UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

AUTOMATIZACIÓN DE PROCESOS INVOLUCRADOS EN UNA PYME (PEQUEÑA Y MEDIANA EMPRESA)

TESIS

PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA:

HUGO ADRIÁN MUÑOZ ANDRADE

DIRECTORA:

ING. GABRIELA BETZABÉ LIZÁRRAGA RAMÍREZ

2011

Universidad Nacional Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

Existen muchas personas importantes para mí y a quienes agradezco todo lo que han hecho. Sin embargo, antes que nada quiero agradecer a Dios por tener destinada para mí la familia que me regaló, pues sin ellos, difícilmente podría estar escribiendo estas líneas. Los amo.

Quiero agradecer a mis padres por el apoyo y cariño incondicional que me han dado. Agradezco todo su esfuerzo implicado desde que comencé mis estudios hasta el día de hoy. Esfuerzo que me ha convertido en la persona que ahora soy.

A mis hermanos: Griselda y Arturo, que siempre han estado en las buenas y en las malas tratando de ayudar a su hermano menor cuando los necesita. Gracias por todas esas risas y consejos.

Un agradecimiento a mis primos con quienes siempre he tenido momentos de diversión, y tíos que tienen palabras de aliento ante cualquier situación.

Gracias a todos mis amigos, primordialmente a los que conocí en la Facultad y de los que pude aprender muchas cosas, no sólamente relacionadas con la ingeniería. Una mención especial a Salvador que colaboró con su talento fotográfico ofreciendo las imágenes que acompañan el sistema.

Agradezco a la Ingenieria Gabriela Betzabé Lizárraga, mi directora de tesis, el tiempo invertido durante meses para las revisiones en la elaboración de este proyecto, a todos los profesores que formaron parte de mi desarrollo profesional y a los sinodales.

Gracias a mi Alma Máter, la Universidad Nacional Autónoma de México, por el conocimiento y los valores que inyectaron en mí durante casi una década. Y finalmente a la Facultad de Ingeniería por regalarme años maravillosos mientras me transformaba en ingeniero.

Por mi raza hablará el espíritu

Hugo

Convenciones Utilizadas:

- Los nombres de clases, interfaces, métodos, etiquetas, atributos, propiedades y archivos que sean utilizados en texto, son especificados con el tipo de letra `courier new`.
- Los fragmentos de código se encuentran enmarcados, numerados y mostrados con tipo de letra `Lucida Console`.

Índice

Objetivo…… 5
Planteamiento del problema…………………………………………………………………………………………5
Requerimientos del sistema……………………………………………………………………………………………6
Requerimientos del área de producción……………………………………………………………………………… 6
Requerimientos del área de ventas……………………………………………………………………………………7
Requerimientos de la página de Internet………………………………………………………………………………8

Capítulo 1: Lógica de Negocio…………………………………………………………………………… 9
1.1 Análisis y diseño de sistema OLTP (OnLine Transaction Processing)……………………… 10
1.1.1 Diagrama Entidad Relación…………………………………………………………………………11
1.1.2 Diagrama Jerárquico Funcional……………………………………………………………………12
1.1.3 Diagrama de Flujo de Datos……………………………………………………………………… 13
1.1.4 Tamaño de la Base OLTP……………………………………………………………………………19
1.2 Especificaciones Técnicas…………………………………………………………………………………23
1.2.1 La interfaz de usuario………………………………………………………………………………23
1.2.2 El color……………………………………………………………………………………………………24
1.2.3 Diseño………………………………………………………………………………………………………26
1.3 CSS………26
1.3.1 Estilos y Etiquetas…………………………………………………………………………………… 27
1.3.2 Estilos y Clases……………………………………………………………………………………………29
1.3.3 Estilos e Id's……………………………………………………………………………………………… 28
1.4 jQuery…… 32
1.4.1 Funciones de jQuery…………………………………………………………………………………… 32
1.4.2 Ajax (Asynchronous JavaScript and XML) ……………………………………………………… 34
1.4.2.1 load()……………………………………………………………………………………………… 35
1.4.2.2 ajax()……………………………………………………………………………………………… 35
1.4.2.3 post() / get()……………………………………………………………………………………37

Capítulo 2: Java y el mundo Web……………………………………………………………………………… 38
2.1 El protocolo HTTP…………………………………………………………………………………………………39
2.1.1 Solicitud de la petición…………………………………………………………………………… 40
2.1.2 Métodos de petición……………………………………………………………………………….. 40
2.2 La evolución de Java en el mundo web………………………………………………………………. 41
2.2.1 Servlets…………………………………………………………………………………………………….. 42
2.2.1.1 Funcionamiento de los Servlets…………………………………………….. 42
2.2.1.2 El API (Application Programming Interface) de los Servlets…….. 43
2.2.2 Java Server Pages (JSP) ……………………………………………………………………….. 48
2.2.2.1 El uso de scriptlets……………………………………………………………….. 48
2.2.2.2 JSTL y el lenguaje EL………………………………………………………….... 49
2.3 El patrón MVC (Model, View, Controller) ……………………………………………………….. 50
2.4 Struts………..52
2.4.1 Componentes y funcionamiento de Struts…………………………………………… 52
2.4.1.1 API de Struts…………………………………………………………………………… 52
2.4.1.2 Ciclo de vida de una petición en Struts…………………………………. 53
2.4.1 Configuración de Struts…………………………………………………………………………… 55
2.4.1.1 Configuración de ActionForm……………………………………………….. 56
2.4.1.2 Configuración de ActionMapping – Action – ActionForward……. 56
2.4.1.3 Configuración de ActionServlet en web.xml………………………….. 57
2.4.2 Librerías de Acciones JSP…………………………………………………………………….. 58
2.4.2.1 Librería Bean…………………………………………………………………………. 58
2.4.2.2 Librería Logic………………………………………………………………………….60
2.5 JDBC (Java DataBase Conectivity)…………………………………………………………………….. 62
2.5.1 API de JDBC…………………………………………………………………………………………….. 62
2.6 Hibernate……… 68
2.6.1 Componentes de Hibernate……………………………………………………………………. 69
2.6.2 Configuración de Hibernate…………………………………………………………………… 73
2.6.2.1 Configuración de Configuration……………………………………………. 73
2.6.2.2 Configuración de SessionFactory………………………………………….. 74
2.6.2.3 Configuración de Session……………………………………………………….75
2.6.2.4 Configuración del mapeo con archivo.hbm.xml……………………….. 76

2.6.2.5 Configuración del mapeo con anotaciones.................................. 77
2.6.2.6 Anotaciones para el mapeo de atributos.................................. 80
2.6.2.7 Anotaciones para el mapeo de asociaciones.............................. 82
2.7 Spring..86
2.7.1 Inyección de Dependencias...89
2.7.2 Integrando Spring con Struts...93
2.7.3 Integrando Spring con Hibernate.. 95
2.7.4 Manejo de Transacciones...98
2.7.4.1 La anotación @Transactional...98
2.7.4.2 Configuración.. 101

Capítulo 3: Implementación.. 103
3.1 Implementación del Cliente... 104
3.1.1 Hojas de estilo... 104
3.1.2 jQuery... 107
3.1.3 Plugins de jQuery.. 111
3.1.3.1 QuickPager.js.. 111
3.1.3.2 Validate.js.. 112
3.1.3.3 Datepicker.js.. 113
3.2 Implementación del Servidor... 115
3.2.1 Jerarquía de directorios..115
3.2.2 Módulos.. 115
3.2.3 DispatchAction.. 117
3.2.4 Arquitectura de la aplicación..119
3.2.5 Software involucrado en la construcción...122

Capítulo 4: Toma de Decisiones.. 123
4.1 Necesidad de un Cubo OLAP.. 124
4.2 Componentes y diseño.. 125
4.3 Implementación.. 127
4.3.1 ETL (Extracción Transformación y carga)....................................... 129
4.3.2 Dimensiones de lenta variación (SCD)...130

4.3.3 Implementación en Kettle…………………………………………………………………… 132

Apéndice A: El lenguaje EL y las librerías JSTL……………………………………………………………… 139

A.1 Variables implícitas EL………………………………………………………………………………… 140

A.2 Ámbito de datos…………………………………………………………………………………………… 141

A.3 Funciones de EL…………………………………………………………………………………………… 142

A.4 La librería Core…………………………………………………………………………………………… 143

A.5 La librería Format………………………………………………………………………………………… 144

A.6 La librería SQL…………………………………………………………………………………………… 146

Conclusiones……………………………………………………………………………………………… 149

Bibliografía y mesografía.. 151

Glosario…………………………………………………………………………………………………… 157

Índice de Figuras…………………………………………………………………………………………… 160

Índice de Tablas…………………………………………………………………………………………… 162

Índice de Códigos………………………………………………………………………………………… 165

Objetivo

Desarrollar un sistema web aplicado a una PyME dedicada al diseño, elaboración y venta de uniformes escolares; que se encargue de la optimización de ciertos procesos que aún se llevan a cabo manualmente. Tales como inventarios, gestión de producción y ventas, recursos invertidos, gastos y utilidades. Así como para tener un acercamiento con los clientes, intentando mejorar la experiencia de su compra y eventualmente, que nuevas escuelas conozcan el trabajo realizado por la empresa y confíen la producción de sus uniformes a ésta.

Con esto, se pretende obtener un ahorro de tiempo en la elaboración de reportes y tomar decisiones acertadas y soportadas por información real.

Planteamiento del problema

Confecciones Ana Laura es una PyME creada hace 25 años como un proyecto temporal y basado en su totalidad del conocimiento empírico de los integrantes. Debido al crecimiento inesperado, la administración de algunos procesos importantes fueron descuidados, resultado de la falta de conocimiento en ciertas áreas (ajenas al campo de aplicación de la empresa)

Actualmente, debido a la alta demanda de producción y a la expansión de la empresa se cuenta con bastantes problemáticas:

- No se tiene control adecuado de las compras de materia prima.
- Para saber si un uniforme se encuentra en existencia el vendedor debe buscarlo antes.
- La materia prima no se encuentra inventariada con datos exactos.
- Las producciones son registradas en papel y al final del año son contabilizados todos los uniformes producidos.
- Cada año la producción de tallas se basa en la experiencia, sin embargo, por falta de datos puntuales, algunas tallas se agotan mientras que otras no son vendidas.
- Muchos pedidos son retrasados debido a que no son comunicados a tiempo por olvido de los vendedores.
- Los pagos de los pedidos, así como las ventas, son calculados manualmente por el vendedor, por lo que no se está exento de fallas.

- La atención al cliente no puede ser personalizada en muchas ocasiones debido a la gran cantidad de estos y a la amplia variedad de productos.
- Al final del periodo, los cálculos para obtener la utilidad suelen ser muy demandantes e inexactos.

Por lo anterior descrito, es necesario el uso de un sistema computacional que una los diferentes procesos envueltos en la producción y venta, así como involucrar al cliente, de manera que su compra se realice de manera más cómoda.

Requerimientos del sistema

La empresa se divide en 2 áreas, que aunque trabajan en conjunto, realizan actividades diferentes, y debido a esto, ambas cuentan con información no siempre útil y necesaria para la otra. Estas áreas son:

1. Producción: Se encarga de la compra de materia prima, la confección y producción de los uniformes, la administración de los gastos y en conjunto con el área de ventas, del inventario de los recursos económicos.
2. Ventas: Encargada de la distribución de los uniformes y el envío de pedidos al área de producción.

Requerimientos del área de producción

- Se requiere tener un catálogo de las telas que se compran, así como de los materiales auxiliares, de igual manera un catálogo de los proveedores (actuales e históricos) con la información de precios, ubicación y teléfono.
- Se podrá consultar el inventario de la materia prima disponible y actualizarla (debido a que haya sido utilizada o bien se haya adquirido nueva).
- Se necesita que el sistema pueda estimar el costo unitario del uniforme por medio de los componentes requeridos para su elaboración.
- Debe existir un catálogo con los uniformes que se producen, sus precios e imágenes. Estos deben estar categorizados por escuela y grado (debido a que algunas secundarias cuentan con

uniforme diferente de acuerdo al año escolar). El precio de algunos uniformes depende de la talla.

- Cuando una producción está terminada se hace el conteo de todos los uniformes elaborados y se envía un inventario de producto terminado. Se requiere que las producciones queden registradas en el sistema con información de las piezas elaboradas, tallas, fecha de envío y fecha de producción.
- Se requiere que los pedidos levantados en el área de ventas puedan ser vistos en el área de producción, ordenados por fecha y prioridad para ser atendidos equitativamente.
- Una vez que un pedido es atendido, el sistema lo mostrará junto con su fecha de envío (debido a que no necesariamente al ser producido será enviado a la tienda), para que el cliente pueda ver a través de internet el día estimado en que podrá recoger su uniforme.
- Se requiere que el sistema sea capaz de mostrar reportes estadísticos, ya sea por escuela, o bien totales de:
    - Gastos
    - Ventas
    - Utilidades
    - Costos
    - Tallas ordenadas de acuerdo a su demanda

Requerimientos del área de ventas

- Se debe contar con un inventario de lo que hay en la tienda actualmente, en este deben mostrarse todos los uniformes existentes en el catálogo con su respectiva información y las piezas existentes en tienda (o bien si está agotado).
- Las ventas tienen un tratamiento especial dependiendo de la existencia de los uniformes:
    - Si hay en existencia, el sistema debe calcular el costo total de la venta y registrarla en el sistema.
    - Si no hay en existencia, se puede levantar un pedido (nombre del cliente, el/los uniforme(s) que aparta, la cantidad acreditada, teléfono, y observaciones extra que se puedan llegar a hacer). El sistema deberá devolver un número (que será único por pedido) y que le servirá al cliente para identificarlo, revisar su estatus por internet, o bien para hacer pagos a éste. Los pedidos pueden ser modificados

(o cancelados) debido a decisiones personales del cliente, por lo que el sistema deberá ser capaz de llevar a cabo dicha modificación. De igual manera el sistema debe registrar los pedidos que han sido pagados y entregados.

  - Se debe tener un control de los vendedores que levantaron los pedidos, así como el que realizó la entrega.

- Existen 3 tipos de cambio que deben ser gestionados por el sistema:
  - Cambio de talla
  - Cambio de uniforme
  - Cambio de talla o uniforme pero que no se encuentra disponible actualmente. En este caso existen 2 posibilidades:
    1. El cliente quiere la devolución de su dinero.
    2. Se levanta un nuevo pedido con el uniforme que se requiere.
- Tanto el personal del área de ventas, como del área de producción, deben autenticarse para hacer uso del sistema.

Requerimientos de la página de Internet

- Existirán 2 tipos de usuario: Registrados y no registrados.
- Los usuarios no registrados podrán tener acceso a la información estática de la página:
  - Catálogos de los uniformes con precios, foto, y demás datos.
  - Información de la empresa.
  - Ubicación de la tienda.
  - Recomendaciones sobre lavado de los uniformes.
- Los usuarios registrados, aparte de tener los mismos permisos que los no registrados:
  - Podrán revisar el estatus actual de su pedido, así como los datos con que fue dado de alta. Esto por medio del número de identificación que se les proporcionará al levantar su pedido.
  - Podrán hacer preguntas o comentarios en línea.
- Las preguntas hechas por los clientes podrán ser consultadas tanto en el área de ventas como en el área de producción para que den seguimiento de estas en el área pertinente.
- Para que un usuario se registre en el sistema, deberá proporcionar la siguiente información: Nombre, teléfono (tipo y extensión de ser necesaria) y dirección.

1

Lógica de Negocio

1.1 Análisis y diseño de sistema OLTP (OnLine Transaction Processing)

Una vez que se ha llevado a cabo el levantamiento de los requerimientos y se han estudiado detalladamente (resolviendo con el cliente las dudas resultantes) es importante tener en cuenta que gran parte del tiempo invertido durante el desarrollo se utilizará en la etapa del diseño, pues es en esta fase donde se hacen los bosquejos teóricos que serán implementados en etapas posteriores.

Estos bocetos incluyen:

- El diseño de la base de datos que recogerá la información del sistema para ser procesada por las diferentes funcionalidades de la aplicación. Este se muestra en el Diagrama Entidad/Relación (Figura 1.1) que detalla la descripción de las tablas.

- En el Diagrama Jerárquico Funcional (Figura 1.2) se señalan, de manera general, los componentes funcionales en los que se divide la aplicación. Partiendo del diagrama anterior se tienen las tareas relacionadas con las tablas volátiles, las no volátiles y los diferentes tipos de reportes requeridos.

- La navegabilidad del sistema, la interacción con sus actores y la interacción entre las funciones se obtiene puntualizando en cada una de las tareas definidas del diagrama anterior para así llegar al Diagrama de Flujo de Datos (Figura 1.3).

En las siguientes páginas se exponen estos diagramas que muestran la abstracción del funcionamiento del sistema con sus diferentes elementos.

