo se implemento una caja negra).

5.2.2.5 Diagrama de componentes para la consulta (descarga) de una caja negra.

La Figura 5.6 presenta el diagrama de componentes para la consulta de una caja negra, dicho diagrama se obtuvo a partir del diagrama de clases de la Figura 4.17 del capítulo anterior.

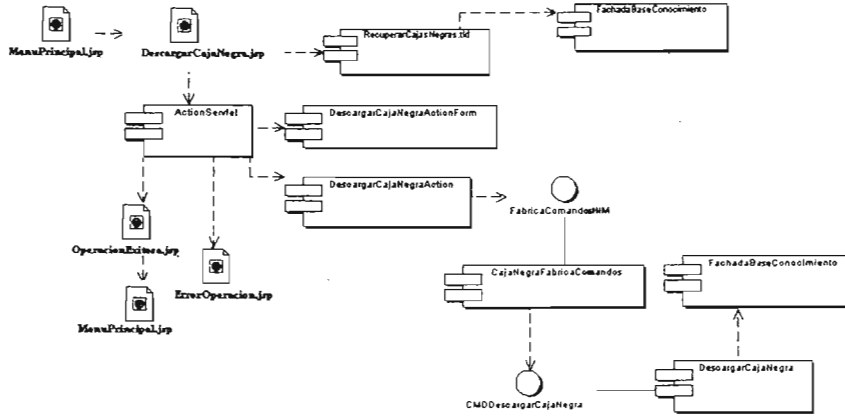


Figura 5.6. Diagrama de componentes para descargar una caja negra.

En la consulta de una caja negra se descarga un producto de MoProsoft desde el repositorio. Para poder consultar el contenido de una caja negra en particular, se debe de mostrar los nombres de las cajas negras existentes y así posteriormente seleccionar la caja negra deseada. El componente encargado de realizar esta consulta es conocido como "RecuperarCajasNegras.tld". Dicho componente es una etiqueta dinámica de "*Struts*" creada a la medida para realizar la búsqueda de todas las cajas negras existentes. En esta descarga se debe de especificar la versión y el estado del producto deseado. Estos dos atributos los especifica el usuario a través de "DescargarCajaNegra.jsp". Con estos datos se procede a descargar la caja negra.

El diagrama de la figura anterior es aplicable para la consulta de la caja negra presentada en la Tabla 5.1.

5.2.2.4 Diagrama de componentes para la actualización de una caja negra.

La Figura 5.7 presenta el diagrama de componentes para la creación de una caja negra, dicho diagrama se obtuvo a partir del diagrama de clases de la Figura 4.19 del capítulo anterior.

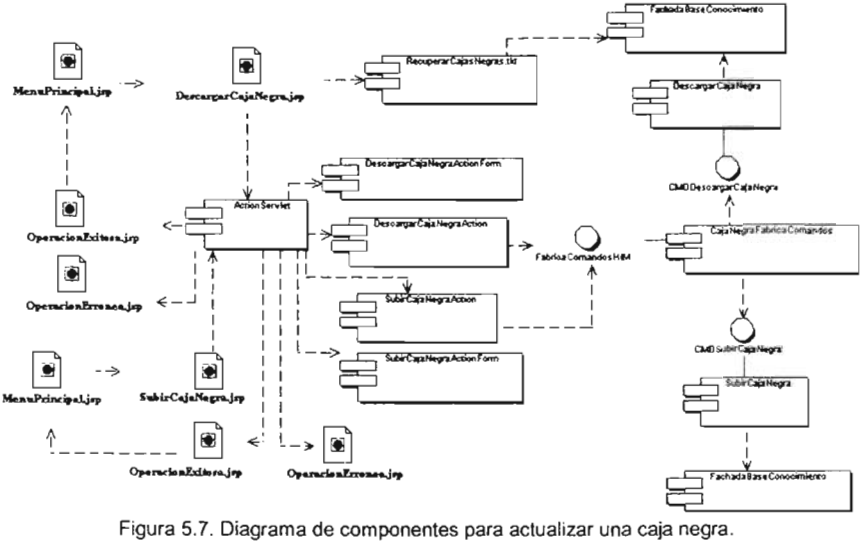


Figura 5.7. Diagrama de componentes para actualizar una caja negra.

Para poder actualizar la información de una caja negra en particular, primero se debe de mostrar los nombres de las cajas negras existentes y posteriormente seleccionar la caja negra deseada. El componente encargado de realizar esta consulta es conocido como "RecuperarCajaNegras.tld". La información entregada por esta etiqueta dinámica es desplegada en "DescargarCajaNegra.jsp", acto seguido el usuario seleccionará la caja negra de la cual tendrá la capacidad de editar su contenido. Con la caja negra en poder del usuario y posiblemente modificada, dicho usuario procede a ingresarla de nueva cuenta al repositorio (con una nueva versión y probablemente un estado diferente) mediante "SubirCajaNegra.jsp".

El diagrama de la figura anterior es aplicable para la actualización de la caja negra presentada en la Tabla 5.1.

5.2.3 Diagrama de componentes para "salir del sistema".

La Figura 5.8 presenta el diagrama de componentes para salir del sistema, dicho diagrama se obtuvo a partir del diagrama de clases de la Figura 4.21 del capítulo anterior.

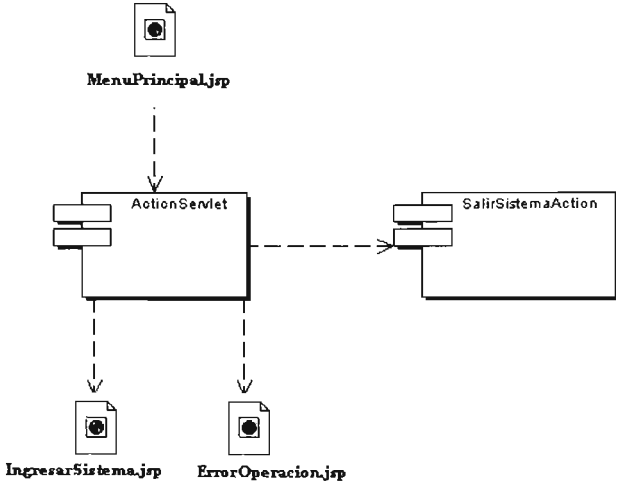


Figura 5.8. Diagrama de componentes para salir del sistema.

Cuando el usuario requiera salir del sistema podrá llevar a cabo dicho requerimiento mediante "MenuPrincipal.jsp". Al completarse esta tarea se dirigirá al usuario a la pantalla de entrada "IngresarSistema.jsp".

5.3 Diagrama de despliegue de HIM.

El diagrama de despliegue define la arquitectura física de HIM por medio de nodos que se encuentran interconectados, es decir, son elementos hardware sobre los cuales pueden ejecutarse los elementos software.

En el capítulo dos se mencionó que HIM es una herramienta que se ejecuta en WEB y por ello, cada cliente accede a HIM mediante un navegador. En lo que respecta a la capa de control de HIM, como tal no existe un diagrama de despliegue para dicha capa. El diagrama de despliegue debe de contemplarse para la herramienta completa. Lo que si se debe de indicar, es que la capa de control de HIM debe de radicar del lado del servidor WEB y a demás que las peticiones que reciba la capa deben de provenir de clientes que contengan un navegador. En la Figura 4.24 se muestra el diagrama de despliegue de HIM.