1.1.1 Diagrama Entidad Relación

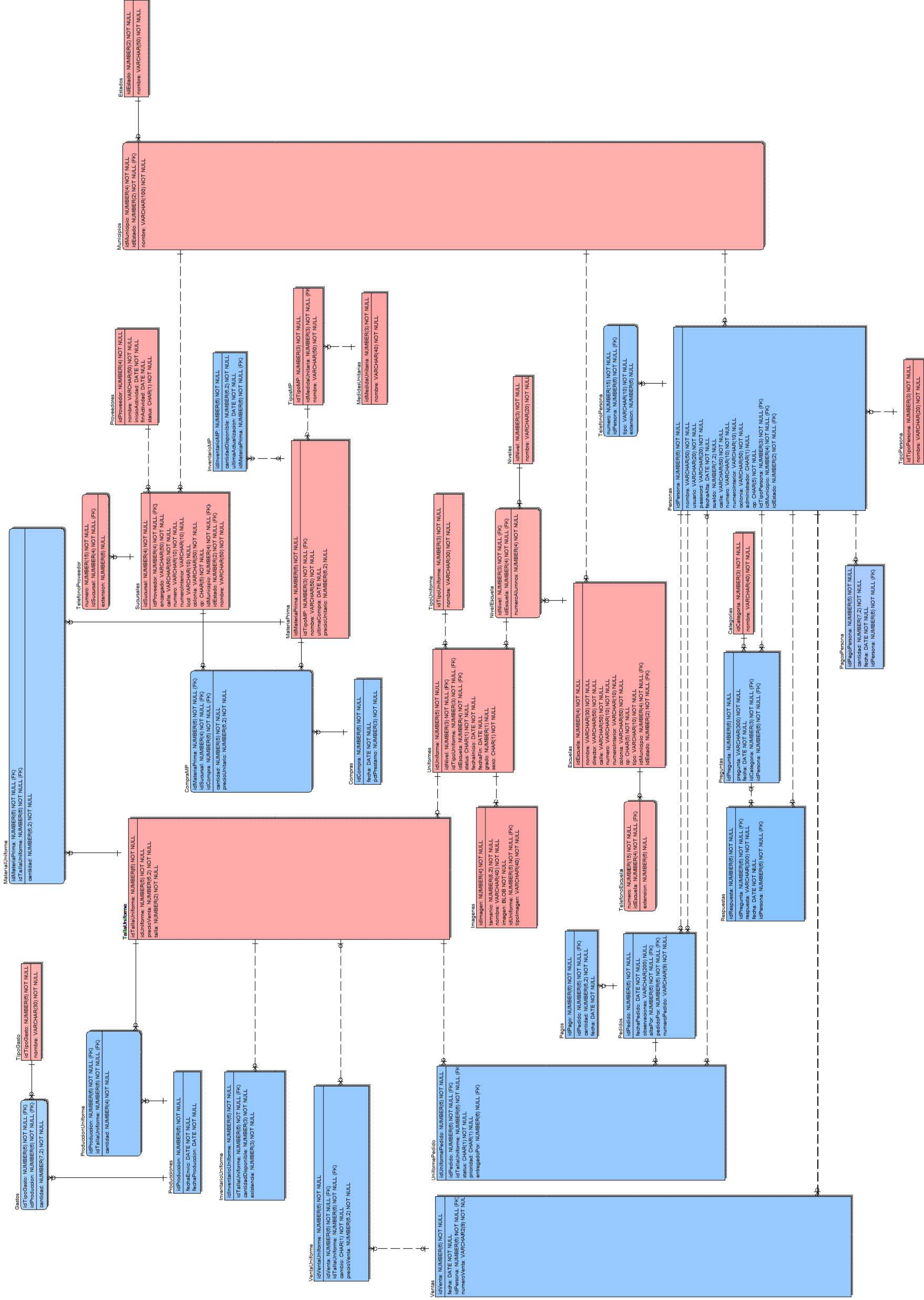


Figura 1.1: Diagrama Entidad Relación

1.1.2 Diagrama Jerárquico Funcional

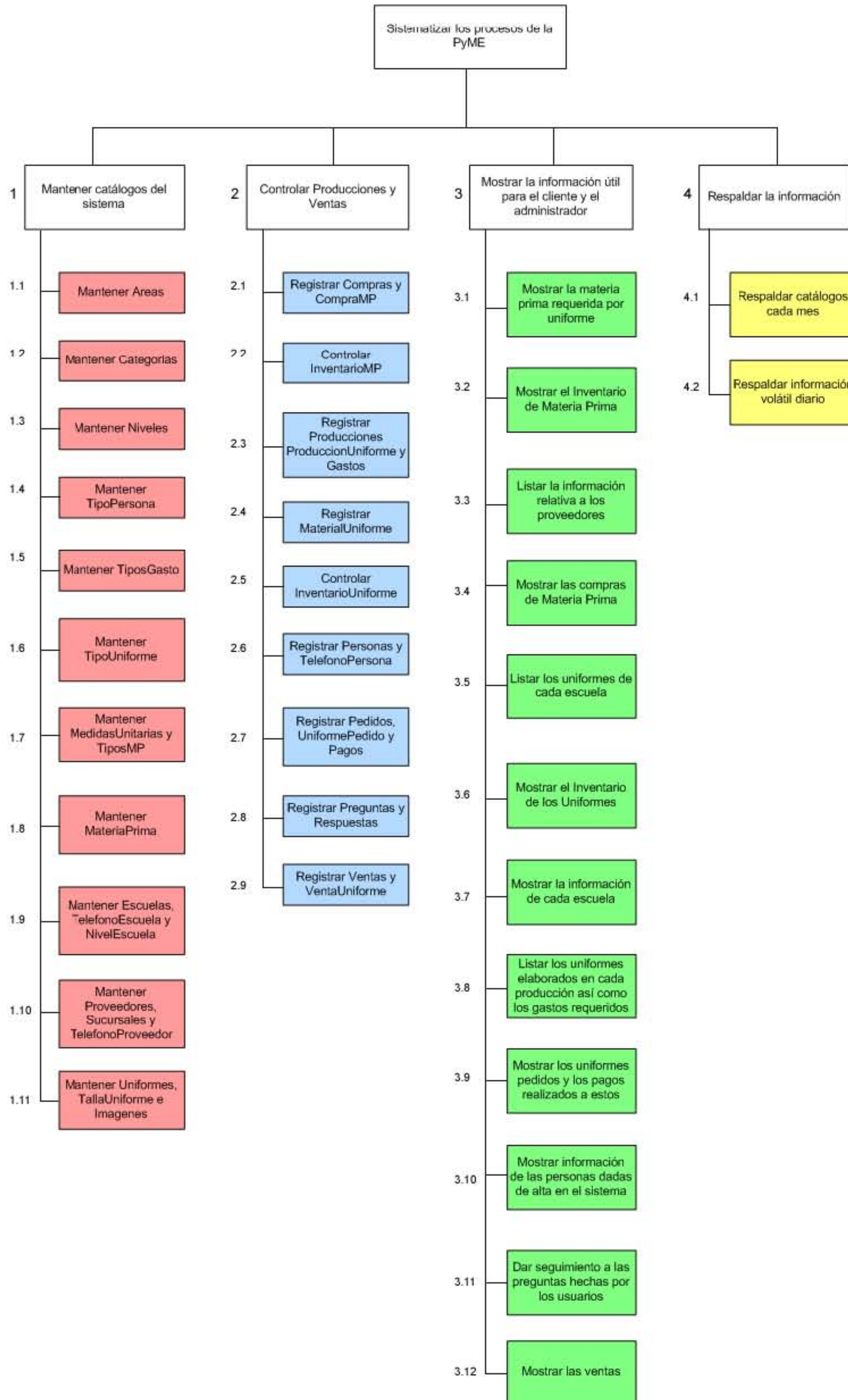


Figura 1.2: Diagrama Jerárquico Funcional

1.1.3 Diagrama de Flujo de Datos

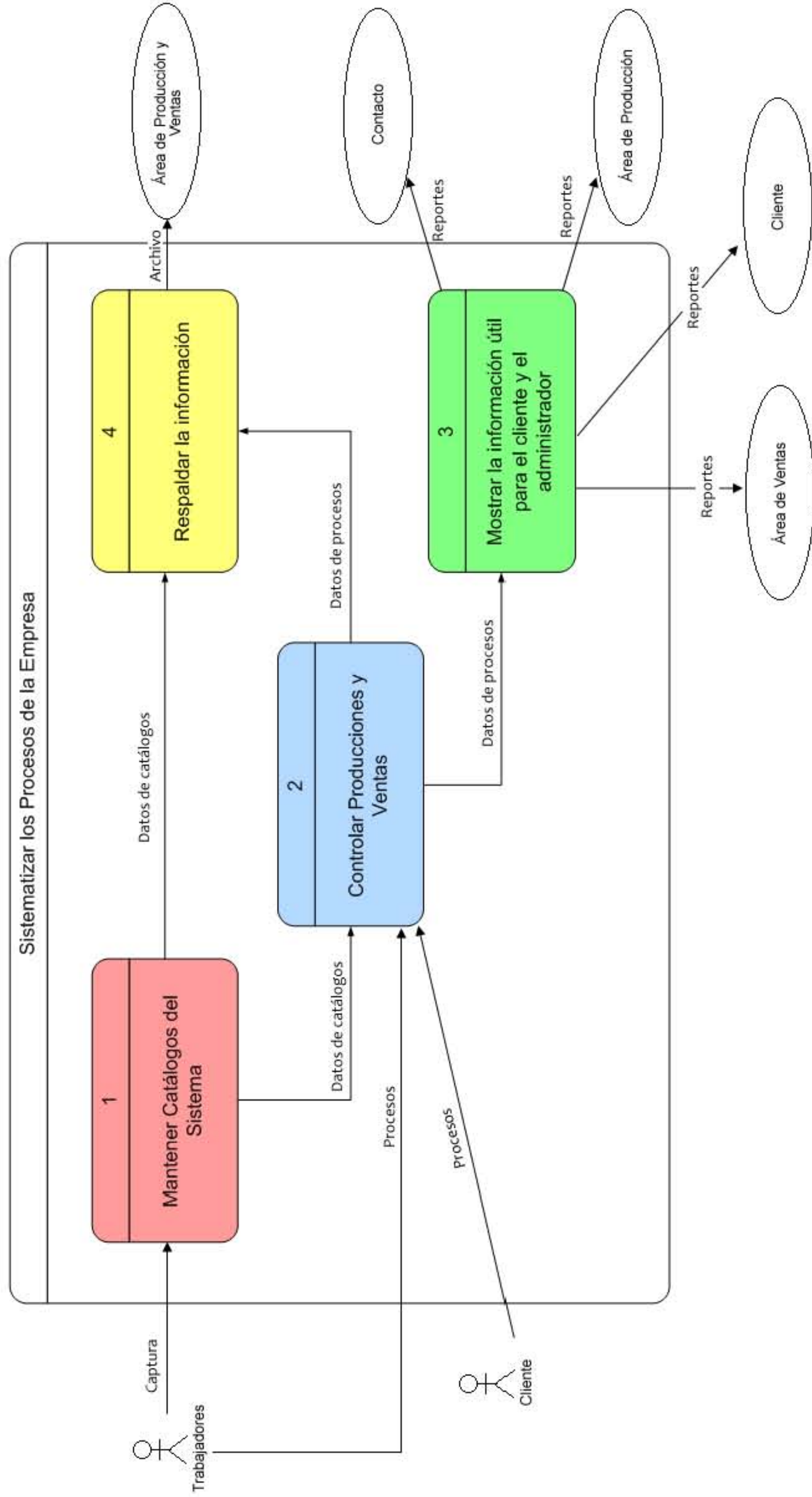


Figura 1.3: Diagrama de Flujo de Datos

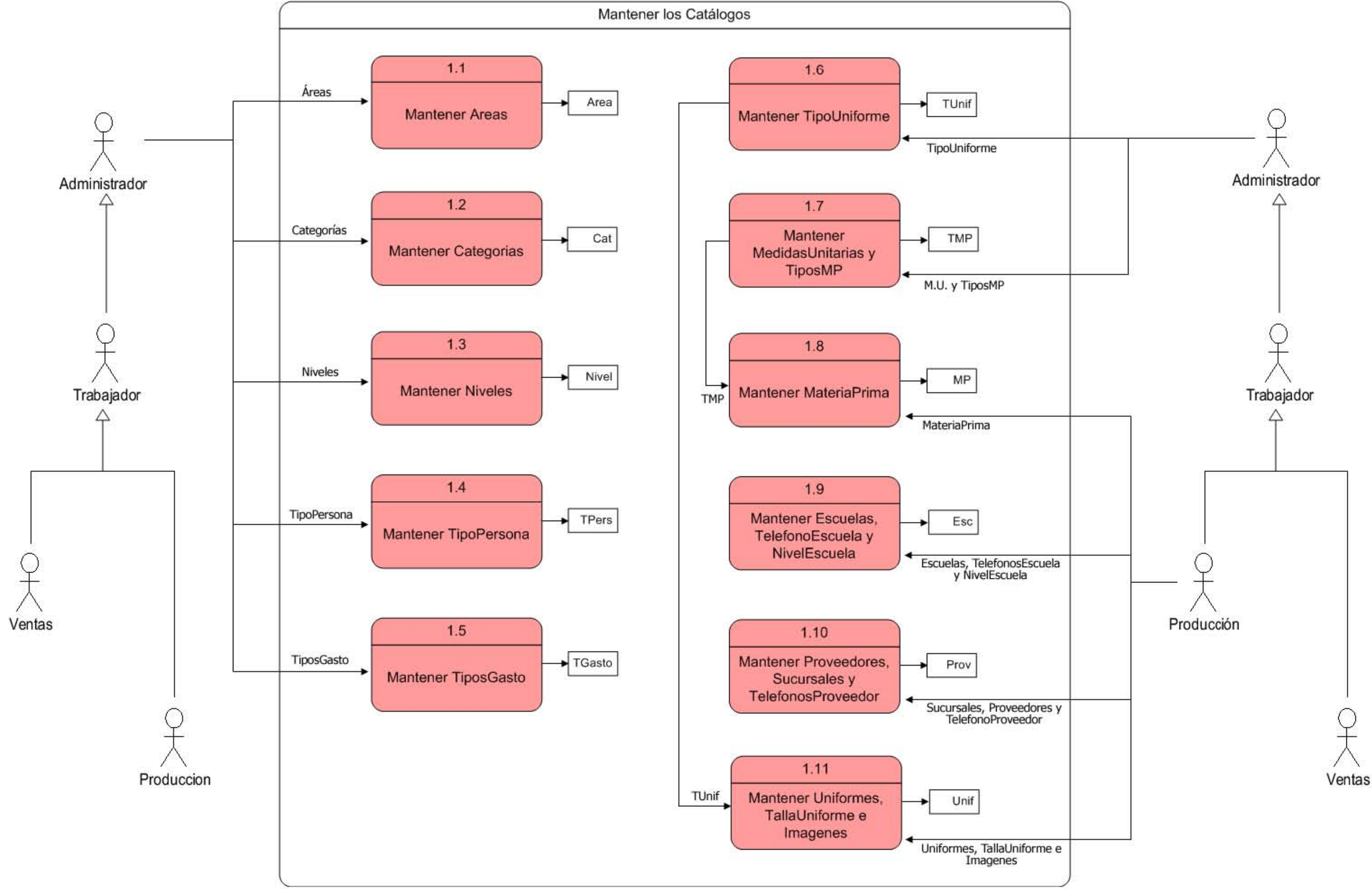


Figura 1.3 (*continuación*)

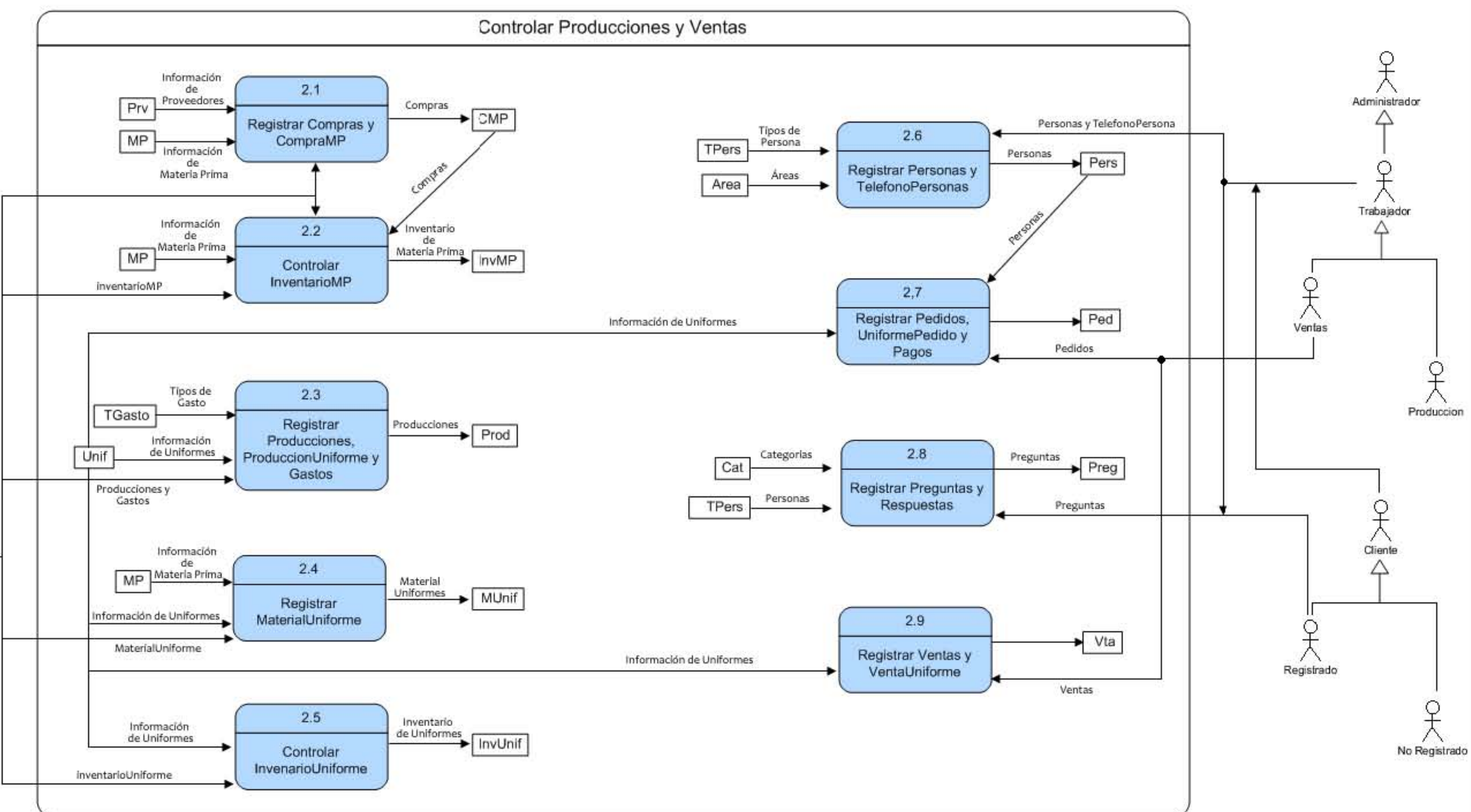


Figura 1.3 (*continuación*)

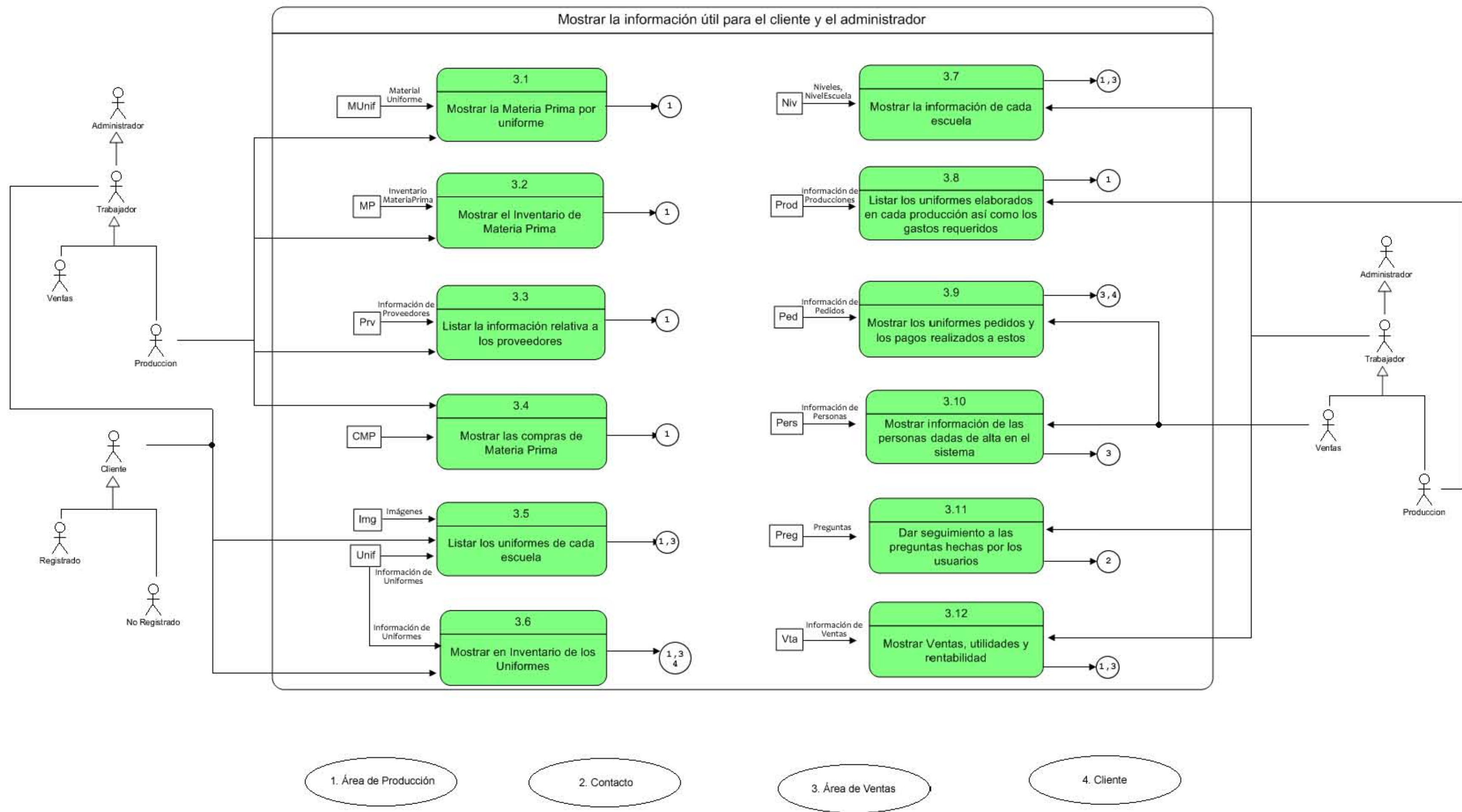


Figura 1.3 *(continuación)*

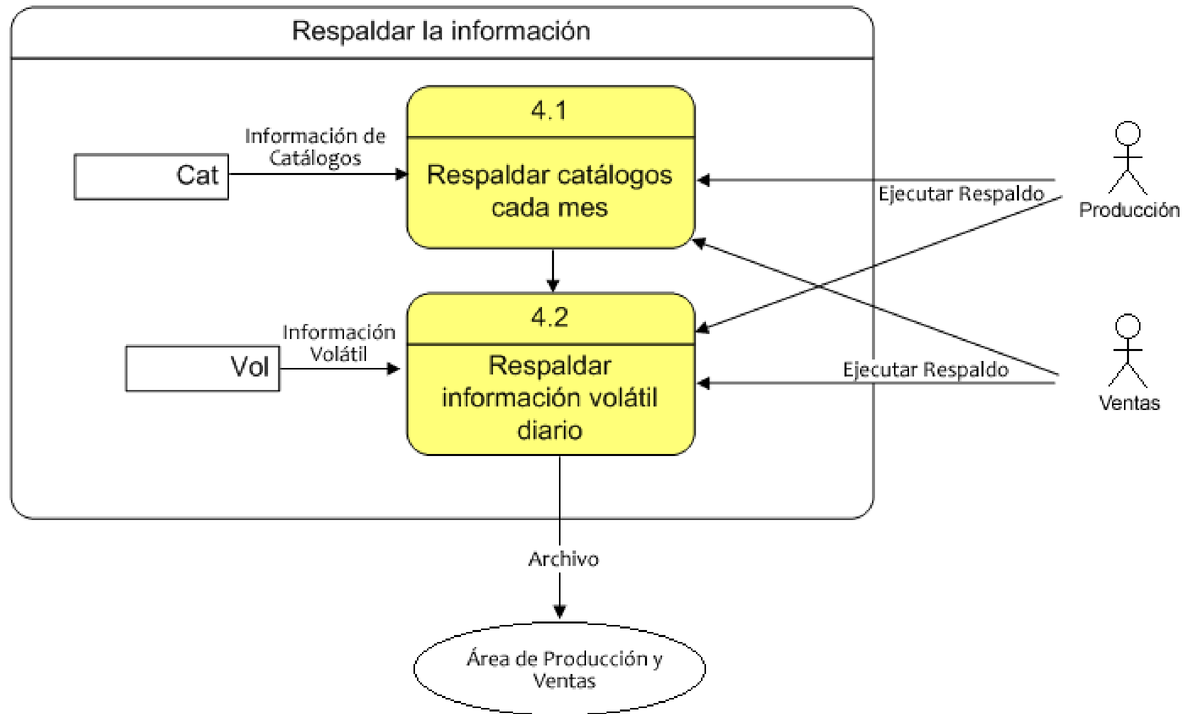


Figura 1.3 (*continuación*)

1.1.4 Tamaño de la Base OLTP

La información almacenada en la base de datos está organizada por medio de estructuras de almacenamiento lógicas (bloques, segmentos, tablespace) y físicas (archivos).

La base de datos se compone por al menos un tablespace, que es la unidad lógica de almacenamiento y en los cuáles se separan los diferentes grupos de objetos con que cuenta la base de datos (índices, tablas, usuarios). Cada tablespace tiene asociado diferentes archivos (datafiles) en los que se guarda la información física en el disco. El control de los tablespaces está dado por estructuras más pequeñas: bloques, segmentos y extensiones.

La unidad mínima de almacenamiento de datos es el bloque, compuesto por datos relativos a sí mismo (header) y los datos almacenados.

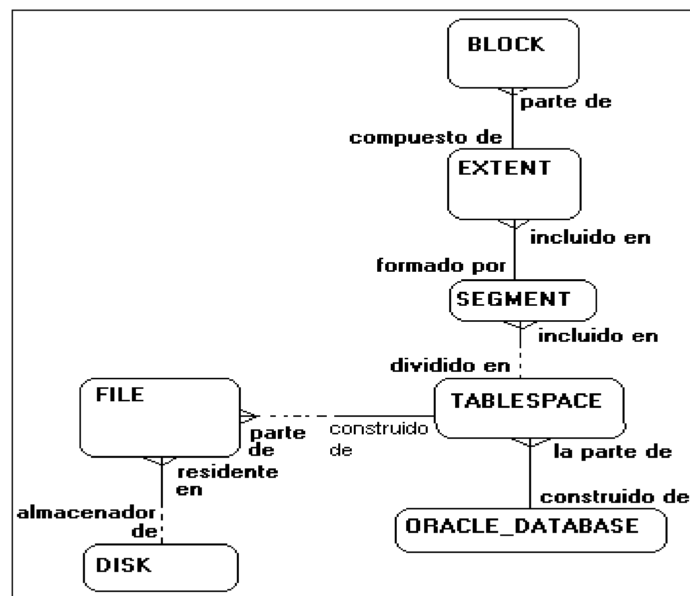


Figura 1.4: Almacenamiento físico en la Base de Datos

Un grupo de bloques forman la extensión, que tendrá un tamaño inicial fijo e irán creciendo conforme los datos y la configuración de ciertos parámetros: INITIAL, NEXT, PCTFREE, PCTUSED, INITTRANS, MAXTRANS, PCTINCREASE, MINEXTENTS y MAXEXTENTS (detallados más adelante).

Los segmentos están formados por extensiones y se encargarán de almacenar diferentes tipos de datos (rollback, temporales, índices y datos).

Es de suma importancia conocer el tamaño total de la base de datos para poder optimizar el rendimiento de esta, hacer un ajuste a su diseño (grado de normalización) o bien, elegir el hardware adecuado para su implementación.