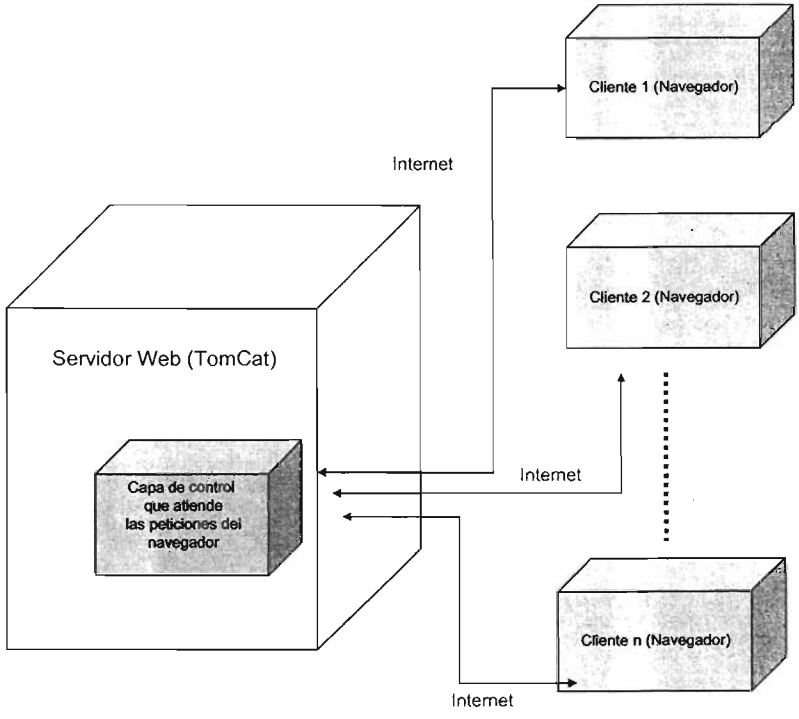


Figura 4.24. Diagrama de despliegue de HIM.

Con este diagrama de despliegue se concluye con la etapa de implementación de la capa de control de HIM y por lo tanto con el contenido de esta tesis.

Conclusiones

La utilización de MoProSoft para el desarrollo de su propia herramienta fue un factor importante para la consolidación de HIM. Mediante las actividades planteadas en MoProSoft se pudo realizar la planeación de la herramienta, dicha planeación incluyo labores tanto de administración como de desarrollo. Con estas actividades se logró conformar un equipo de trabajo que analizó los requerimientos de HIM y encaminó el desarrollo de cada una de las tesis.

En lo que respecta al desarrollo de la capa de control de HIM, el cumplimiento de sus objetivos fue un factor primordial para su exitosa implementación.

Con una investigación completa, se pudo concretar de una forma satisfactoria el análisis y selección de los patrones J2EE que más se adecuaron a los requerimientos de HIM.

La implementación de los patrones previamente seleccionados se lleva a cabo en su totalidad gracias a la correcta utilización de algunos "*Frameworks*" como es el caso de "*Struts*".

En lo que respecta a "*Struts*" se pudo aprovechar sus propiedades debido a una extensa investigación en lo concerniente a su funcionamiento. Gracias a la funcionalidad que brindó "*Struts*", se pudo llevar a cabo de forma rápida y eficiente la estructuración y construcción de la capa de control de HIM.

Con la elección adecuada de éstos diversos factores, se buscó, poder brindarle a HIM el carácter de mantenible, reusable, portable, disponible, escalable y segura.

Una vez más se pudo comprobar que la utilización adecuada del Proceso Unificado simplificó y facilitó en gran medida el desarrollo de HIM. Sin embargo, existieron algunos problemas en cuanto a la integración de las tres capas que finalmente se concluyó de forma satisfactoria.

Con la obtención de la herramienta HIM en su primera versión se pretende fomentar el uso de un modelo de procesos que le permita a las micro y mediana empresas gestionar todas y cada una de las actividades involucradas en el desarrollo de software de buena calidad.

Gracias a la buena asimilación de MoProSoft por diversas empresas de desarrollo de software, actualmente HIM esta siendo analizada para ser desarrollada con nuevas tecnologías de implementación como es .NET.

Bibliografía

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A pattern language. Oxford University Press, New York, 1977.

[Cop92] James O. Coplien. Advanced C++ programming styles and idioms. Addison Wesley, reading, MA, 1992.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Designpatterns, elements of reusable object-oriented software. Addison Wesley, 1995.

[SM02] Sthephen Stelting, Oval Maasen. Patrones de diseño aplicados a java. Prentice Hall 2002.

[DE1] Dirección Electrónica 1: http://www.javaworld.com/javaworld/jw-04-2001/jw- 0420-eplus.html

[DE2] http://struts.apache.org

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

[FGI+01] Jayson Falkner, Ben Galbraith, Romin Irani, Casey Kochmer, Sathya Narayana Panduranga, Krishnaraj Perrumal , John Timney, Meeraj Moidoo Kunnumpurath. Fundamentos de Desarrollo Web con JSP. Wrox, 2001.

[OKT+03] Hanna Oktaba, Claudia Alquicira Esquivel, Angelica Su Ramos. Modelo de Procesos para la Industria de Software (MoProSoft), software.net.mx, versión 1.1, mayo 2003.