Oracle ofrece ciertos parámetros en la creación de las tablas que permiten indicar el tamaño, número de extensiones y porcentaje de crecimiento que estas tendrán. A continuación se muestra la forma en que se llevan a cabo estos cálculos, que sumados, dan como resultado el tamaño total de la Base de Datos.

Tabla PROVEEDORES

Columna	**TipoDato**	**Longitud máxima**	**Longitud + byte de separación**
inicioActividad	DATE NOT NULL	7	8
nombre	VARCHAR(50) NOT NULL	30 + 20 = 50	51
idProveedor	NUMBER(6) NOT NULL	Ceil(6/2+1) = 4	5
status	CHAR(1) NOT NULL	1	2
finActividad	DATE NOT NULL	7	8

74 + 5 = 79

Posible espacio Libre:
20 (debido al varchar2)

Longitud de Registro:
79

Número de Registros
200

Tamaño de bloque
8 K

Valor constante del Header
360 bytes

Bytes libres del bloque
7832

– **INITIAL:**

Es el tamaño de la extensión inicial, asignada una vez que se crea el objeto.

Initial = (bloques necesarios)(tamaño de bloque)

Bloques necesarios = (registros)(longitudRegistro)/bytesLibres

= (200)(79)/7832 = 2 bloques

INITIAL = 2 x 8 = 16 K

- **NEXT:**

Es el tamaño de la segunda extensión, una vez que INITIAL se ha llenado. Es recomendable que tanto INITIAL como NEXT sean del mismo valor.

- **PCTFREE:**

Indica el porcentaje de espacio libre que tendrá la tabla en cada renglón, debido a la existencia de datos nulos y de tipo VARCHAR2.

Se obtiene por medio de una regla de 3:

Si la longitud máxima de registro es el 100%, el posible espacio libre es X%

X = 100 (posibleEspacioLibre)/ longitudMaxima

= (100)(20)/79 = 25.3

PCTFREE = 25%

- **PCTUSED:**

Es el porcentaje mínimo de espacio requerido para la inserción de datos en un renglón.

Para calcular este parámetro, primero se calcula el porcentaje de espacio que ocupa cada registro

Si el tamaño de bloque es el 100%, la longitud del registro es el X%

X = 100(longitudDeRegistro)/tamañoDeBloque

= 100(79)/ 7832 = 1%

PCTUSED = 100 – (PCTFREE + X)

=100 – (25 + 1) =74

PCTUSED = 74%

- **INITTRANS:**

Número inicial de transacciones concurrentes en el bloque.

- **MAXTRANS:**

Número máximo de transacciones concurrentes en el bloque.

Se calcula dividiendo el tamaño del bloque entre el tamaño de registro.

MAXTRANS = 7832 / 79

MAXTRANS = 99

- **PCTINCREASE:**

Indica el porcentaje de crecimiento a partir de la tercera extensión (una vez que INITIAL y NEXT se llenan).

- **MINEXTENTS:**

Número mínimo de extensiones que serán asignadas a un objeto.

- **MAXEXTENTS:**

Número máximo de extensiones que un objeto podrá tener (el tamaño de estas dependerá del PCTINCREASE).

En la siguiente tabla se indica el resumen de los parámetros calculados para cada una de las tablas de la base de datos clasificadas según la volatilidad de su información y pensadas a un crecimiento de 10 años.

Tabla	Tipo	Registros [Anual]*	INITIAL [K]	NEXT [K]	PCTFREE[%]	PCTUSED[%]	INITTRANS	MAXTRANS	PCTINCREASE	MINEXTENTS	MAXEXTENTS	Registros [10 años]**	Tamaño a 10 años
Areas	Catálogo	2	64	64	34	66	1	255	50	1	1	2	64
Categorias	Catálogo	5	64	64	20	80	1	160	50	1	1	10	64
Escuelas	Catálogo	10	64	64	34	66	1	33	50	1	1	20	64
Estados	Catálogo	2	64	64	42	58	1	133	50	1	1	2	64
TipoGasto	Catálogo	20	64	64	24	76	1	191	50	1	1	30	64
Imágenes	Catálogo	150	48	48	10	90	1	26	50	1	2	500	150
MateriaPrima	Catálogo	100	8	8	22	78	1	88	50	1	2	300	24
MedidasUnitarias	Catálogo	10	64	64	25	75	1	196	50	1	1	10	64
Municipios	Catálogo	500	64	64	35	65	1	69	50	1	1	500	64
Niveles	Catálogo	5	64	64	34	66	1	255	50	1	1	5	64
NivelEscuela	Catálogo	15	64	64	0	100	1	255	50	1	1	30	64
Proveedores	Catálogo	10	64	64	25	75	1	99	50	1	1	30	64
Sucursales	Catálogo	20	64	64	40	60	1	30	50	1	1	60	64
TallaUniforme	Catálogo	1500	40	40	0	100	1	255	50	1	1	3000	80
TelefonoEscuela	Catálogo	20	64	64	0	100	1	178	50	1	1	50	64
TelefonoProveedor	Catálogo	10	64	64	0	100	1	218	50	1	1	30	64
TipoPersona	Catálogo	2	64	64	33	67	1	255	50	1	1	2	64
TiposMP	Catálogo	20	64	64	32	68	1	126	50	1	1	30	64
TipoUniforme	Catálogo	20	64	64	13	87	1	201	50	1	1	30	64
Uniformes	Catálogo	150	64	64	0	100	1	224	50	1	1	300	64
CompraMP	Volátil	1000	24	24	0	100	1	255	50	1	4	10000	240
Compras	Volátil	100	64	64	0	100	1	255	50	1	1	1000	64
Gastos	Volátil	500	8	8	0	100	1	255	50	1	5	5000	120
InventarioMP	Volátil	100	64	64	14	86	1	218	50	1	1	300	64
InventarioUniforme	Volátil	1500	48	48	0	100	1	255	50	1	1	3000	100
MaterialUniforme	Volátil	15000	440	440	0	100	1	255	50	1	1	30000	880
Pagos	Volátil	1000	24	24	0	100	1	255	50	1	5	10000	256
Pedidos	Volátil	600	168	168	37	63	1	29	50	1	4	6000	1680
Personas	Volátil	2000	528	528	37	63	1	30	50	1	5	20000	5304
Preguntas	Volátil	500	168	168	31	69	1	24	50	1	5	5000	3232
Producciones	Volátil	100	64	64	0	100	1	255	50	1	1	1000	64
ProduccionUniforme	Volátil	2000	64	64	0	100	1	255	50	1	4	20000	640
Respuestas	Volátil	500	168	168	61	39	1	24	50	1	5	5000	1680
TelefonoPersona	Volátil	2000	72	72	28	72	1	218	50	1	2	20000	736
UniformePedido	Volátil	3000	80	80	0	100	1	255	50	1	4	30000	800
Ventas	Volátil	2000	40	40	0	100	1	255	50	1	3	20000	408
VentaUniforme	Volátil	10000	288	288	0	100	1	255	50	1	5	100000	2856
Total		43956	3496									286016	20466

Total de datos:	20466
+ Índices (+20%)	4093.2
+ Δ En incrementos de LOG	7367.76
	32 M

* Valores Aproximados

** Estimación a 10 años

Tabla 1.1: Tamaño total de la Base de Datos

1.2 Especificaciones Técnicas

1.2.1 La interfaz de usuario

Se sabe que el ser humano tiende a reaccionar de manera diferente ante el estímulo propiciado por ciertos colores, debido a que estos forman parte del espectro lumínico y su energía tiende a afectarle directamente (ya sea positiva o negativamente).

Un error muy frecuente en el diseño de interfaces gráficas, es el uso erróneo de los colores para el mensaje que se desea transmitir y para el tipo de personas al que va dirigido. Por tal motivo, resulta conveniente plantear qué se quiere transmitir, las sensaciones que se desean estimular, el comportamiento que se esperaría obtener y el tipo de personas que usarán el sistema.

– El mensaje a transmitir:

La imagen de una empresa pequeña pero consolidada, que cuenta con experiencia y prestigio en la zona, conocimiento en el ramo, comprometida con la calidad y sus clientes y en búsqueda de expansión.

– Estímulos y comportamientos:

Que el cliente se sienta cómodo navegando por las opciones de la página, para que su atención esté enfocada en ella y no sienta la necesidad de cerrarla a los pocos minutos de haber ingresado.

De los trabajadores, se busca estimular el uso del sistema, pues en un principio podrían mostrar cierta aversión contra este. De manera que, al igual que con los clientes, se requiere que mientras hagan uso de este, lo encuentren atractivo, cómodo y fácil de usar.

– Hacía quién va dirigido:

No hay una edad específica en el uso de la aplicación. La página de internet será vista por niños, adolescentes, padres y quizá abuelos. El sistema de administración será usado por trabajadores (hombres y mujeres) de edad entre 20 y 60 años.

1.2.2 El color

Antes de hacer la elección de los colores, es indispensable entender el mensaje que cada uno de estos envía y cómo serán interpretados, con la finalidad de atraer aquellos que tendrán impactos positivas y evitar los que no.

- Blanco:

  Es el que posee la mayor sensibilidad frente a la luz por surgir de la unión de todos los colores. Su uso hace resaltar las tonalidades que se encuentren adyacentes a él.

  Siempre tendrá una connotación positiva pues se asocia con el optimismo, la paz, la felicidad, la simplicidad, la modestia y los buenos valores.

- Negro:

  Es la ausencia de todo color, no refleja ni emite luz. Es recomendable para catálogos on-line debido a que ayuda a resaltar los colores de las imágenes. Su uso aporta elegancia, paz, fortaleza, nobleza y ayuda a aumentar la profundidad de lo que se está viendo. Su efecto adelgazante en personas también se ve reflejado en una página saturada de elementos, ayudando a reducir la perspectiva de sobrecarga de estos.

  El exceso del negro produce un efecto intimidatorio y distante.

- Amarillo:

  Es el color más luminoso del espectro. Su uso moderado estimula la felicidad, la imaginación, la energía, la memoria y el apetito.

  En combinación con el negro, hace que la atención se dirija de inmediato al amarillo, por lo que es de gran utilidad para anuncios importantes que requieran atención.

  En su tono brillante, representa la juventud y espontaneidad de las cosas dejando detrás la formalidad y estabilidad. En un tono pálido representa envidia, celos, cobardía, debilidad y enfermedad.

  El exceso de amarillo produce agotamiento e irritación.

- Rojo:

  Ayuda a estimular el metabolismo del cuerpo humano, aumenta la presión sanguínea y el ritmo cardiaco.

Resalta el texto y las imágenes, poniéndolas en primer plano y ayudando a que las tomas de decisiones se lleven de manera rápida.

Por sí solo, representa fuerza, poder, vitalidad, agresividad, impulso y pasión. En combinación con el negro, estimula la imaginación, representa dominio y tiranía, combinándolo con blanco, representa inocencia y juventud.

Su uso en exceso provoca ansiedad, agitación y tensión.

- Naranja:

  Nace de la combinación del amarillo y el rojo. Produce mayor oxigenación en el cerebro por lo que estimula la actividad mental así como también la felicidad, la determinación, el apetito y una sensación acogedora (contraria al rojo).

  Es muy utilizado para la comunicación con la gente joven y para resaltar asuntos importantes dentro de una página.

  El naranja brillante representa amistad, amabilidad, carácter positivo y estimulante. En un tono más oscuro, o en combinación con el negro, representa engaño y desconfianza.

  Utilizar el naranja en exceso podría aumentar la ansiedad.

- Azul:

  Debido a que no se encuentra en los alimentos de la naturaleza, este color es un supresor del apetito, retarda el metabolismo y produce paz, tranquilidad, armonía, serenidad, estabilidad emocional y calma.

  En general, representa la elegancia, la verdad y la responsabilidad, en un tono claro da la sensación de frescura y frío, en su tono oscuro simboliza la sabiduría e inteligencia, mezclado con blanco la pureza, en combinación con el negro, la desesperación y con un color cálido provoca atención.

  El azul en exceso puede provocar depresión, tristeza y pasividad.

- Verde:

  Es el color al que más acostumbrado está el ojo humano por ser el que más se encuentra en la naturaleza, es beneficioso para el sistema nervioso, estimula la empatía y ayuda a equilibrar las emociones.

Representa la inexperiencia, el dinero y la seguridad, cuando en él predomina el amarillo se asocia con la cobardía, discordia y enfermedad, cuando predomina el azul se relaciona con la salud emocional, la protección y la sofisticación.
Su uso en exceso podría provocar baja de energía.

- Púrpura:
  Nace de la combinación del rojo y el azul. Produce sentimientos nostálgicos, serenidad y ayuda a los problemas nerviosos y mentales.
  Su uso va dirigido a niños, adolescentes y mujeres.
  Representa fantasía, dignidad, misterio, lucidez, templanza y experiencia.
  Abusar del uso del púrpura podría provocar dolor de cabeza y tristeza.

- Gris:
  El gris, por sí mismo, no es un color, sino una transición negro-blanco.
  Representa el éxito, la estabilidad, el prestigio y la tenacidad.

1.2.3 Diseño

El hecho de que un sistema (tanto web como de escritorio) desempeñe adecuadamente las funciones para el que fue programado, no es suficiente para garantizar el éxito de este. Un aspecto muy importante a considerar, es la interfaz gráfica (GUI por sus siglas en inglés *Graphical User Interface*) con la que el usuario va a interactuar.

Hoy en día, las GUI cada vez son más complejas y atractivas. Por lo que el programador se ve en la tarea de implementar varias funcionalidades gráficas en poco tiempo y si este no tiene un control adecuado sobre estas, el mantenimiento y correcciones pueden convertirse en un verdadero problema.

Para la GUI del sistema se utilizó CSS, una tecnología aplicada al HTML que la mayoría de las páginas actuales usan, JQuery que es una librería basada en Javascript para minimizar los trabajos de programación y JSP, que es el lenguaje de servidor soportado por J2EE (Java 2 Enterprise Edition).

1.3 CSS

CSS (Cascade Style Sheet) u hojas de estilo en cascada, es un lenguaje especificado por el W3C (World Wide Web Consortium) como un estándar para los navegadores web. Es utilizado para separar los contenidos de la página con la manera en que estos serán presentados.

El uso de hojas de estilos puede hacerse de tres maneras diferentes:

1. Estilo en línea: Aplicar el estilo directamente en la etiqueta en la que se requiera
2. Hoja de Estilo Interna: Se usa dentro de la etiqueta `<head>` y las reglas de estilo deben ser especificadas para cada una de las páginas que formen parte del sistema.
3. Hoja de Estilo Externa: Se centralizan las cláusulas de los estilos en un archivo externo de extensión `.css` y se mandan a llamar desde las páginas correspondientes.

La última opción es la que supone una mejor práctica para el uso de CSS puesto que de esta forma es como realmente se está aplicando la separación del contenido y la presentación concentrando los estilos en un archivo externo.

1.3.1 Estilos y Etiquetas

Una de las grandes ventajas de CSS es el ahorro y reutilización de código, puesto que una etiqueta HTML puede ser personalizada para que a lo largo del sistema sea usada con la misma definición evitando repetición de código y haciendo más fácil las tareas de mantenimiento.

Por ejemplo: Suponiendo que se desea tener de fondo una imagen llamada “logo.png” para todas las páginas que componen el sistema. Sin el uso de hojas de estilo. Esto quedaría de la siguiente manera:

```
1   <body background=”logo.png”>
2       […]
3   </body>
```

Código 1.1: Imagen de fondo en página HTML sin CSS.

Lo anterior funciona perfectamente, sin embargo, si el sistema cuenta con un número extenso de páginas, el atributo `background` deberá ser especificado en cada una de ellas, y si por requerimientos del sistema, se pide que cada cierto tiempo esta imagen cambie por otras, con

otro nombre y/u otra extensión, la actualización puede ser bastante tardada y tediosa. Es por ello, que CSS entra en juego para evitar este tipo de problemas.

Lo que se hace, es personalizar la etiqueta <body> por medio de una hoja de estilo, en la cual se especificarán los atributos correspondientes a esta:

```
1   body{
2        background-image:url(logo.png);
3   }
```

Código 1.2: Imagen de fondo en página HTML con CSS y etiquetas.

```
1    <head>
2       <link rel="stylesheet" href="estilos.css" type="text/css" />
3    </head>
4    <body>
5       […]
6    </body>
```

Código 1.2 (*continuación*)

Como se puede notar, cualquier cambio que se tenga que hacer a la imagen del fondo, bastaría con alterar la definición de la etiqueta <body> hecha en el archivo `estilos.css` para que en cuestión de segundos la modificación sea terminada.

Para mandar a llamar la hoja de estilo externa, se coloca la etiqueta <link> con sus atributos:

- `rel`: Muestra la relación que hay entre la página y el archivo externo
- `href`: Indica la ubicación del archivo .css
- `type`: Especifica qué tipo de archivo se está llamando

1.3.2 Estilos y Clases

Hay ocasiones en que la personalización de etiquetas no puede ser posible debido a que no sería viable que en todo el sistema una misma etiqueta tuviera el mismo comportamiento cuando sea usada, tal es el caso de `<font>`. No obstante, se pueden agrupar ciertas características compartidas y definirlas en una clase.

Por ejemplo: Si el sistema requiere que los títulos sean tamaño grande, color rojo y letra tipo "Arial", los subtítulos, tamaño mediano, color azul y letra tipo "Verdana", y el resto del texto, tamaño pequeño, color negro y letra tipo "Calibri". Sin utilizar hojas de estilo quedaría de la siguiente manera:

```
<body>
<font face=”Arial” color=”#FF0000“ size=”6”>Título</font>
<font face=”Verdana” color=”#0000FF“ size=”4”>Subtítulo</font>
<font face=”Calibri” color=”#000000“ size=”2”>Texto</font>
</body>
```

Código 1.3: Tipos de letra en página HTML sin CSS.

Al igual que en el ejemplo del punto anterior, la repetición de código en cada página es necesaria, por lo que es bastante ineficiente programar de esta manera. La solución de CSS consistiría en separar en clases las diferentes definiciones de las letras.

```
1    .Titulos{
2        font-family:Arial;
3         font-size:16px;
4         color:#FF0000;
5    }
6    .Subtitulos{
7         font-family:Verdana;
8         font-size:12px;
9         color:#0000FF;
10   }
```

Código 1.4: Tipos de letra en página HTML con CSS y clases.

```
11   .Texto{
12        font-family:Calibri;
13        font-size:8px;
14        color:#000000;
15   }
```

Código 1.4 (*continuación*)

```
1    <body>
2        <div class="Titulos">Título</div>
3        <div class="Subtitulos">Subtítulo</div>
4        <div class="Texto">Texto</div>
5    </body>
```

Código 1.4 (*continuación*)

Para indicar que se trata de una clase, se antecede con un punto (.) al nombre de esta, y posteriormente se declara su definición.

La etiqueta `<div>` permite agrupar en bloques las clases que se están llamando para su separación.

1.3.3 Estilos e Id's

La declaración de estilos por medio de su id, es bastante similar a la anterior, con la diferencia de que estos sólo podrán ser utilizados una vez en toda la página.

Una de sus principales aplicaciones es para posicionar los objetos en la GUI de manera que resulte más sencillo que con el uso de tablas, pues en vez de ello, se hace uso de capas que se declaran en la hoja de estilo por medio de un id (único para cada capa).

```
1     #Capa1 {
2          position:absolute;
3          left:313px;
4          top:49px;
5          width:256px;
6          height:233px;
7     }
8     #Capa2 {
9          position:absolute;
10         left:318px;
11         top:57px;
12         width:115px;
13         height:219px;
14     }
15     #Capa3 {
16         position:absolute;
17         left:440px;
18         top:56px;
19         width:122px;
20         height:220px;
21      }
```

Código 1.5: Capas en página HTML con CSS e ids.

```
1     <body>
2         <div id="Capa1">[…]</div>
3         <div id="Capa2">[…]</div>
4         <div id="Capa3">[…]</div>
5     </body>
```

Código 1.5 (*continuación*)

Para indicar que la declaración se hace a través de su id, se antecede el símbolo número (#) al nombre del id y posteriormente se declaran los atributos relativos a la posición de la capa, tales como largo, ancho, posición en la pantalla, color de fondo, imagen de fondo, márgenes, bordes, etcétera.

1.4 jQuery

jQuery es una librería de código abierto desarrollada en 2006 para facilitar la programación visual y funcional del lado del cliente. Es muy liviana y al estar basada en javascript, permite implementar en su totalidad las funcionalidades de este último, pero con las siguientes ventajas:

- Ahorro de líneas de código.
- Curva de aprendizaje bastante reducida en comparación con javascript.
- Integración sencilla con Ajax.
- Ahorro en tiempo de desarrollo y mantenimiento.
- Compatibilidad de navegadores.
- Fácil manejo con CSS.

1.4.1 Funciones de jQuery

La sintaxis que maneja jQuery es bastante sencilla y fácil de aplicar:

```
$(elemento).evento(función(){
   […]
});
```

Por medio de su función principal `$()` se seleccionan los elementos de la página, estos pueden ser por clase, por identificador o por etiquetas para ser aplicados a algún método de conveniencia. Las funciones que ofrece jQuery pueden clasificarse de acuerdo a su uso:

- **Manipulación de los elementos:**

Función	Descripción
.addClass(clase)	Permite agregar una clase al elemento seleccionado.
.attr(atributo) .attr(atributo,valor)	Obtiene el valor del atributo marcado en el primer parámetro. En caso de ser especificado el segundo, actualiza el atributo con el valor indicado.

Tabla 1.2: Funciones principales de jQuery para la manipulación de elementos en una página.

Función	Descripción
.css(propiedad) .css(propiedad,valor)	Obtiene el valor de la propiedad CSS especificada en el primer parámetro. En caso de ser especificado el segundo, actualiza la propiedad con el valor indicado.
.html() .html(codigoHTML)	Obtiene el código HTML del elemento seleccionado. De manera adicional recibe un argumento en código HTML, que de ser especificado, será agregado al elemento.
.removeAttr(atributo)	Elimina el atributo especificado de los elementos seleccionados.
.removeClass(clase)	Elimina la clase especificada de los elementos seleccionados.

Tabla 1.2 (*continuación*)

- **Efectos Visuales:**

Función	Descripción
.fadeIn(velocidad)	Modifica la transparencia del elemento desde 0. La velocidad debe ser indicada con "fast", "slow" o "normal".
.fadeOut(velocidad)	Modifica la transparencia del elemento desde el nivel actual hasta 0.
.fadeTo(velocidad, transparencia)	Modifica la transparencia del elemento desde el nivel actual hasta el indicado (comprendido entre 0 y 1).
.hide()	Oculta el elemento seleccionado.
.show()	Muestra el elemento seleccionado.

Tabla 1.3: Funciones principales de jQuery para dar efectos visuales a los elementos de una página.

- **Eventos:**

Función	Descripción
.blur(), .change(), .click(), .dblclick(), .focus(), .keydown(), .keypress(), .keyup(), .mousedown(), .mouseenter(), .mouseleave() ,.mousemove(), .mouseout(), .mouseover(), .mouseup(),	jQuery ofrece estas funciones que permiten la captura de un evento para ejecutar cierta acción.
.hover(mouseEntra1,mouseSale2)	Engloba en un mismo evento cuando el mouse entra y sale del área del elemento seleccionado y las acciones a realizar en cada caso.
.ready()	Asegura que los elementos de la página se encuentren listos para ser manipulados.

Tabla 1.4: Funciones principales de jQuery para el manejo de eventos.

- **Navegación a través de los elementos:**

Función	Descripción
. children()	Obtiene los elementos hijos del seleccionado.
.each()	Itera sobre los objetos jQuery para ejecutar cierta acción en cada uno de estos.
.parent()	Obtiene el elemento padre del seleccionado.

Tabla 1.5: Funciones principales de jQuery para la navegación de elementos de una página.

1.4.2 Ajax (Asynchronous JavaScript and XML)

La tendencia actual de desarrollo web está encaminada al uso de Ajax para mejorar la eficiencia y vista de los sistemas haciéndolos más interactivos. Su principal aplicación es actualizar fragmentos de página sin tener que ser recargada completamente. jQuery ofrece distintos métodos para la integración de Ajax a la aplicación:

1.4.2.1 load()

Es la función más simple de ajax, encargada de mostrar una página en la capa indicada. Su uso se ejemplifica a continuación:

```
1     $(".links").click(function(){
2         var pagina=$(this).attr("href");
3         $("#Centro").load(pagina);
4         return false;
5     });
```

Código 1.6: Funcionamiento de `load()`.

Cuando se da clic en cualquier vínculo perteneciente a la clase `links`, con ayuda de la función `attr()` se guarda en la variable `pagina` el atributo `href` del link seleccionado, para posteriormente cargar la respuesta del servidor con la función `load()` en la capa `Centro` (previamente definida en CSS). Finalmente se regresa `false`, para que la página requerida no sea redirigida sino que sea cargada en la capa indicada.

1.4.2.2 ajax()

Aunque muy similar a la función anterior, `ajax()` es más completa y permite la configuración de parámetros extra para poder manipular datos del usuario. Por lo anterior, se aplica para enviar información de formularios y ser procesados. Sus parámetros principales se muestran a continuación:

Parámetro	Descripción
async	Permite indicar que la carga se haga de manera síncrona o asíncrona. Es recomendable que se haga de manera asíncrona para que no interfiera con la carga página.
data	Especifica los datos del formulario que serán enviados al servidor.
error	En caso de que ocurra un error, se ejecutará y regresará los detalles de este.

Tabla 1.6: Parámetros de `ajax()`.

Parámetro	Descripción
success	Una vez que la ejecución haya sido exitosa, se envían los datos al servidor y se realiza la acción corresponiente.
type	Especifica el método de envío de datos que será utilizado: POST o GET
url	Indica la ruta que procesará los datos del formulario.

Tabla 1.6 (*continuación*)

Un ejemplo de su uso para el procesamiento de formularios se muestra a continuación:

```
1      $("form").submit(function(){
2          $.ajax({
3              type:"POST",
4              async:true,
5              url:$(this).attr("action"),
6              data:$(this).serialize(),
7              success: function(data){
8                  $("#Centro").html(data);},
9              error:function(){
10                 alert("Hubo un error");}
9          });
10          return false;
11      });
```

Código 1.7: Funcionamiento de `ajax()`.

La llamada a `ajax()` comienza una vez que se ha dado `submit` al formulario. En la línea 6, `serialize` se encarga de concatenar los datos en una cadena del tipo POST o GET para ser procesada por la URL indicada. Si la ejecución de `ajax()` se hizo de manera correcta, se llama al parámetro `success` y se envían los datos recogidos en el formulario para que el resultado del procesamiento se muestre en la capa que se indica, en este caso Centro.

1.4.2.3post() / get()

Estas funciones permiten lanzar peticiones del tipo POST o GET al servidor, según se requiera. Son especialmente útiles para el llenado dinámico de combos dependientes. Reciben tres parámetros: La URL que se encargará de procesar la petición, los datos que serán enviados al servidor con la forma {nombre : valor} y una función, llamada de callback, que será invocada una vez que termine el procesamiento. El siguiente ejemplo muestra un combo que contiene los estados del país y al momento que este cambia, se imprimen en un segundo combo los municipios del estado seleccionado.

```
1     $("#estados").change(function(){
2         var estado=$("#estados").attr("value");
3         $.post("/URLDeProcesamiento",
4             {idestado: estado},
5              function(data){
6             $("#municipios").html(data);
7         });
8     });
```

Código 1.8: Funcionamiento de `post()`.

2

Java y el mundo Web

2.1 El protocolo HTTP

En la actualidad, difícilmente se imagina la vida sin el uso de internet: Comprar un boleto para un concierto, enviar correo electrónico, dejar un mensaje en el blog que seguimos, revisar el estado de cuenta bancario, son actividades cotidianas y transparentes para la persona promedio del siglo XXI. Sin embargo, detrás de estas existe un mundo virtual llamado *World Wide Web*, que está formado por una inmensa cantidad de equipos llamados servidores, que proveen recursos específicos a otros tantos, llamados clientes.

Para que la comunicación entre cliente y servidor se lleve a cabo, se requiere de un lenguaje que ambos puedan interpretar. Este lenguaje en común, HTTP (*HyperText Transfer Protocol*), fue definido por el W3C en 1990 y se trata de un protocolo de petición/respuesta entre cliente y servidor por medio de alguno de sus métodos.

La comunicación comienza cuando el cliente lanza una petición que será atendida por el servidor. El cliente recibe como respuesta los datos solicitados, así como un código que indica el procesamiento de dicha petición.

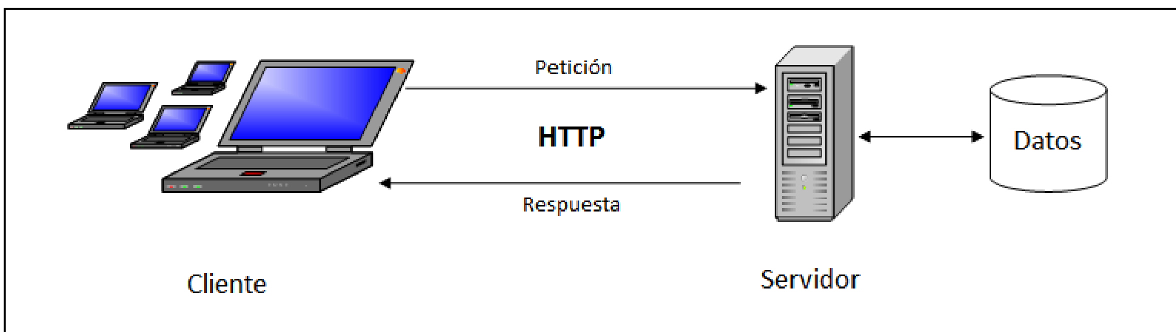


Figura 2.1: Funcionamiento básico del protocolo HTTP

2.1.1 Solicitud de la petición

Cada petición hecha por el cliente debe contener la información necesaria para poder ser tratada. Esto es, a qué servidor va dirigida y qué recurso es el que la atenderá. Dicha información se encuentra en la dirección URL:

[protocolo]://[servidor]:[puerto]/[recurso]

Ejemplos:

http://localhost:8084/Uniformes/paginasProduccion/Inicio.jsp

http://www.oracle.com/index.html

https[1]://boveda.banamex.com.mx/serban/index.htm

El servidor puede ser expresado por su nombre canónico, su dirección IP o el nombre del equipo en red. En caso de que se omita el puerto, se conecta al asignado por defecto según el protocolo utilizado. Puerto 80 para http y 443 para https.

2.1.2 Métodos de petición

Cada petición enviada al servidor debe ser atendida por alguno de los métodos explicados a continuación:

1. GET: Es el método que el navegador elige por default cuando se hace clic en algún enlace o se escribe directamente la URL. Generalmente es utilizado para *obtener* información de sólo lectura del servidor. Para encaminar la petición, la URL debe contener los parámetros que se desean enviar. Por ejemplo */recurso.jsp?param1=1¶m2=2*
2. POST: Empleado cuando se requiere enviar información, comúnmente proveniente de un formulario para actualizar los datos del servidor (en una base de datos). La información del cliente es enviada por medio del cuerpo de la petición y no por la URL como es el caso del método GET.
3. PUT: Se utiliza normalmente para almacenar un archivo en el servidor.
4. DELETE: Es la contraparte de PUT, y es utilizado para eliminar un recurso. En algunos servidores, tanto PUT como DELETE, no se encuentran disponibles debido a cuestiones de seguridad.
5. HEAD: Es muy similar al método GET, sin embargo, este sólo regresa las cabeceras de la respuesta y no su cuerpo.

1. https (*http secure*): Es una combinación del protocolo http con protocolos criptográficos para obtener peticiones más seguras.

6. OPTIONS: Devuelve al cliente los diferentes métodos HTTP que pueden ser ejecutados por el servidor. Por lo general: GET, POST y HEAD.
7. TRACE: Este método se utiliza para saber si existe el receptor del mensaje y usar dicha información para tareas de diagnóstico y depuración.

Por lo general, GET y POST son los métodos más empleados por el servidor y la elección de estos debe depender del tipo de petición que se haga, teniendo siempre en cuenta que con el uso de GET, los datos enviados al servidor serán visibles por medio de la URL, mientras que POST los esconde en el cuerpo de la petición haciendo que viajen de manera más segura.

2.2 La evolución de Java en el mundo web

La primera aportación que hizo Java al mundo web, en el año 1995, fue el uso de los applets, programas hechos bajo la tecnología Java, que pueden ser embebidos en una página HTML y ejecutados por la Máquina Virtual de Java (JVM por sus siglas en inglés) del navegador. Pese a tratarse de un recurso muy útil y novedoso, no estaba especificado propiamente para el desarrollo de aplicaciones web y se encontraba condicionado al soporte que cada navegador diera a esta tecnología.

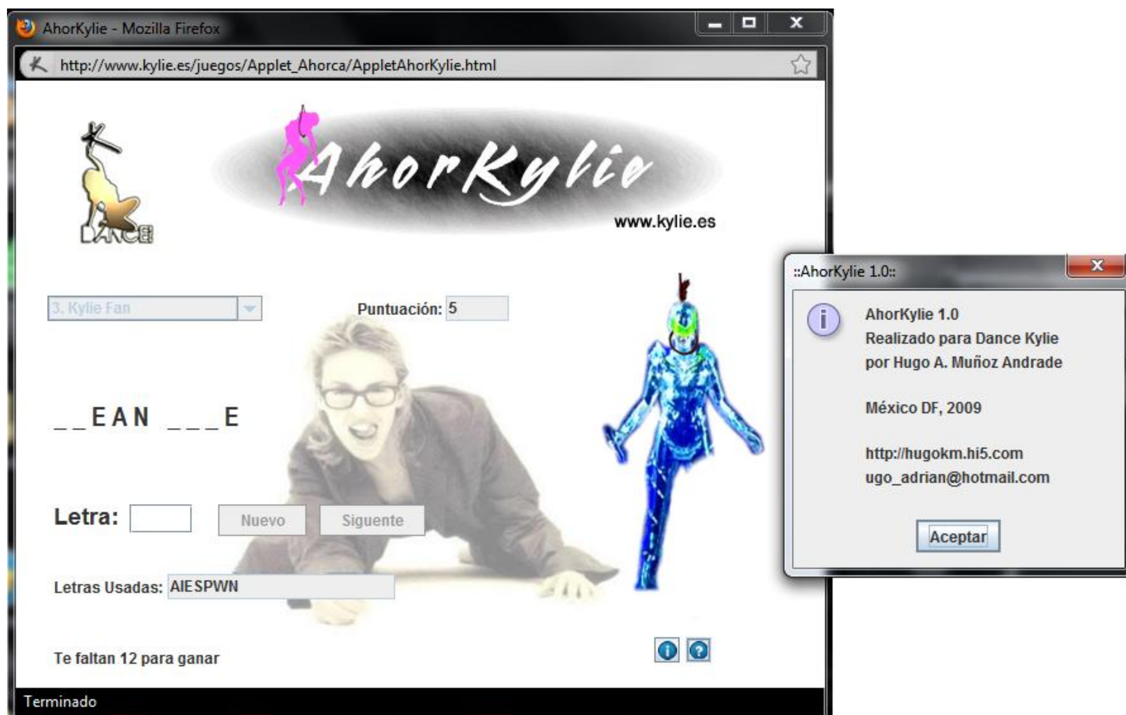


Figura 2.2: Juego hecho con la tecnología applet de Java y embebido en una página web

Es así como en 1997, Sun Microsystems, libera la primer especificación de los servlets como una alternativa de desarrollo web, cuyo mercado era mayormente dominado por lenguajes como PHP, ASP y Perl.

2.2.1 Servlets

Un servlet es una clase que tiene como función principal atender peticiones hechas por el navegador, procesarlas del lado del servidor y regresar una respuesta al cliente, la cuál puede ser un documento HTML, XML u otro formato como imágenes o datos binarios. Para que un servlet pueda funcionar, debe poder entender las peticiones, por lo que debe usar el protocolo HTTP.

2.2.1.1 Funcionamiento de los Servlets

Un servlet está compuesto principalmente de 3 métodos: `init()`, `service()` y `destroy()`. Su ciclo de vida comienza cuando es instanciado y cargado en memoria por el contendedor[2] haciendo una llamada a su método principal `init()` tras la primer petición recibida para este. El método `init()` será ejecutado sólo una vez durante el ciclo de vida, por lo que existirá únicamente una copia en memoria de cada servlet. Por cada petición interceptada, el contenedor creará un hilo de ejecución, transparente para el programador, llamando al método `service()`, que se encargará de procesarla y enviar su respuesta al cliente. Finalmente, cuando el servlet termina, el contenedor ejecuta el método `destroy()` para limpiar la memoria.

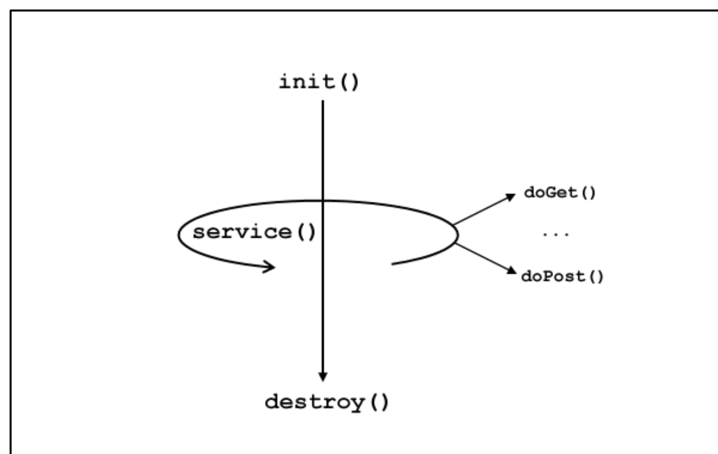


Figura 2.3: Funcionamiento del servlet.

2. El contenedor de servlets es el servidor encargado de ejecutar los servlets.

2.2.1.2 El API (*Application Programming Interface*) de los Servlets

Java tiene una extensa propuesta de clases e interfaces que forman el API completa de los servlets. Quizá la más importante sea la interface `Servlet`, que define los métodos que deben ser implementados por todos los servlets.
La implementación de aquellos cuyo protocolo sea independiente, se logra heredando a la clase `GenericServlet` por lo que el método `service()` debe ser sobreescrito para manejar apropiadamente la petición.

Por otro lado, los servlets que usen el protocolo HTTP deben heredar su funcionalidad a la clase `HttpServlet`, subclase de `GenericServlet`, pero con funciones específicas del protocolo, en cuyos métodos se encuentran definidos los 7 tipos de petición HTTP: `doGet()`, `doPost()`, `doDelete()`, `doHead()`, `doOptions()`, `doPut()` y `doTrace()`. En este caso, `service()` se encarga de llamar al método `doXxx()` apropiado, el cual debe ser sobreescrito para el manejo de la petición. Estos métodos, reciben objetos `HttpServletRequest` y `HttpServletResponse`, que contienen la petición del cliente y la respuesta del servidor, respectivamente.

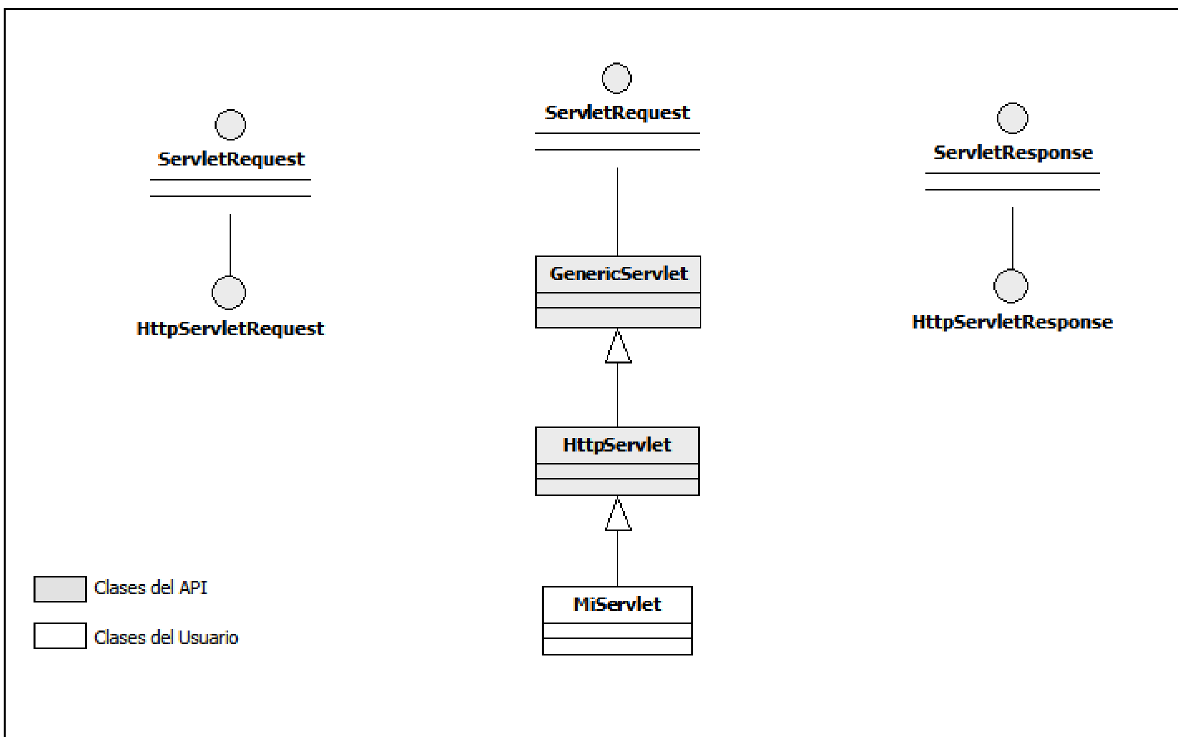


Figura 2.4: API general de los Servlets

En las siguientes tablas se muestra la descripción de los métodos principales de las clases e interfaces del API.

Método	Retorno	Descripción
destroy()	void	Llamado por el contenedor para indicar que la ejecución del servlet ha terminado.
getServletConfig()	ServletConfig	Proporciona el acceso a la información de configuración del servlet.
getServletInfo()	String	Devuelve información del servlet, como el autor y la versión.
init(ServletConfig config)	void	Llamado por el contenedor para crear la instancia del servlet.
service(ServletRequest peticion, ServletResponse respuesta)		Llamado por el contenedor para responder a una petición del cliente.

Tabla 2.1: Métodos de la interface `Servlet`

Método	Retorno	Descripción
getInitParameter(String nombre)	String	Regresa el nombre del ServletConfig de inicialización.
getInitParameterNames()	Enumeration	Regresa los nombres de los parámetros de inicialización.
getServletContext()	ServletContext	Regresa la referencia del ServletContext sobre el que el servlet está corriendo.
getServletName()	String	Regresa el nombre de la instancia del servlet.
log(String mensaje)	void	Escribe en una bitácora del servlet el mensaje indicado.
log(String mensaje, Throwable t)		Escribe en una bitácora del servlet el mensaje de error en caso de excepción Throwable.

Tabla 2.2: Métodos de la clase abstracta `GenericServlet` (implementa interface `Servlet`)

Método	Retorno	Descripción
doXxx(HttpServletRequest pet, HttpServletResponse resp)	void	Se llama en respuesta a una de los 7 tipos de petición de HTTP (siendo GET y POST los más comunes).
getLastModified(HttpServletRequest pet)	long	Regresa en milisegundos el tiempo en el que el objeto pet fue modificado por última vez.

Tabla 2.3: Métodos de la clase abstracta `HttpServlet` (hereda a la clase abstracta `GenericServlet`)

Método	Retorno	Descripción
getParameter(String nombre)	String	Obtiene el valor de un parámetro, con el nombre indicado, al servlet.
getParameterNames()	Enumeration	Devuelve el nombre de todos los parámetros asociados a la petición.
getParameterValues(String nombre)	String[]	Para un parámetro con múltiples valores, devuelve cada uno de estos.
getCookies()	Cookie[]	Devuelve un arreglo de objetos Cookie almacenados en el cliente por el servidor.

Tabla 2.4: Algunos métodos de la interface `HttpServletRequest`

Método	Retorno	Descripción
addCookie	void	Agrega un objeto Cookie al encabezado de la respuesta.
getOutputStream()	ServletOutputStream	Obtiene un flujo de salida en bytes para enviar datos binarios al cliente.

Tabla 2.5: Algunos métodos de la interface `HttpServletResponse`

Método	Retorno	Descripción
getWriter()	PrintWriter	Obtiene un flujo de salida en carácteres para enviar datos de texto al cliente.
setContentType(String tipo)	void	Especifica el tipo MIME[3] de la respuesta para mostrarlo adecuadamente en el navegador.

Tabla 2.5 (*continuación*)

El servlet que se presenta a continuación recoge los valores de *nombre* y *edad* del cliente, el servidor lo procesa y envía como respuesta una página en formato HTML con los datos recogidos.

```
1   public class ServletNombreEdad extends HttpServlet {
2       public void doPost(HttpServletRequest request,
3               HttpServletResponse response)
4               throws ServletException, IOException {
5
6           PrintWriter out = response.getWriter();
7           response.setContentType("text/html");
8           String nombre = request.getParameter("nombre");
9           String edad = request.getParameter("edad");
10          out.println("<html>");
11          out.println("<head>");
12          out.println("<title>Servlet de Prueba</title>");
13          out.println("</head>");
14          out.println("<body>");
15          out.println("<br />");
16          out.println("<h1>"+nombre);
17          out.println("<br />usted tiene "+edad+" años </h1>");
18          out.println("</body>");
19          out.println("</html>");
20      }
21   }
```

Código 2.1: Servlet que procesa una petición del cliente

3. MIME (Multipurpose Internet Mail Extensions): Es un estándar que ayuda al navegador a comprender y manejar contenidos, por ejemplo: image/jpeg, text/html, video/mpeg.

En la línea 2 y 3 se indica que se trabajará con una petición de tipo POST y se lleva a cabo la sobreescritura del método `doPost()` recibiendo los objetos `request` y `response`. Se crea un objeto `PrintWriter` en la línea 6 para poder mandar caracteres al cliente y en la línea 7 se define el tipo MIME de la respuesta, en este caso un documento de texto HTML. El método `getParameter()` es el encargado de recoger los datos de la petición y en las líneas 8 y 9 se almacenan en su respectiva variable. Finalmente, de las líneas 10 a 19, se construye la respuesta que será enviada al cliente en forma de HTML.

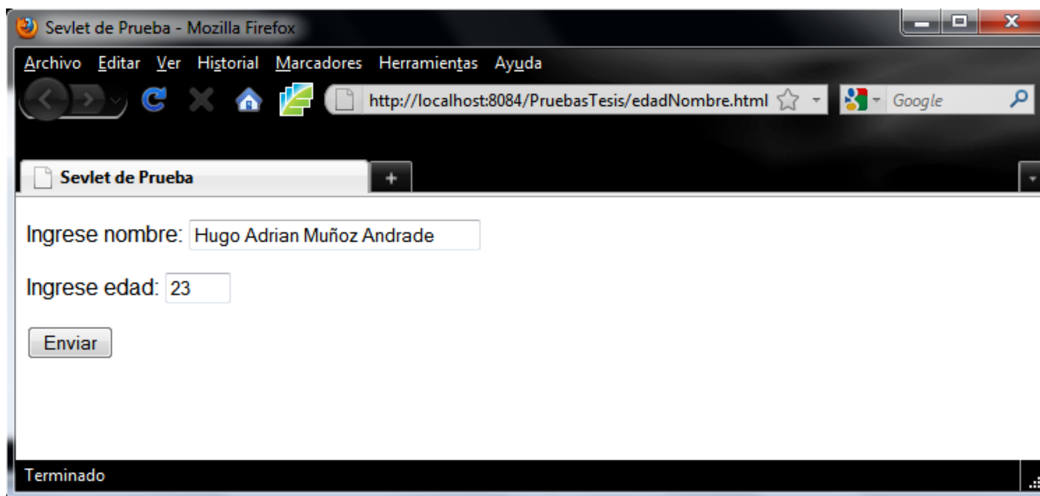


Figura 2.5: Petición de cliente

Figura 2.6: Respuesta del servidor

Esta primer solución de *Sun* para desarrollar aplicaciones web en Java no fue aceptada con facilidad debido al desconocimiento de la programación orientada a objetos y a la complicada manera de construir páginas sencillas por la inclusión de etiquetas HTML en el código del servlet. Las demás tecnologías disponibles, mencionadas con anterioridad, se encontraban basadas en soluciones tipo script y ofrecían un desarrollo más rápido aunque no necesariamente eficiente. Dada esta situación, se introduce JSP (*Java Server Pages*) que facilitaría la creación de contenido dinámico del lado del servidor.

2.2.2 Java Server Pages (JSP)

En sus inicios fue fuertemente criticado por los programadores debido a las similitudes que presentaba con otros lenguajes que hacían uso de scripts, y aunque a primera vista así parecía, JSP era más que eso, se trataba de una extensión de la tecnología servlet. El contenedor transforma cada JSP en un servlet que compila y procesa cada petición recibida.

2.2.2.1 El uso de scriptlets

Aunque el trabajo se reducía considerablemente, la solución que JSP daba a los problemas tenía ciertas debilidades, como piezas de código Java embebido en forma de *scriptlets*, que provocaba desorden, complejidad en la JSP y no se promovía su reutilización. A continuación se muestra una JSP que resuelve el problema previamente hecho en el servlet:

```
1   <html>
2       <head>
3           <title>JSP Prueba con Sriptlets</title>
4       </head>
5       <body>
6           <%
7               String nombre = request.getParameter("nombre");
8               String edad = request.getParameter("edad");
9           %>
10           <h1>Hola <%=nombre%><br />
11               usted tiene <%=edad%> años
12           </h1>
13       </body>
14   </html>
```

Código 2.2: JSP que procesa la petición, usando scriptlets. El elemento <% %> permite la inclusión de código Java. Mientras que <%= %> inserta el valor de una variable.

En 1999 aparecen las etiquetas personalizadas que ofrecen un código más limpio, comprensible y reutilizable siguiendo la sintaxis de HTML, pero no es sino hasta el año 2002, en el que son introducidas como parte de la especificación de Sun para JSP, cuando comienza el auge de estas etiquetas.

2.2.2.2 JSTL (*JSP Standard Tag Library*) y el lenguaje EL (*Expression Language*)

La librería JSTL, creada por la *fundación Apache* y *Sun,* introduce el uso de etiquetas personalizadas que permiten el control de flujo de datos, evitando con esto, la inclusión de código Java en la JSP. De esta manera, se favorecía el aprendizaje de dicha tecnología, pues aunque no se tuviera el conocimiento de la programación orientada a objetos o del propio lenguaje, usando las etiquetas personalizadas de JSTL se aprovechaban las ventajas que Java ofrecía.

El lenguaje EL surgió como parte de la especificación JSTL para acceder de manera directa a los datos de petición y a las propiedades de los objetos. Su sintaxis básica es: ${objeto.propiedad}.

Las principales librerías que ofrece JSTL se muestran en la siguiente tabla:

Función JSTL	Prefijo	Descripción
Core	c	Se encarga de las condiciones, bucles, asignación de variables y redirecciones. Algunas etiquetas son: <c:choose>, <c:if>, <c:redirect>, <c:set>, <c:forEach>.
Format	fmt	Útil para dar formato a la información usando los patrones dados. Algunas etiquetas son: <fmt:formatDate> y <fmt:formatNumber>
SQL	sql	Maneja la conectividad con la base de datos. Algunas etiquetas son: <sql:query>, <sql:transaction>, <sql:update>.
Functions	fn	Se trata de un conjunto de funciones útiles para el acceso de datos con EL. Algunas de estas son: fn:containts(cadena, subc), fn:endsWith(cadena,sufijo), fn:strartsWith(cadena,prefijo), fn:indexOf(cadena, sub), fn:length(cadena), fm:trim(cadena)

Tabla 2.6: Funciones de JTSL. Ver apéndice A para mayor información.

El siguiente código muestra una JSP que resuelve el problema de los ejemplos anteriores usando la librería JSTL y el lenguaje EL:

```
1   <html>
2       <head>
3           <title>JSP Prueba con JSTL y EL</title>
4       </head>
5       <body>
6           <h1>Hola ${param.nombre}<br />
7               usted tiene ${param.edad} años
8           </h1>
9       </body>
10  </html>
```

Código 2.3: JSP que procesa la petición, usando la librería JSTL y lenguaje EL. El objeto param es una colección de los parámetros de la petición.

Tras esta evolución, es como los desarrolladores web comenzaron a considerar JSP como una tecnología eficaz, ágil, de fácil aplicación y potente, pues seguía conservando su metodología orientada a objetos así como el uso de los servlets, aunque ya de manera transparente para el programador. No obstante seguía habiendo un problema: El código de la lógica de negocio y de acceso a datos estaba mezclado con el de presentación en la JSP, resultando en sistemas cuyo mantenimiento era difícil y no siempre resultaba apropiado. Ante esta complicación, se vio la necesidad de separar en capas la aplicación, usando un modelo que permitiera gestionar de manera independiente el acceso a datos, la lógica de negocio y la presentación de la información al cliente.

2.3 El patrón MVC (Model, View, Controller)

MVC es un patrón de diseño en el que se separan los componentes de una aplicación (generalmente web) de acuerdo a su funcionalidad:

- Modelo: Esta capa encapsula la lógica de negocio y el acceso a datos.
- Vista: Es la representación gráfica de los datos generados en el modelo para ser enviados al usuario.

- Controlador: Dirige las peticiones del cliente hacia el modelo para su procesamiento y espera su respuesta para ser enviada a la vista.

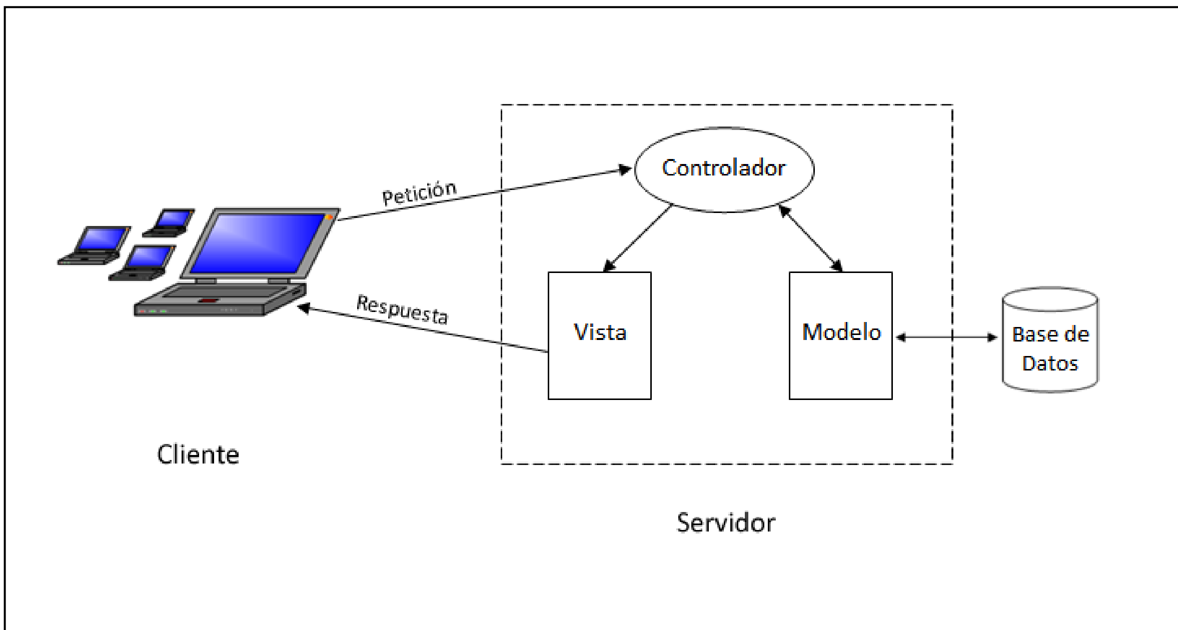


Figura 2.7: Funcionamiento básico del patrón MVC

El uso de este patrón facilita el mantenimiento, debido a que los componentes del sistema se encuentran muy bien ubicados, y por su desacoplamiento, la modificación de uno de estos no afecta al funcionamiento de los demás. A su vez, promueve la reutilización de código y aumenta la escalabilidad.

En este punto se ve con claridad cómo se pueden combinar la simplicidad que ofrece JSP para crear contenido HTML dinámico y la potencia que brinda la gestión de peticiones HTTP por parte de los servlets, para crear aplicaciones web rápidas, robustas y escalables.

A pesar de que las especificaciones de Java permiten un desarrollo ágil de sistemas web, para la construcción de esta aplicación serán utilizados tres frameworks, que en conjunto, simplifican la implementación del patrón MVC: Struts, Spring e Hibernate.

2.4 Struts

Struts es el pionero de los frameworks que implementan MVC. Su historia comienza en mayo del 2000 cuando Craig McClanahan lanza el proyecto de crear un estándar MVC basado en Java, en 2001, año en que se libera la versión 1.0, fue adoptado por buena parte de los desarrolladores a nivel mundial, siendo hasta ahora el líder de los frameworks MVC de Java. Sus beneficios directos se ven reflejados sólo en las capas de la *Vista* y el *Controlador*.

2.4.1 Componentes y funcionamiento de Struts

Struts requiere de un archivo de configuración llamado `struts-config.xml` en el que se registran los elementos del API contenidos en la aplicación.

2.4.1.1 API de Struts

- `ActionServlet`: Este servlet actúa como el cerebro de la aplicación pues se encarga de interceptar todas las peticiones recibidas del cliente delegando su tratamiento a un objeto del tipo `RequestProcessor` responsable de encaminar la petición para su respetivo proceso.

- `Action`: Si `ActionServlet` es el cerebro, la clase `Action` puede ser considerada como el corazón, pues sirve como puente de enlace entre la petición del cliente y el Modelo. Es aquí donde se hacen las llamadas a la lógica de negocio y la transferencia de datos a la Vista (generalmente páginas JSP).

- `ActionForm`: Se trata de JavaBeans que contienen los datos de captura del cliente (usualmente procedentes de formularios) y únicamente sirve para el transporte de datos, no deben ser usados por la lógica u otras capas de la aplicación.

- `ActionMapping`: Asocia una petición con algún objeto `Action` que la procesará. Contiene la información de la URL que provoca la ejecución del `Action`, así como las posibles vistas a las que serán enviados los datos de la respuesta.

– `ActionForward`: Se trata de objeto que contiene la dirección lógica de algún recurso al que será direccionado el usuario una vez que el `Action` haya terminado de ejecutarse. Esto trae una ventaja de mantenimiento, pues cuando una URL cambie por motivos de actualización, no se tendrá que modificar cada referencia con su ubicación física desde código, sino simplemente modificar en el archivo `struts-config.xml` dicha referencia.

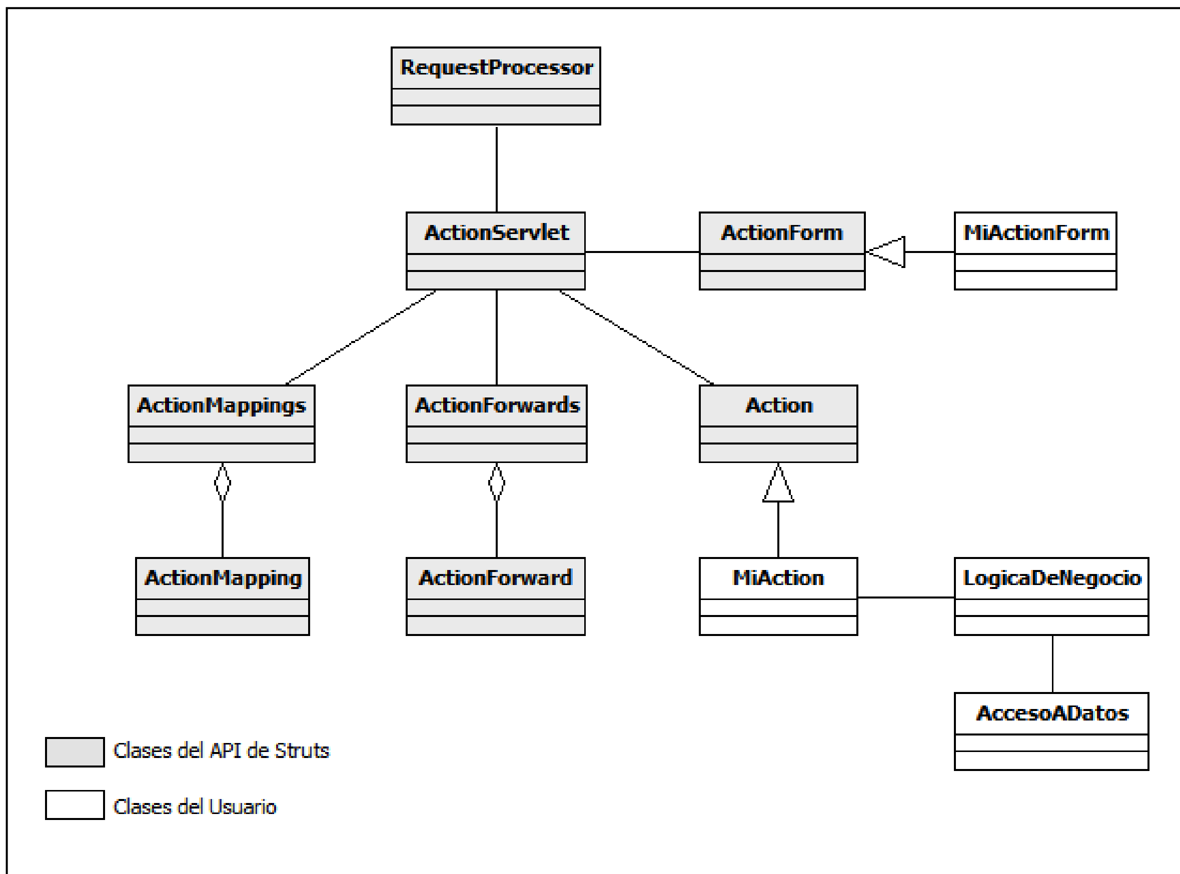


Figura 2.8: API general de Struts

2.4.1.2 Ciclo de vida de una petición en Struts

Como convención, para que una petición sea interceptada por `ActionServlet`, el recurso de la URL debe contar con la terminación `.do` (por ejemplo http://localhost:8084/Uniformes/verEmpleados.do), cuando esto sucede por primera vez, el contenedor instancia un servlet del tipo `ActionServlet` y la petición es enviada a un objeto `RequestProcessor`, el cual compara la terminación de la URL (ignorando el `.do`) con la información de los `ActionMapping` configurados en `struts-config.xml` para determinar

la subclase `Action` que debe encargarse de la petición e instanciar un objeto `ActionForm` rellenándolo con los datos del cliente. Una vez que se encuentra y ejecuta el objeto `Action`, se llama a su método principal, `execute()`, en el que se encuentran las llamadas a las diferentes clases del Modelo que se encargarán de manejar dicha petición. Cuando se obtiene la respuesta dada, `Action` devuelve un objeto `ActionForward` al `ActionServlet` para buscar la dirección lógica del recurso en el archivo de configuración y finalmente ser direccionado.

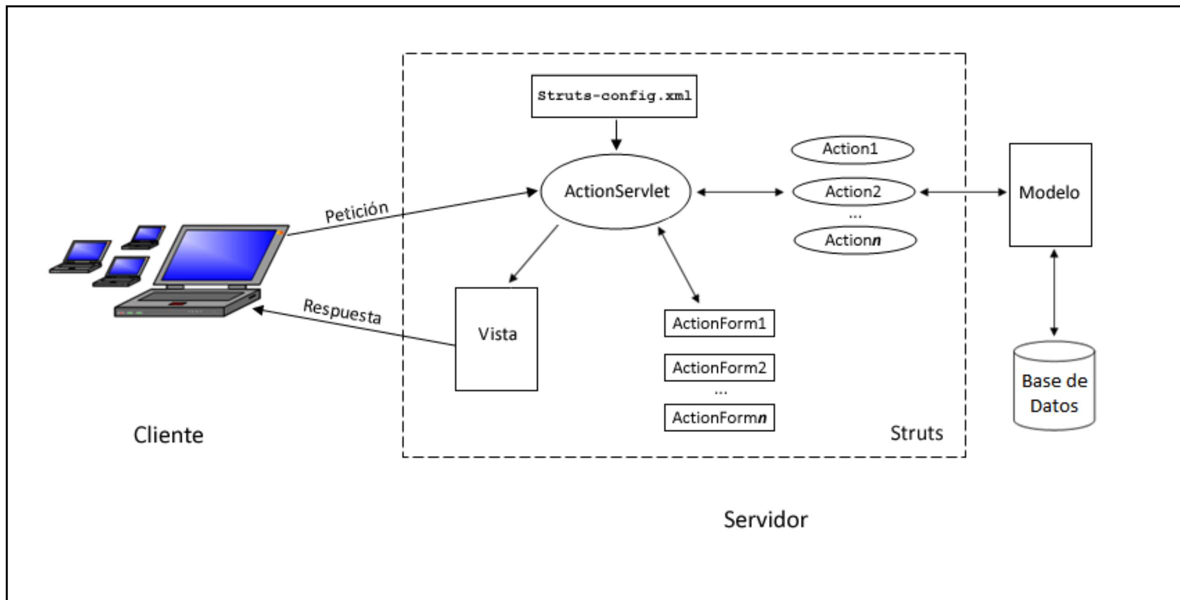


Figura 2.9: Funcionamiento de Struts

En las siguientes tablas se especifican los métodos principales de las clases fundamentales de Struts:

Método	Retorno	Descripción
execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)	ActionForward	Procesa la petición HTTP y envía la respuesta a un recurso lógico.

Tabla 2.7: Método principal de la clase Action

Método	Retorno	Descripción
findForward(String nombre)	ActionForward	Devuelve el objeto ActionForward cuya dirección lógica se específica en el nombre que recibe.
getInputForward()		Devuelve un ActionForward que corresponde al input (definido en struts-config.xml) de la Action.

Tabla 2.8: Métodos de la clase `ActionMapping`

2.4.1 Configuración de Struts

Como ya se mencionó, para que los elementos que conforman Struts puedan ser utilizados, deben estar debidamente registrados y configurados en `struts-config.xml`, cuya estructura básica se muestra a continuación:

```
1   <?xml version="1.0" encoding="UTF-8" ?>
2   <!DOCTYPE struts-config PUBLIC
3   "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
4   "http://jakarta.apache.org/struts/dtds/struts-config_1_3.dtd">
5   <struts-config>
6        <form-beans>
7           <!– Declaración de ActionForm -->
8        </form-beans>
9
10       <global-exceptions>
11          <!– Declaración de excepciones (de ser requeridas)  -->
12       </global-exceptions>
13
14       <global-forwards>
15          <!– Declaración de forward globales (de ser requeridos) -->
16       </global-forwards>
17
18       <action-mappings>
19           <!– Declaración de ActionMapping -->
20       </action-mappings>
21    </struts-config>
```

Código 2.4: Estructura básica del archivo de configuración de Struts

2.4.1.1 Configuración de `ActionForm`

```
1    <form-beans>
2        <form-bean name="ProduccionForm" type="ActionForm.ProduccionForm"/>
3        <form-bean name="UniformeForm" type="ActionForm.UniformeForm"/>
4        <form-bean name="PedidoForm" type="ActionForm.PedidoForm"/>
5                                  …
6    </form-beans>
```

Código 2.5: Registro de objetos ActionForm en `struts-config.xml`

En la etiqueta `<form-beans>` de la línea 1 se declara la lista de objetos `ActionForm` que serán utilizados a lo largo de la aplicación. Su declaración individual se lleva a cabo con la etiqueta `<form-bean>` que tiene como atributos principales:

- `name`: Es un identificador único del bean que será usado como referencia por Struts.
- `type`: Se trata del nombre calificado de la clase.

2.4.1.2 Configuración de `ActionMapping – Action – ActionForward`

```
1    <action-mappings>
2        <action name="UniformeForm" path="/UniformeAction" type="Action.UniformeAction"
3         scope="request">
4            <forward name="verUniformes" path="/verUniformes2.jsp"/>
5        </action>
6        <action name="PedidoForm" path="/PedidoAction" type="Action.PedidoAction"
7         scope="request">
8           <forward name="cambiosP" path="/verPedido.jsp"/>
9           <forward name="cambioPedido" path="/PedidoAction.do?accion=nuevoPedido"/>
10           <forward name="nuevoPedido" path="/verPedido2.jsp"/>
11       </action>
12    </action-mappings>
```

Código 2.6: Registro de objetos `Action`, `ActionMapping` y `ActionForward` en `struts-config.xml`

En la línea 1, con la etiqueta `<action-mappings>` se declaran todos los `ActionMapping` que serán utilizados. Dentro de este, se deben definir los objetos `Action` implementados con la etiqueta `<action>` cuyos atributos principales son:

- `name`: El nombre del objeto `ActionForm` que estará asociado con este `Action`, debe coincidir con el identificador `name` de la etiqueta `<form-bean>`
- `path`: Se trata en realidad del nombre del `Action` que provoca la captura de la petición por el `ActionServlet` y está ubicado entre el último carácter **/** y antes de la extensión `.do` de la URL (/PedidoAction.do)
- `type`: Es el nombre calificado de la clase.
- `scope`: Se trata del alcance que tendrán las variables del `ActionForm` en la apliación, sus posibles valores son: `session` (ámbito de sesión), `request` (ámbito de petición) y `application` (ámbito de aplicación). De no ser especificado, `session` es el valor por defecto.

Para cada objeto `Action`, se deben especificar los posibles `Forward`, a los que será direccionado después de su ejecución, con la etiqueta `<forward>` que tiene como elementos principales:

- `name`: Es el nombre lógico del recurso al que es enviado el usuario.
- `path`: Contiene la URL relativa de dicho recurso (puede ser un documento JSP, HTML o bien otro objeto `Action`) .

2.4.1.3 Configuración de `ActionServlet` en `web.xml`

Este archivo contiene las definiciones necesarias para la aplicación, servlets, algunos mapeos y en ocasiones parámetros de conexión a bases de datos.

```
1      <servlet>
2           <servlet-name>action</servlet-name>
3           <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
4              <init-param>
5                    <param-name>config</param-name>
6                    <param-value>/WEB-INF/struts-config.xml</param-value>
7              </init-param>
8     </servlet>
```

Código 2.7: Registro de `struts-config.xml` en el descriptor de despliegue `web.xml`

```
9     <servlet-mapping>
10              <servlet-name>action</servlet-name>
11              <url-pattern>*.do</url-pattern>
12     </servlet-mapping>
```

Código 2.7 (*continuación*)

De la línea 1 a 8 se lleva a cabo la definición del `ActionServlet` con la etiqueta `<servlet>`, en el que se especifica el nombre y clase del Servlet (líneas 2 y 3), así como la dirección relativa del archivo de configuración `struts-config.xml` en el parámetro de inicialización `config` (líneas 4 a 7).

También es importante definir el tipo de extensión que debe tener toda URL para ser atendida por Struts. De las líneas 9 a 12, se indica mediante el mapeo, que toda URL con terminación `.do` (por convención) sea atendida por el `ServletAction`, previamente configurado.

2.4.2 Librerías de Acciones JSP

Hasta ahora, sólo se ha hablado de lo que Struts hace con el *Controlador*, sin embargo, como se mencionó antes, Struts trabaja también para la *Vista*, y es aquí donde ofrece una serie de librerías de acciones para ser utilizadas en JSP y facilitar la manipulación de los datos obtenidos desde el servidor.

2.4.2.1 Librería Bean

Esta librería es útil para acceder a las propiedades de los JavaBeans usados en la aplicación, así como la creación de nuevos a partir de datos disponibles en los diferentes ámbitos.

Para ser utilizada en una página JSP, debe ser incluida mediante la directiva

```
<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
```

Sus acciones principales son las siguientes:

`<bean:cookie>`

Almacena el valor de una cookie en una variable JSP. Sus atributos principales son:

- `id`: Especifica el nombre de la variable JSP que se creará.
- `name`: Es el nombre de la cookie cuyo valor será almacenado en la variable JSP.

- `value`: En caso de que la cookie no sea encontrada, se crea una con el valor indicado en este atributo.

<bean:define>

Almacena el valor de un objeto existente en la aplicación en una variable JSP. Sus principales atributos son:

- `id`: Especifica el nombre de la variable JSP que se creará.
- `name`: Es el nombre del objeto cuyo contenido será almacenado en la variable JSP.
- `property`: Indica el atributo del objeto especificado en `name`, para que sea este el que se almacene en la variable JSP.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.
- `toScope`: Define el ámbito que tendrá la nueva variable.
- `type`: Indica el tipo de dato de la variable creada (ejemplo java.lang.Integer) el valor por defecto es String.

<bean:message>

Muestra un mensaje almacenado en el archivo `ApplicationResource.properties`. Este es un archivo de texto plano y es usado por Struts para guardar parejas del tipo *clave = valor* y argumentos opcionales (*{0}…{4}*), útil para mostrar mensajes y no modificarlos desde código, también ayuda a la internacionalización de la aplicación mostrando dichos mensajes en diferentes idiomas. Los atributos principales de esta etiqueta son:

- `key`: Clave asociada en el archivo de propiedades al mensaje de texto.
- `arg0`, `arg1`, `arg2`, `arg3`, `arg4`: Indica los argumentos que deben ser mostrado en el mensaje, en sustitución de {0},{1},{2},{3} y {4}.

<bean:write>

Obtiene el valor de un objeto y lo muestra en la página. Sus principales atributos son:

- `name`: Es el nombre del objeto cuyo contenido será mostrado.
- `property`: Indica el atributo del objeto especificado en `name`, para que este se muestre en la página.

- `format`: Especifica el formato en que será mostrado el objeto.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.

2.4.2.2 Librería Logic

Es en esta librería donde se definen las diferentes maneras de manejar el control de flujo de datos recibidos por el servidor.

Para ser utilizada en una página JSP, debe ser incluida mediante la directiva

```
<%@taglib uri="http://struts.apache.org/tags-logic" prefix="logic"%>
```

Sus acciones principales se muestras a continuación:

`<logic:empty>` y `<logic:notEmpty>`

La primera evalúa si un objeto se encuentra vacío, su contraparte es la etiqueta `notEmpty`, que buscará que un objeto contenga elementos. Sus atributos principales son:

- `name`: Es el nombre del objeto que será evaluado.
- `property`: Indica el atributo del objeto especificado en `name`, para su evaluación.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.

`<logic:equal>` y `<logic:notEqual>`

La primer etiqueta evalúa que el contenido de un objeto sea igual al valor de especificado. Como contraparte está la etiqueta `<logic:notEqual>` encargada de evaluar que el contenido del objeto no sea igual al valor especificado. Sus atributos principales son:

- `cookie`: Nombre de la cookie cuyo contenido será evaluado.
- `name`: Es el nombre del objeto cuyo contenido será evaluado.
- `property`: Indica el atributo del objeto especificado en `name`, para que este sea evaluado.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.
- `value`: Especifica el valor a comparar con el objeto y cookie dados.

<logic:greaterEqual> <logic:greaterThan> y
<logic:lessEqual> <logic:lessThan>

El propósito de estas etiquetas, al igual que en las 2 anteriores, es evaluar el contenido de un objeto o una cookie, indicando si es *mayor o igual*, *mayor que*, *menor o igual* o *menor que* el valor especificado en `value`. Sus atributos principales son los mismos que en las etiquetas anteriores.

<logic:iterate>

Se encarga de recorrer una colección según las condiciones dadas. Sus principales atributos son:

- `name`: Es el nombre de la colección que será iterada.
- `property`: Indica el atributo del objeto especificado en `name`, que contiene la colección.
- `scope`: Ámbito en el que se debe buscar el objeto especificado en `name`. De no ser señalado, se buscará en el siguiente orden: `page`, `request`, `session`, y `application`.
- `id`: Nombre de la variable JSP que almacenará la referencia del objeto actual en la iteración. Su ámbito estará limitado a la página.
- `indexId`: Nombre de la variable JSP que almacenará el número de iteración actual. Su ámbito estará limitado a la página.
- `length`: Limita el número de iteraciones en la colección.
- `offset`: Especifica el índice del cual debe comenzar la iteración.

Struts no ofrece un soporte directo para la capa del Modelo, el cual es conveniente que se divida en 2 subcapas: La lógica de negocio, que contendrá todas aquellas clases que encaminen a las peticiones de los objetos `Action` hacia la subcapa de acceso a datos responsable de obtener la información solicitada desde una fuente de datos.

2.5 JDBC (*Java DataBase Conectivity*)

JDBC es el API estándar proporcionado por Sun para las aplicaciones Java, que se encarga de la manipulación de datos por medio de un conjunto de clases e interfaces que permiten, una vez implementadas, llevar a cabo la comunicación entre una base de datos relacional y la aplicación.

La mayoría de los manejadores de bases de datos ofrecen su propia implementación de este API (llamados *drivers* o *controladores*), que por estar basado en interfaces, define los métodos, los tipos de objetos que recibe y los tipos de valores u objetos que debe regresar. Esto tiene como ventaja fundamental que la aplicación pueda ser portable en cuanto a manejador de base de datos se refiere. Así, si la aplicación lo llegara a requerir, los datos pueden ser migrados a un RDBMS (*Relational Database Magement System*) distinto sin que el código deba ser alterado, tan sólo bastaría con modificar la configuración y el driver del manejador específico.

2.5.1 API de JDBC

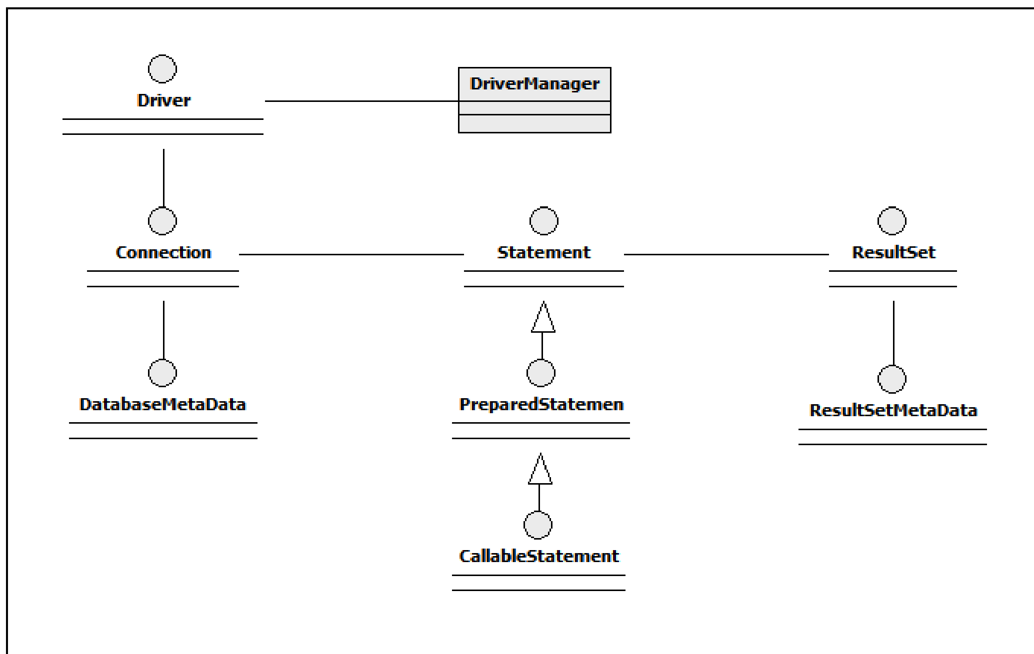


Figura 2.10: API general de JDBC

Las componentes principales del API se enlistan a continuación:

- `DriverManager`: Es la clase responsable de cargar el driver indicado y manejar las distintas conexiones con la base de datos.
- `Driver`: Es la interface que cada controlador debe implementar para ser llamada por el `DriverManager`.
- `Connection`: Se encarga de administrar la sesión entre la base de datos y la aplicación.
- `Statement`: Permite ejecutar sentencias SQL. Cuenta con 2 subclases: `PreparedStatement` y `CallableStatement`. La primera representa una sentencia SQL precompilada que puede ser ejecutada `n` veces con parámetros diferentes, útil para aumentar el rendimiento de sentencias `insert` múltiples. La segunda permite la manipulación de datos por medio de procedimientos almacenados.
- `DatabaseMetaData`: Encapsula la información relacionada con la base de datos, el driver y los metadatos.
- `Resultset`: Contiene las filas y columnas que regresa la base de datos tras la ejecución de la sentencia SQL.
- `ResultSetMetaData`: Contiene los metadatos del objeto `ResultSet`.

Las siguientes tablas se muestra el detalle de las clases e interfaces que componen el API de JDBC.

Método	Retorno	Descripción
getConnection(String url)	Connection	Intenta establecer una conexión a la base de datos con la URL y parámetros dados.
getConnection(String url, String usuario, String password)		
getConnection(String url, Properties info)		

Tabla 2.9: Métodos principales de la clase `DriverManager`

<table>
<tr><th>Método</th><th>Retorno</th><th>Descripción</th></tr>
<tr><td>close()</td><td rowspan="2">void</td><td>Libera los recursos ocupados para la conexión.</td></tr>
<tr><td>commit()</td><td>Guarda los cambios hechos, desde el último commit/rollback, de manera permanente en la base de datos.</td></tr>
<tr><td>createStatement()</td><td>Statement</td><td>Crea un objeto <code>Statement</code> para la ejecución de sentencias SQL</td></tr>
<tr><td>getMetaData()</td><td>DatabaseMetaData</td><td>Obtiene los metadatos de la conexión actual.</td></tr>
<tr><td>prepareStatement(String sql)</td><td>PrepareStatement</td><td>Crea un objeto <code>PrepareStatement</code> para la ejecución de sentencias SQL parametrizadas.</td></tr>
<tr><td>rollback()</td><td>void</td><td>Deshace todos los cambios hechos desde el último commit/rollback en la base de datos.</td></tr>
</table>

Tabla 2.10: Métodos principales de la Interface `Connection`

Método	Retorno	Descripción
close()	void	Libera los recursos ocupados por el objeto `Statement`.
execute(String sql)	boolean	Ejecuta una sentencia SQL que podría regresar múltiples valores.
executeQuery(String sql)	ResultSet	Ejecuta una sentencia SQL que regresa un objeto `ResultSet`.
executeUpdate(String sql)	int	Ejecuta una sentencia INSERT, DELETE o UPDATE.
setMaxRows(int max)	void	Fija el número máximo de filas que podrá tener el `ResultSet`.

Tabla 2.11: Métodos principales de la Interface `Statement`

Método	Retorno	Descripción
setTipo(int índice, Tipo valor) *Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).*	Tipo	Determina el tipo de dato de la columna indicada por el índice.

Tabla 2.12: Métodos principales de la Interface `PreparedStatement`

Método	Retorno	Descripción
close()	void	Libera los recursos ocupados por el objeto `ResultSet`.
first()	boolean	Mueve el puntero a la primer fila del `ResultSet`.
getTipo(int índice) *Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).*	Tipo	Regresa el contenido de la columna indicada por el índice, según sea el tipo de dato, en la fila actual del `ResultSet`.
last()	boolean	Mueve el puntero a la última fila del `ResultSet`.
next()		Mueve el puntero a la siguiente fila de la actual en el `ResultSet`.
previous()		Mueve el puntero a la fila anterior de la actual en el `ResultSet`.
relative(int n)		Mueve el puntero `n` filas anteriores o posteriores a la actual el `ResultSet`.

Tabla 2.13: Métodos principales de la Interface `ResultSet`

En el siguiente código se muestra cómo se realiza la conexión a una base de datos por medio de los componentes del API:

```
1   public class AccesoConJDBC {
2       Connection conexion = null;
3       Statement consulta = null;
4       PreparedStatement insert = null;
5       ResultSet resultados = null;
6       static final String Driver = "oracle.jdbc.OracleDriver";
7       static final String URL = "jdbc:oracle:thin:@127.0.0.1:1521:XE";
8       static final String usuario = "hugo";
9      static final String pass = "hugo";
10
11      public List<Proveedores> mostrarProveedores() {
12          List<Proveedores> lista = new ArrayList<Proveedores>();
13          try {
14              Class.forName(Driver);
15              conexion = DriverManager.getConnection(URL, usuario, pass);
16
17              consulta = conexion.createStatement();
18              resultados = consulta.executeQuery("SELECT idProveedor" +
19                      ",nombre,inicioactividad,finActividad," +
20                      "status FROM proveedores");
21              while (resultados.next()) {
22                  Proveedores proveedor = new Proveedores();
23                  proveedor.setIdproveedor(resultados.getInt(1));
24                  proveedor.setNombre(resultados.getString(2));
25                  proveedor.setInicioactividad(resultados.getDate(3));
26                  proveedor.setFinactividad(resultados.getDate(4));
27                  proveedor.setStatus(resultados.getString(5));
28                  lista.add(proveedor);
29              }
30          } catch (SQLException e) {
31              System.out.println("Se reportó una excepción: " + e);
32          } catch (ClassNotFoundException e) {
33              System.out.println("No se encontró el driver: " + e);
34          } finally {
35              try {
36                  consulta.close();
37                  resultados.close();
38                  conexion.close();
39              } catch (SQLException e) {
40                  System.out.println("Error al liberar recursos: " + e);
41              }
42          }
43          return lista;
44      }}
```

Código 2.8: Conexión y consulta a una base de datos por medio de JDBC

De las líneas 4 a 11 se declaran los distintos objetos utilizados para la manipulación de datos: `Connection`, `Statement`, `PreparedStatement` y `ResultSet`. Así como los parámetros requeridos por JDBC:

- La cadena que contiene la clase cualificada del Controlador (que implementa la interface Driver).
- La cadena que contiene la URL de la base de datos (siguiendo el formato [protocolo de comunicación]:[subprotocolo]:identificador propio de cada RDBMS).
- Nombre y contraseña del usuario con los permisos suficientes para acceder a los datos.

En la línea 14 se carga el driver y se inicia la conexión por medio del `DriverManager` con los parámetros previamente declarados. El objeto `Connection` hace referencia a un objeto `Statement` para poder ejecutar sentencias SQL en la línea 18 y se obtiene el objeto `ResultSet` tras la ejecución en la línea 18.

Para que los resultados puedan ser manejados se debe iterar sobre el `ResultSet`, y en este caso, de la línea 22 a la 27 los valores de cada fila se recuperan y almacenan (con los diferentes métodos que ofrece `ResultSet` para cada tipo de dato) en un objeto de tipo `Proveedores` que son agregados a una lista que contiene dichos objetos en la línea 28.

Finalmente de la línea 30 a la 33 se reportan las excepciones que puedan existir tanto por el Driver como por alguno de los elementos involucrados en la conexión y se liberan los recursos utilizados de la línea 36 a la 40.

Pese a las ventajas de portabilidad que ofrece JDBC, y tal como se observa en el ejemplo anterior, el código que se necesita escribir para la manipulación de datos es demasiado extenso, a pesar de que el problema queda resuelto en 3 líneas (18 a 20). Se estima que el 30% del código escrito en una aplicación se ocupa del problema de acceso a datos, desperdiciando con esto muchos de los recursos humanos que deberían ser invertidos en el problema principal: La lógica de negocio.

Otro problema de JDBC es que los objetos no pueden ser persistidos directamente de (o hacia) la base de datos sino que deben ser manejados atributo por atributo (líneas 23 a 27), lo cual se complica cuando la tabla contiene un número significativo de columnas.

2.6 Hibernate

Aquel tipo de problemas a los que se enfrentaban los programadores que usaban JDBC sirvieron como antecedente para el nacimiento de un nuevo paradigma. El paradigma de persistencia en objetos que permite almacenar el estado de un objeto de manera permanente para que sea recuperado cuando sea necesario. Esto trajo consigo una técnica que permite persistir los objetos en una base de datos relacional de manera eficaz: Mapeo Objeto-Relacional (ORM por sus siglas en inglés).

Hibernate se trata de una herramienta ORM que facilita las tareas de persistencia en la aplicación. Sus principales ventajas sobre el uso directo con JDBC son:

- Productividad: Debido a que elimina buena parte del código relacionado con el acceso y manejo de datos.
- Mantenimiento: Al contar con menos líneas de código, este se hace más comprensible y fácil de actualizar.
- Los cambios en el modelo de datos no afectan el funcionamiento programado.
- Simplicidad: Las transformaciones de los objetos para poder ser persistidos son innecesarias.
- Eficiencia: Realiza sus tareas con un mínimo de conexiones a la base de datos.
- Provee su propio lenguaje extendido de SQL: HQL (*Hibernate Query Language*) que de igual manera utiliza uniones (joins), funciones de agregación y funciones de agrupación, con la diferencia que HQL está totalmente orientado a objetos.
- Conserva la portabilidad que ofrece JDBC debido a que ocupa esta API por ser el estándar definido.

2.6.1 Componentes de Hibernate

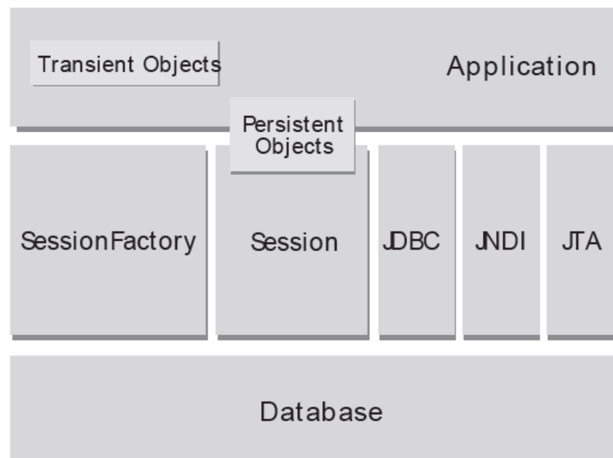


Figura 2.11: Componentes generales de Hibernate

La arquitectura básica de hibernate está formada por los siguientes elementos:

- `Session`: Es el objeto principal utilizado por Hibernate representando la conexión entre la base de datos y la aplicación, también llamada sesión. Se encarga de ejecutar las operaciones de almacenamiento y búsqueda de datos estando asociado a un sólo hilo de ejecución debido a que estos objetos son instanciados y destruidos por cada petición que Hibernate recibe de la aplicación.
- `SessionFactory`: Se encarga de crear los objetos `Session` cuando la aplicación los requiera. Existe sólo una instancia `SessionFactory` a lo largo de la aplicación (una por cada conexión a una base de datos diferente) y su estado es inmutable.
- `Configuration`: Es el responsable de la configuración de Hibernate de acuerdo a los archivos de mapeo y de crear el `SessionFactory` una vez que la configuración fue terminada. Es el primer objeto con el que se tiene contacto cuando se llama a Hibernate y es descartado una vez que se ha instanciado el `SessionFactory`.
- `Query`: Ejecuta una sentencia generalmente escrita en HQL permitiendo la parametrización de esta con resultados óptimos (análogamente con el objeto `PreparedStatement` de JDBC).

- Objetos persistentes: Se trata de la representación de las tablas de la base de datos en la aplicación por medio de JavaBeans, también llamados POJO (*Plain Old Text Object*). Se encuentran asociados a un objeto `Session` para ser persistidos.

A continuación se muestra el resumen de los métodos principales de clases e interfaces que conforman la arquitectura de Hibernate

Método	Retorno	Descripción
buildSessionFactory()	SessionFactory	Crea un objeto `SessionFactory`.
configure()	Configuration	Busca las propiedades y mapeos del archivo de configuración para ser utilizados en la creación del `SessionFactory`. Si no se especifica el archivo de configuración usa `hibernate.cfg.xml` por defecto.
configure(File configFile)		
configure(String resource)		
configure(URL url)		

Tabla 2.14: Métodos principales de Interface `Configuration`.

Método	Retorno	Descripción
close()	void	Libera los recursos ocupados por el objeto `SessionFactory`.
getCurrentSession()	Session	Obtiene el objeto `Session` actual.
openSession()		Instancia un nuevo `Session`.

Tabla 2.15: Métodos principales de Interface `SessionFactory`.

Método	Retorno	Descripción
beginTransaction()	Transaction	Encapsula una unidad atómica de trabajo (transacción).
cancelQuery()	void	Cancela la ejecución de la sentencia SQL.
close()		Libera los recursos ocupados por el objeto `Session`.
createQuery(String hql)	Query	Crea una instancia de un objeto Query que estará asociado con el objeto `Session` y ejecutará una sentencia HQL.
delete()	void	Elimina un objeto persistente de la base de datos.
get(Class clase, Serializable id)	Object	Regresa el objeto persistente perteneciente a la clase e identificador dados, o nulo, de existir.
load(Class clase, Serializable id)		Regresa el objeto persistente perteneciente a la clase e identificador dados, asumiendo que este existe.
save(Object objeto)	Serializable	Persiste en la base de datos el objeto dado.
update(Object objeto)	void	Actualiza el objeto indicado.

Tabla 2.16: Métodos principales de Interface `Session`.

Método	**Retorno**	**Descripción**
iterate()	Iterator	Regresa el resultado de la sentencia en un objeto `Iterator`.
list()	List	Regresa el resultado de la sentencia en un objeto `List`.
scroll()	ScrollableResults	Regresa el resultado de la sentencia en un objeto `ScrollableResults`.
setTipo(int índice, Tipo valor) *Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).*	Query	Establece el valor del parámetro indicado por el índice o el nombre para ser enviado en la consulta.
setTipo(String nombre, Tipo valor) *Tipo: Se refiere al tipo de dato de la columna (int, long, float, double, String, Date, Timestamp…).*		
setFirstResult(int primerFila)		Indica la primer fila que debe ser regresada.
setMaxResults(int maximoFilas)		Fija el número máximo de filas que la sentencia debe regresar.
setParameter(int índice, Object valor)		Establece el valor del parámetro indicado para una sentencia con nombre.
setParameter(String nombre, Object valor		
uniqueResult()	Object	Regresa una instancia única del objeto resultado de la sentencia.

Tabla 2.17: Métodos principales de Interface `Query`.

2.6.2 Configuración de Hibernate

2.6.2.1 Configuración de `Configuration`

Hibernate ofrece 2 alternativas de configuración: Un archivo de texto plano llamado `hibernate.properties` y un archivo XML llamado `hibernate.cfg.xml`, en ellos se especifican los parámetros necesarios para la conexión a la base de datos: URL, driver, dialecto, usuario, contraseña, etc. Así como el mapeo de las clases persistentes y algunos parámetros extra propios del framework. En caso de encontrarse los 2 archivos, `hibernate.cfg.xml` sobreescribe las propiedades configuradas en `hibernate.properties`. Para fines prácticos en la construcción de la aplicación, la configuración se llevará a cabo en el archivo `hibernate.cfg.xml`.

La estructura de este archivo se muestra a continuación:

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <!DOCTYPE hibernate-configuration PUBLIC
3    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5    <hibernate-configuration>
6      <session-factory>
7        <property name="hibernate.dialect">
8            org.hibernate.dialect.OracleDialect
9        </property>
10        <property name="hibernate.connection.driver_class">
11           oracle.jdbc.OracleDriver
12        </property>
13        <property name="hibernate.connection.url">
14            jdbc:oracle:thin:@127.0.0.1:1521:XE
15        </property>
16        <property name="hibernate.connection.username">usuario</property>
17        <property name="hibernate.connection.password">password</property>
18
19        <!– Mapeo de las clases persistentes -->
20
21       </session-factory>
22   </hibernate-configuration>
```

Código 2.9: Configuración de hibernate mediante el archivo `hibernate.cfg.xml`.

2.6.2.2 Configuración de SessionFactory

Una vez que la configuración básica (la conexión a la base de datos, el usuario, contraseña, driver y las clases persistentes) se ha llevado a cabo, es conveniente tener por separado una clase de ayuda llamada HibernateUtil, que se encargue de instanciar el SessionFactory y optimice la creación de los objetos Session. Algunos ambientes de desarrollo como NetBeans, ya traen esta clase prediseñada, por lo que el programador no necesita crearla, sin embargo, es importante comprender su funcionamiento para que pueda ser usada correctamente.

```
1    public class HibernateUtil {
2        private static final SessionFactory sessionFactory;
3
4        static {
5            try {
6
7            sessionFactory = new Configuration().configure().buildSessionFactory();
9            } catch (Throwable e) {
11               System.out.println("Fallé al iniciar SessionFactory"+e);
12               throw new ExceptionInInitializerError(e);
13           }
14       }
15
16       public static SessionFactory getSessionFactory() {
17           return sessionFactory;
18       }
19   }
```

Código 2.10: Clase HibernateUtil encargada de crear el objeto SessionFactory

En la línea 2 se declara el objeto SessionFactory que se utilizará a lo largo de la aplicación y es inicializado en la línea 7 por medio de la creación de un objeto Configuration, el cual busca el archivo hibernate.cfg.xml para cargar las propiedades y mapeos con el método configure() y a partir de esto crear el objeto SessionFactory con el método buildSessionFactory(). Finalmente se regresa la referencia del SessionFactory en el método getSessionFactory() de la línea 16.

2.6.2.3 Configuración de `Session`

En este punto, que ya se tiene el objeto `SessionFactory` listo para crear objetos `Session`, se debe contar con la clase que defina al objeto que será persistido en la base de datos, la cual debe estar registrada en `hibernate.cfg.xml`. Al conjunto de estas clases se le conoce como Modelo de Domino.

La configuración de las clases persistentes puede ser hecha de 2 maneras:

1. Por medio de un archivo XML en el que se definirán las propiedades de mapeo, tales como el nombre de la tabla equivalente a esa clase, el atributo id, las relaciones con otros objetos del modelo de dominio y el nombre de la columna correspondiente al atributo. Debe existir un archivo archivo `objeto.hbm.xml` por cada clase persistente.
2. A partir de la versión 5 de Java se introdujo el uso de anotaciones dentro del código para disminuir el tiempo de desarrollo. Hibernate ofrece soporte para esta funcionalidad y la incluye para definir las propiedades de mapeo directamente en los objetos POJO.

Antes de poder elegir entre estas dos opciones es conveniente analizar los siguientes puntos:

- El soporte para las anotaciones está dado a partir de Hibernate3 y si se está migrando una aplicación previa a esta versión, es conveniente conservar los archivos XML originales para no volver a configurar los mapeos y así evitar errores.
- Si no se cuenta con el código de las clases de dominio será imposible el uso de anotaciones.
- El uso de anotaciones hace que los mapeos se vuelvan más intuitivos y comprensibles.
- Para aplicaciones cuyo modelo de dominio es extenso, resulta complicada la gestión de los archivos XML por separado.

Retomando el ejemplo mostrado para JDBC (código 2.8) con la tabla proveedores se muestra a continuación los 2 tipos de configuración:

2.6.2.4 Configuración del mapeo con archivo.hbm.xml

Proveedores.class

```
1   public class Proveedores {
2       private int idproveedor;
3       private Date inicioactividad;
4       private String nombre;
5       private char status;
6       private Date finactividad;
7
9    //métodos getter y setter
10  }
```

Código 2.11: Configuración del mapeo de la clase Proveedores en un archivo xml externo

Proveedores.hbm.xml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <!DOCTYPE hibernate-mapping PUBLIC
3   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5   <hibernate-mapping>
6     <class name="Persistencia.Proveedores" table="proveedores">
7       <id name="idproveedor" column=" idproveedor"/>
8       <property name="nombre" type="string"/>
9       <property name="inicioactividad" type="java.util.Date"/>
10      <property name="status" type="string"/>
11      <property name="finactividad" type="java.util.Date"/>
12    </class>
14  </hibernate-mapping>
```

Código 2.11 (*continuación*)

La clase Proveedores únicamente cuenta con sus atributos y los métodos getter y setter para el transporte de los datos.

Por otro lado, en el archivo Proveedores.hbm.xml se lleva a cabo la configuración del mapeo a partir de la línea 5. En la etiqueta class de la línea 6 se declara el nombre calificado de la clase y la tabla a la que será mapeada. El atributo identificador se debe configurar dentro de la etiqueta id de la línea 7 y se especifican los nombres de este y de la columna, si ambos son iguales, este

último puede ser omitido. Para los atributos restantes se utiliza la etiqueta property, de las líneas 8 a 11, en la que se especifican el nombre y tipo de dato del atributo.

2.6.2.5 Configuración del mapeo con anotaciones

Proveedores.class

```
1   @Entity
2   @Table(name="proveedores")
3   public class Proveedores {
4
5       @Id
6       private int idproveedor;
7       @Temporal(TemporalType.DATE)
8       private Date inicioactividad;
9       private String nombre;
10      private char status;
11      @Temporal(TemporalType.DATE)
12      private Date finactividad;
13
14      //métodos getter y setter
15   }
```

Código 2.12: Configuración del mapeo de la clase Proveedores usando anotaciones.

La anotación @Entity indica que esa clase se trata de un objeto persistente hacia la tabla indicada con @Table. Cuando el nombre de la clase y la tabla es el mismo, esta última puede ser omitida. La anotación @Id de la línea 5 debe ser empleada en el atributo identificador. Cada una de las propiedades puede ser anotada con @Column indicando el nombre de su columna equivalente y puede ser excluida cuando el nombre de estos coinciden. Aquellos atributos que hagan referencia a columnas del tipo DATE o TIMESTAMP deben ser anotados con @Temporal.

Debido a las ventajas que presenta el mapeo de clases usando anotaciones, con respecto a los archivos XML, será esta la opción utilizada para la construcción de la aplicación.

Una vez que se ha configurado el mapeo, la clase debe ser registrada en hibernate.cfg.xml para que sea cargada cuando se crea la instancia de Configuration. Mediante el atributo class de la etiqueta mapping especificando el nombre calificado de la clase persistente (En caso de utilizar archivos XML, este es el que se registra).

```
19   <!– Mapeo de las clases persistentes -->
20   <mapping class="Persistencia.Proveedores"/>
```

Código 2.8 (*continuación*):

Ya en este punto, cuando la configuración previa se ha realizado, es el que pueden ser instanciados los objetos Session para comenzar con la persistencia de datos. El siguiente código muestra cómo recuperar de la base de datos una lista de objetos Proveedores.

```
1    public List<Proveedores> mostrarProveedores(){
2        Session = sesion;
3        List<Proveedores> proveedores;
4        sesion=HibernateUtil.getSessionFactory().openSession();
5        proveedor = sesion.createQuery("From Proveedores").list();
6        sesion.close();
7        return proveedor;
8     }
```

Código 2.13: Persistencia de los objetos Proveedores desde la base de datos con Hibernate.

En la línea 2 se declara un objeto Session inicializado por el método openSession() del SessionFactory que es referenciado a través de HibernateUtil en la línea 4. Una vez que se tiene abierta una nueva sesión, por medio del método createQuery() se ejecuta una sentencia HQL que será traducida al lenguaje nativo del RDBMS y se enviará a JDBC para su ejecución, que una vez realizada, regresa los datos en el formato indicado, en este caso, una lista de objetos Proveedores. Finalmente en la línea 6 se cierra la sesión para liberar recursos.

En el siguiente código se muestra la forma en que el estado del objeto es persistido a la base de datos:

```
1    public void guardarProveedores (Proveedores Pv){
2        Session sesion;
3        Transaction tran;
4        sesion=HibernateUtil.getSessionFactory().openSession();
5        tran=sesion.beginTransaction();
6        sesion.save(Pv);
7        tran.commit();
8        sesion.close();
9        tran.close();
10   }
```

Código 2.14: Persistencia de los objetos `Proveedores` hacia la base de datos con Hibernate.

Para modificar información en la base de datos (inserción, borrado o actualización), aparte del objeto `Session`, se necesita un objeto del tipo `Transaction` que maneje la transacción de la modificación. En la línea 3 es declarado e inicializado en la línea 5 con el método `beginTransaction()` del objeto `Session`. En la línea 6 se persiste el objeto recibido y para que su estado quede de manera permanente en la base de datos debe ser confirmado por medio del método `commit()`, de lo contrario no será almancenado. Finalmente en la líneas 7 y 8 se liberan los recursos de la sesión y la transacción.

Es importante tener en cuenta que Hibernate **no** sustituye a JDBC, pues la comunicación con la base de datos siempre se realiza por medio de este, el propósito de Hibernate es que el desarrollador sólo se enfoque en resolver qué datos necesita y no cómo acceder a estos. Para ello sólo se requiere configurar el mapeo de las entidades persistentes con las anotaciones precisas.

2.6.2.6 Anotaciones para el mapeo de atributos

- **`@Entity`**

  Sirve para indicarle a Hibernate las clases persistentes que deben ser mapeadas.

- **`@Id`**

  Debe ser empleada en el campo que funge como identificador o clave primaria de la clase.

- **`@GeneratedValue(strategy, generator)`**

  A pesar de que `@Id` es capaz por sí misma de elegir la mejor estrategia de generación de claves primarias, esta anotación sobreescribe dicha funcionalidad para ser asignada manualmente. Recibe 2 atributos:

  1. `strategy`: Define la estrategia de generación de claves, puede ser de 4 tipos:
     - 1.1. `GenerationType.AUTO`: Hibernate elige qué método usar de acuerdo al soporte dado por el RDBMS.
     - 1.2. `GenerationType.IDENTITY`: El RDBMS es el responsable de asignar este valor automáticamente conforme se van insertando lo datos.
     - 1.3. `GenerationType.SEQUENCE`: La asignación se lleva a cabo por medio de una secuencia creada en la base de datos.
     - 1.4. `GenerationType.TABLE`: Una tabla que contiene los valores primarios es la encargada de hacer la asignación.
  2. `generator`: Es el identificador de la generación de las claves.

- **`@SequenceGenerator(name,sequenceName,initialValue,allocationSize)`**

  Establece las propiedades de la generación de claves primarias a través de una secuencia. Sus atributos son:

  1. `name`: Es el nombre que identifica la secuencia y que deberá coincidir con el atributo `generator` de `@GeneratedValue`.

2. `sequenceName`: Es el nombre real de la secuencia en el RDBMS.
3. `initialValue`: Define el número con el que debe comenzar la secuencia
4. `allocationSize`: Indica el incremento que debe haber entre casa número creado.

- **`@EmbeddedId y @Embeddable`**

  Las claves primarias compuestas deben ser tratadas por JavaBeans aislados, en los que se declaren los atributos que forman la llave compuesta. Esta clase debe estar anotada con `@Embeddable` (en vez de `@Entity`) y el objeto persistente deberá indicar por medio de `@EmbeddedId` el objeto que define las clave compuesta (en sustitución de `@Id`).

- **`@Column(name, length, nullable, unique,…)`**

  Es una anotación opcional, útil cuando la columna y el atributo no tienen el mismo nombre o cuando se pretende crear las tablas a partir de los mapeos configurados. Sus atributos más importantes son:

  1. `name`: Indica el nombre de la columna.
  2. `length`: Especifica la longitud del atributo cuando es de tipo String. Si se trata de `double`, `float` o `int` se usan los atributos `scale` y `precision`.
  3. `nullable`: Indica si la columna debe ser NULL o NOT NULL.
  4. `unique`: Asigna la restricción de valor único a la columna.

- **`@Table(name, schema,…)`**

  Se trata de una anotación opcional que indica el nombre de la tabla a la que será mapeada la clase, útil cuando los nombres de estas no coinciden o cuando se intenta crear las tablas a partir de los mapeos configurados. Sus atributos más importantes son:

  1. `name`: Indica el nombre de la tabla.
  2. `schema`: Define el usuario de la base de datos sobre el cual se está trabajando.

- **`@Temporal(TemporalType)`**

  Los atributos que manejan valores temporales deben ser anotados con esta anotación para indicar cómo serán mapeados hacia la base de datos: `DATE`, `TIME` o `TIMESTAMP`.

2.6.2.7 Anotaciones para el mapeo de asociaciones

Una asociación es la relación que guardan las clases persistentes entre sí: uno a uno, uno a muchos, muchos a uno y muchos a muchos. Existen 2 tipos de asociaciones:

- Unidireccionales: La navegabilidad entre las clases se lleva a cabo sólo en un sentido, y únicamente la clase en la que se define la asociación tiene conocimiento de la existencia de su clase asociada.
- Bidireccionales: La navegabilidad de las clases asociadas se lleva a cabo en los 2 sentidos. Cada clase es "consciente" de la existencia de la otra.

Es recomendable el uso de asociaciones bidireccionales con la finalidad de no limitar la navegabilidad en los datos, sin embargo, esto debe ser decidido por el desarrollador de acuerdo a las especificaciones. A continuación se muestran las anotaciones necesarias para mapear asociaciones:

- **`@OneToOne(cascade, fetch, mappedBy, optional,…)`**

  Esta anotación representa una asociación uno a uno entre 2 clases. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. Sus opciones son:

     1.1. `CascadeType.PERSIST`: Inserción de datos en cascada.

     1.2. `CascadeType.REFRESH`: Actualizaciones en cascada.

     1.3. `CascadeType.REMOVE`: Borrado de datos en cascada.

     1.4. `CascadeType.ALL`: Indica que todas las operaciones serán realizadas en cascada.

     En caso de no ser indicado este atributo, ninguna operación se realizará en cascada.

  2. `fetch`: Cuando el estado de un objeto es persistido desde la base de datos existen 2 maneras de manejar las relaciones con los diferentes objetos.

     2.1 `FetchType.EAGER`: Indica que el objeto persistido traerá consigo los datos de sus objetos asociados.

     2.2 `FetchType.LAZY`: Las asociaciones del objeto no serán persistidas cuando este lo sea.

Es recomendable usar siempre asociaciones con carga perezosa (LAZY) para mejorar el rendimiento de las consultas y sólo llamar a los objetos asociados cuando sea necesario (a través de los joins de HQL).

3. `mappedBy`: En las relaciones bidireccionales la clase padre debe indicar con este atributo los objetos que dependen de ella.
4. `optional`: Indica si el atributo mapeado puede ser nulo.

- **`@OneToMany(cascade, fetch, mappedBy, optional,…)`**

  Representa la asociación uno a muchos entre 2 clases. El lado izquierdo al `To` indica la cardinalidad de la clase que lleva la anotación, el lado derecho es la cardinalidad de la clase asociada representada como una colección de objetos del tipo `Set`. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. En caso de no ser incluido ninguna operación se realizará en cascada.
  2. `fetch`: Especifica el tipo de carga que tendrán las asociaciones relacionadas con el objeto.
  3. `mappedBy`: En las relaciones bidireccionales la clase padre debe indicar con este atributo los objetos que dependen de ella.
  4. `optional`: Indica si el atributo mapeado puede ser nulo.

- **`@ManyToOne(cascade, fetch, optional,…)`**

  Representa la asociación muchos a uno entre 2 clases. Sus atributos principales son:

  1. `cascade`: Define las operaciones que podrán ser realizadas en cascada. En caso de no ser incluido ninguna operación se realizará en cascada.
  2. `fetch`: Especifica el tipo de carga que tendrán las asociaciones relacionadas con el objeto.
  3. `optional`: Indica si el atributo mapeado puede ser nulo.

- **@JoinColumn(name,insertable,updatable)**

  Indica la clave foránea en las relaciones hijas anotadas con @ManyToOne. Sus atributos principales son:

  1. name: El nombre del atributo que representa la clave foránea.
  2. insertable: Indica si el atributo debe ser insertado cuando sea mapeado.
  3. updatable: Indica si el atributo debe ser actualizado cuando sea mapeado.

Aquellas clases que tengan la clave foránea como parte de una clave primaria compuesta deben especificar los atributos insertable y updatable como false para evitar un doble mapeo de la clave foránea.

- **@ManyToMany (cascade, fetch, mappedBy,…)**

  Representa la asociación muchos a muchos entre 2 clases. Sus atributos principales son:

  1. cascade: Define las operaciones que podrán ser realizadas en cascada. En caso de no ser incluido ninguna operación se realizará en cascada.
  2. fetch: Especifica el tipo de carga que tendrán las asociaciones relacionadas con el objeto.
  3. mappedBy: En las relaciones bidireccionales la clase padre debe indicar con este atributo los objetos que dependen de ella.

Esta anotación sólo es útil cuando se trata de una relación muchos a muchos pura, es decir, que no tenga algún atributo extra a las claves foráneas que forman el identificador compuesto, de otra manera, la relación debe ser tratada por separado como 2 asociaciones uno a muchos.

Continuando con el ejemplo de los proveedores, a continuación se muestra cómo se realiza el mapeo de la asociación entre Proveedores y Sucursales.

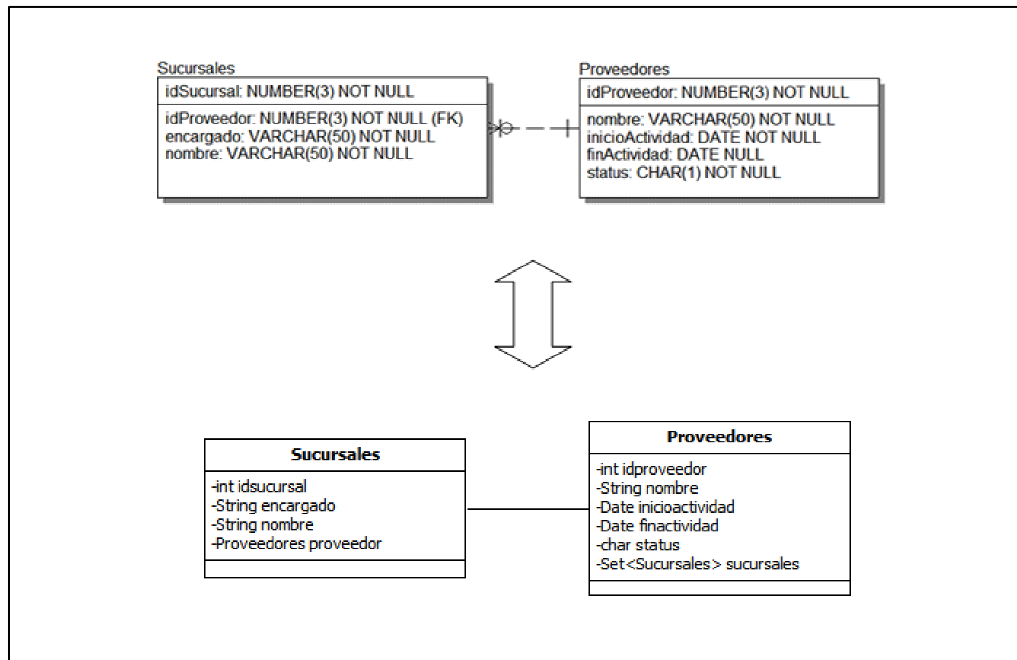


Figura 2.12: Mapeo de una relación `@OneToMany` bidireccional

`Proveedores.class`

```
1     @Entity
2     public class Proveedores {
3       @Id
4       @GeneratedValue(generator="secuencia2",strategy=GenerationType.SEQUENCE)
5       @SequenceGenerator(name="secuencia2",  sequenceName="secuencia_pv",
6       initialValue=1, allocationSize=1)
7       private int idproveedor;
8       @Temporal(TemporalType.DATE)
9       private Date inicioactividad;
10      private String nombre;
11      private char status;
12      @Temporal(TemporalType.DATE)
13      private Date finactividad;
14      @OneToMany(mappedBy="proveedor",fetch=FetchType.LAZY)
15      private Set<Sucursales> sucursales = new HashSet<Sucursales>();
16      //métodos getter y setter
17    }
```

Código 2.15: Mapeo entre `Proveedores` y `Sucursales`

Sucursales.class

```
1    @Entity
2    public class Sucursales{
3      @Id
4      @GeneratedValue(generator="secuencia3",strategy=GenerationType.SEQUENCE)
5      @SequenceGenerator(name = "secuencia3", sequenceName = "secuencia_suc",
6      initialValue = 1, allocationSize = 1)
7      private int idsucursal;
8      @ManyToOne(fetch = FetchType.LAZY)
9      @JoinColumn(name = "idproveedor")
10     private Proveedores proveedor;
11     private String encargado;
12     private String nombre;
13     //métodos getter y setter
14   }
```

Código 2.15 (*continuación*)

La asociación está definida como una relación bidireccional, pues en ambas clases se declara una referencia hacia la otra. De las líneas 4 a 6 en ambas clases, se define como estrategia de creación de llaves primarias una secuencia que comenzará en 1 e irá incrementándose a paso de 1.

La clase `Proveedores` define una colección `Set` de objetos `Sucursales` por medio de la asociación uno a muchos (un proveedor podrá tener muchas sucursales) en las líneas 14 y 15.

Por su parte, `Sucursales` define en la línea 8 la relación que guardará con `Proveedores` (una o más sucursales pueden pertenecer a un sólo proveedor) y establece la clave foránea en la línea 9.

2.7 Spring

Hasta el momento los 2 frameworks analizados cumplen con el cometido de simplificar el desarrollo de aplicaciones web con las ventajas de modularidad, fácil mantenimiento, ahorro de líneas de código y eficiencia en el tiempo de desarrollo. Sin embargo, las ventajas podrían acrecentarse aún más si se incluye un framework más a la lista: Spring.

El principal objetivo de Spring es facilitar el desarrollo de aplicaciones creadas en Java por medio de la unión de sus componentes (ya sea creados por el usuario, provenientes de otros frameworks o bien provenientes del API de Spring). Una de sus mayores virtudes es que se trata de un framework no invasivo, esto es, que en ninguna clase de la aplicación se programará código de Spring.

Spring está compuesto de 7 módulos que no necesariamente son útiles para todas las necesidades de negocio por lo que el programador tendrá la libertad de emplear aquellos que puedan tener un impacto tangible sobre su aplicación. Estos módulos son mostrados a continuación:

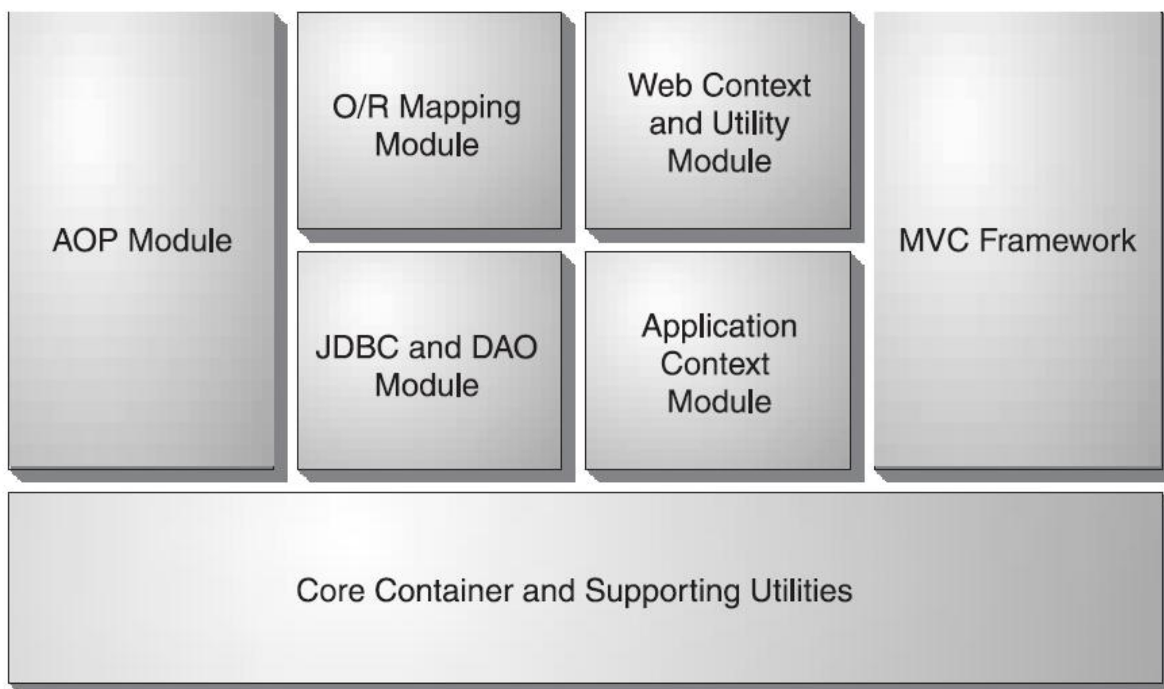


Figura 2.13: Arquitectura general de Spring

- **Core Container:** Spring también es conocido como un BeanFactory debido a que uno de sus roles fundamentales es la creación de objetos con las posibles dependencias que estos pudieran tener con otros. Este módulo es el encargado de proveer dicha funcionalidad a Spring siendo la base para cualquier aplicación que utilice el framework.

- **ApplicationContext:** Se trata de una subinterface de BeanFactory que se encarga de tener en memoria los beans declarados en el archivo de configuración `ApplicationContext.xml`.

- **AOP (Aspect-Oriented Programming):** Es una técnica que permite al programador separar la lógica de negocio con los aspectos técnicos de la aplicación, tales como logging, debugging y seguridad.

- **JDBC y DAO:** Spring ofrece un template o plantilla que contiene la funcionalidad del acceso a datos por medio de JDBC, que manejará la instanciación de los diferentes objetos derivados del API (`Connection`, `Statement`, `PreparedStatement` y `ResultSet`) así como el manejo de excepciones y liberación de recursos.

- **ORM:** Para aquellas aplicaciones que no hacen uso de JDBC para el manejo de datos sino que se basan en la implementación de un ORM que facilite esta tarea. Spring provee de un template o plantilla que soporta la interacción de sus objetos, liberación de recursos, manejo de excepciones y ejecución de transacciones, que unido, hace que el código responsable de la persistencia de datos se simplifique aún más.

- **Web:** Este módulo ofrece diversas tareas para aplicaciones web y contiene soporte para la integración con Struts.

- **MVC:** Spring también provee su propio framework basado en MVC (similar a Struts) que ayuda a la integración de este patrón de diseño en la aplicación.

2.7.1 Inyección de Dependencias

Uno de los retos a los que se debe enfrentar todo desarrollador de software es que el nivel de acoplamiento entre los objetos involucrados sea el más pobre posible. Una solución muy eficaz se encuentra en el uso de interfaces y en el concepto fundamental de Spring: La inyección de dependencias.

A través del siguiente caso de una clase que se encargará de sumar 2 números se mostrará cómo se logra tal nivel de acoplamiento usando Spring:

```
1   public class Suma{
2
3      public void operar(double num1, double num2) {
4          double resultado = num1 + num2;
5          System.out.println("El resultado es: " + resultado);
6      }
7   }
```

Código 2.16: Suma de 2 números

```
1    public class Calculo {
2
3       public void calcular() {
4           Suma suma=new Suma();
5           operación.operar(3.21,5.24);
6       }
7   }
```

Código 2.16 (*continuación*)

```
1    public class Main {
2
3       public static void main(String[] args) {
4            Calculo calculo = new Calculo();
5            calculo.calcular();
6        }
7    }
```

Código 2.16 (*continuación)*

En la clase Calculo se manda a llamar la clase concreta encargada de llevar a cabo la suma por medio de su método operar() que recibe los números a ser sumados. Sin embargo, el problema que se tiene es que la clase Calculo se encuentra muy acoplada a Suma. Debido a la instanciación directa de esta última en Calculo (línea 4).

Si en algún momento del desarrollo se pide que la aplicación no sume, sino que reste, se debe crear la nueva clase encargada de restar, así como modificar el tipo de objeto a instanciar, de Suma a Resta. Lo mismo tendría que suceder si en un futuro el requerimiento de la aplicación vuelve a cambiar y se pide multiplicación o división de 2 números.

Para solucionar este problema, se puede delegar a otro componente la implementación concreta de dicha clase.

```
1   public interface Operacion {
2       public void operar(double num1, double num2);
3   }
```

Código 2.17: Suma de 2 números usando Interfaces

```
1   public class Suma implements Operacion{
2       public void operar(double num1, double num2) {
3          double resultado = num1 + num2;
4          System.out.println("El resultado es: " + resultado);
5       }
6   }
```

Código 2.17 (*continuación)*

```
1   public class Calculo {
2       Operacion operacion;
3       public void setOperacion(Operacion operacion) {
4           this.operacion = operacion;
5       }
6       public void calcular() {
7           operacion.operar(1.23, 4.32);
8       }
9   }
```

Código 2.17 (*continuación)*

```
1   public class Main {
2
3       public static void main(String[] args) {
4           Operacion suma = new Suma();
5           Calculo calculo = new Calculo();
6           calculo.setOperacion(suma);
7           calculo.calcular();
8       }
9   }
```

Código 2.17(*continuación)*

La clase `Main` ahora será la encargada de instanciar a `Suma`, pero a través de la interface `Operacion` y será referida hacia `Calculo` por medio del método `setOperacion` definido en este el cuál recibirá la implementación de la interface que debe ser utilizada. De esta manera, `Calculo` sólo sabrá que debe hacer una operación, pero sin saber cuál. Pese a esta mejora, el código de `Main` sigue estando acoplado a la clase concreta de la operación a realizar y es aquí donde Spring comienza su trabajo.

El rol fundamental de Spring es la creación de los Beans que se encuentren en el archivo de configuración que define el contexto de la aplicación (`ApplicationContext`). Si se aprovecha esta funcionalidad, se puede delegar a Spring la creación de los distintos beans requeridos en esta aplicación.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns=http://www.springframework.org/schema/beans>
3
4       <bean id="suma" class="pruebas.Suma"/>
5       <bean id="calculo" class="pruebas.Calculo">
6           <property name="operacion"  ref="suma"/>
7       </bean>
8   </beans>
```

Código 2.18: Suma de 2 números en Spring

Los beans son declarados por medio de la etiqueta `bean` y sus atributos: `id` que recibe el nombre utilizado para identificar el bean y `class` que define la clase que lo contiene.

Los 2 objetos principales de esta aplicación son: `suma` y `calculo` por lo que son declarados en el XML de Spring en la línea 4 y 5. `calculo`, puede tener diferentes implementaciones de `Operacion` (en este caso suma) y a través del método `setOperacion` definido en él se indica cuál será. Para que Spring pueda realizar esto, se debe definir el objeto que contiene tan implemetanción. En la línea 6 con la etiqueta `property` se indica con el atributo `name` la propiedad que será referida a este por medio del método setter y la instancia que será utilizada con el atributo `ref`. En este caso: En el objeto `calculo` se define una instancia `suma` del atributo `operacion`. (`operacion` se encuentra definido en `Calculo` como una variable de clase).

```
1   public class Main {
2
3      public static void main(String[] args) {
4        ApplicationContext ac = new
5              ClassPathXmlApplicationContext("applicationContext.xml");
6         Calculo calculo = ac.getBean("calculo");
7         calculo.calcular();
8      }
9   }
```

Código 2.18 (*continuación*)

Finalmente en `Main` únicamente se manda a llamar el `applicationContext.xml` en la línea 4, y a partir de este, el bean `calculo` y como se observa, desaparece del código la definición de la operación concreta:

```
Operacion suma = new Suma();
Calculo calculo = new Calculo();
calculo.setOperacion(suma);
```

es equivalente a la siguiente delegación de Spring

```
<bean id="suma" class="pruebas.Suma"/>
<bean id="calculo" class="pruebas.Calculo">
    <property name="operacion"  ref="suma"/>
</bean>
```

} Inyección de Dependencias (DI)

Con esta última versión, se puede apreciar la ventaja que ofrece la programación con interfaces y más aún, si las implementaciones de estas son delegadas a Spring. Así si en algún momento el requerimiento de la operación cambia, sólo se debe modificar su implementación en el XML sin tener que recurrir al código para llevar a cabo la actualización.

2.7.2 Integrando Spring con Struts

La manera en que Struts trabaja con los objetos `Action`, por medio del archivo de configuración, permite que el rendimiento de la aplicación sea óptimo, debido a que son almacenados en una colección que contiene las referencias hacia estos y cuando una nueva petición es interceptada, Struts revisa que el `Action` solicitado se encuentre instanciado, de ser así, lo recupera y utiliza. En caso contrario, lo instancia y guarda una referencia de este para futuras solicitudes. No obstante, dicho rendimiento podría ser mejorado si se aprovecha el contenedor de Spring.

Cada `Action` invoca cierta cantidad de objetos necesarios para cumplir con la acción solicitada, es aquí donde el concepto de DI juega un papel importante en el desempeño de la aplicación, pues lo que se busca es que los `Action` sean manejados como beans e instanciados desde el momento que el contenedor arranca con sus respectivas dependencias de los objetos de negocio necesarios. Para lograr esta integración, se necesita llevar a cabo los siguientes puntos:

1. Registrar el `ContextLoaderPlugIn` en el archivo de configuración de Struts:
   Este plug-in es el responsable de cargar el archivo de configuración de Spring cuya ubicación se encuentra definida en el atributo `value` de la propiedad `contextConfigLocation`.

```
1   <plug-in
2        className="org.springframework.web.struts.ContextLoaderPlugIn">
3           <set-property property="contextConfigLocation"
4           value="/WEB-INF/applicationContext.xml"/>
5   </plug-in>
```

Código 2.19: Integración de Struts con Spring

2. Delegar el control de `Action`s a Spring:

   La clase `DelegatingActionProxy` será la responsable de cargar el plug-in de Spring para buscar en el `applicationContext` el `Action` que deberá procesar la petición.

```
1    <action-mappings>
2        <action name="UniformeForm" path="/UniformeAction"
3           type="org.springframework.web.struts.DelegatingActionProxy"
4           scope="request">
5            <forward name="verUniformes" path="/verUniformes2.jsp"/>
6        </action>
7     </action-mappings>
```

Código 2.19 (*continuación*)

   Originalmente en `struts-config.xml` se registraba, en el atributo `type`, la clase de acción correspondiente a la ruta que originaba la ejecución de esta. Con Spring, debido a que cada solicitud recibida deberá pasar por el proxy que la encaminará al `Action` correspondiente, este es el que debe ser registrado en cada etiqueta `action`.

3. Registrar las clases `Action` en el `applicationContext`:

   Para que `DelegatingActionProxy` sepa a qué `Action` ceder el trabajo de la petición, este último debe estar configurado en el `applicationContext` como un bean con sus respectivas dependencias y sus métodos `set` dentro de la clase `Action`.

```
1    <bean name="/UniformeAction" class="Action.UniformeAction">
2        <property name="LUniformes" ref="LUniformes">
3        </property>
4    </bean>
```

Código 2.19 (*continuación*)

2.7.3 Integrando Spring con Hibernate

Spring, en su módulo de ORM, ofrece un *template* que facilita y optimiza el uso de objetos `Session` de hibernate. Las ventajas que este ofrece son:

- Manejo transparente de excepciones y liberación de recursos.
- Uso automático de transacciones.
- Evita instanciar objetos `Session` con la implicación de abrir y cerrar conexiones.
- Reducción (aún más) del código de acceso a datos.

La clase `HibernateTemplate` es la encargada de llevar a cabo estas acciones de Hibernate a través de Spring. La clase de acceso a datos debe heredar su funcionalidad de `HibernateDaoSupport` para que le suministre un objeto `HibernateTemplate` por medio de su método `getHibernateTemplate()`. La forma en que se integra a la aplicación es la siguiente:

1. Configurar la fuente de datos:
   Spring elimina la necesidad del archivo de configuración de Hibernate debido a que desde el `applicationContext.xml` pueden ser declarados todos aquellos parámetros necesarios para la conexión con la base de datos. Como primer instancia se necesita configurar el bean del Data Source con sus valores de conexión.

```
1    <bean id="dataSource"
2    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
3        <property name="driverClassName" value="oracle.jdbc.OracleDriver"/>
4        <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE"/>
5        <property name="username" value="hugo"/>
6        <property name="password" value="hugo"/>
7    </bean>
```

Código 2.20: Integración de Hibernate con Spring

La clase `DriverManagerDataSource` de la línea 2 es la encargada de gestionar la conexión por medio de JDBC.

2. Registrar el `SessionFactory`:

   Spring provee de una clase capaz de crear el `SessionFactory` sin tener que hacer uso del `HibernateUtil` (código 2.10): `LocalSessionFactoryBean` para aquellas aplicaciones que realizan su mapeo por medio de archivos XML o bien, una subclase de esta, `AnnotationSessionFactoryBean` para las aplicaciones que los realizan a través de anotaciones. La clase indicada deberá ser configurada en el `applicationContext.xml` inyectándole el dataSource, el mapeo de las clases persistentes y de ser necesario, algunos parámetros extra propios de Hibernate (por ejemplo el dialecto al que se deben traducir todas las peticiones de Hibernate a la base de dato, y la opción de poder mostrar las secuencias sql que Hibernate genera y envía al servidor).

```
1    <bean id="sessionFactory"
2    class="org.springframework.orm.hibernate3.annotation.
3    AnnotationSessionFactoryBean">
4        <property name="dataSource" ref="dataSource"/>
5        <property name="annotatedClasses">
6            <list>
7                <!-- Mapeos -->
8            </list>
9        </property>
10       <property name="hibernateProperties">
11          <props>
12              <prop key="hibernate.dialect">
13                     org.hibernate.dialect.OracleDialect
14              </prop>
15              <prop key="hibernate.show_sql">true</prop>
16           </props>
17       </property>
18    </bean>
```

Código 2.20 (*continuación*)

En los códigos 2.12 y 2.13 se mostró cómo se persisten los datos de y hacia la base de datos respectivamente haciendo uso de Hibernate, a continuación, se vuelven a presentar estos dos escenarios con la integración de Hibernate y Spring.

```
1    public List<Proveedores> mostrarProveedores() {
2           List<Proveedores> proveedores;
3           proveedores = getHibernateTemplate().find("from Proveedores");
4           return proveedores;
5     }
```

Código 2.21: Persistencia de los objetos `Proveedores` desde la base de datos con Hibernate y Spring

El método `find()` del `HibernateTemplate` sustituye al `createQuery()` del objeto `Session`, y ahora este *template* es el encargado de instanciar estos objetos.

```
1   public void guardarProveedores (Proveedores pv) {
2         getHibernateTemplate().save(pv);
3   }
```

Código 2.22: Persistencia de los objetos `Proveedores` hacia la base de datos con Hibernate y Spring.

Para persistir el objeto hacia la base de datos únicamente se llama al método `save()` del *template* y será el que confirme la transacción en la base de datos.

2.7.4 Manejo de Transacciones

En toda aplicación, ya sea web o de escritorio, es vital que se garantice alta integridad y consistencia en los datos, por tal motivo, es indispensable el uso de transacciones para que esto sea posible.

Una transacción es un grupo de instrucciones relacionadas en un aspecto de la lógica de negocio que deben ser ejecutadas en su totalidad con éxito y en caso de falla, ninguna de estas se verá reflejada en la Base de Datos. El comportamiento esperado de una transacción se describe a partir de sus propiedades:

1. Atomicidad: Todas las instrucciones de la transacción son vistas como una sola que no puede ser dividida. Los cambios surgidos desde la primera hasta la última instrucción deben realizarse con éxito (commit) o bien abortarse en caso de que alguna de estas falle (rollback).
2. Consistencia: Una vez que la transacción haya finalizado, los datos deben permanecer en una manera consistente, esto es que ninguna de sus restricciones de integridad sea violadas.
3. Aislamiento: Las transacciones pueden ser lanzadas en forma paralela y estas no deben ser afectadas por la acción de otras que corran sobre el mismo conjunto de datos. Existen varios niveles de aislamiento: Desde uno completo, en el que 2 ó más transacciones que sean ejecutadas en un mismo dominio de datos provoque el bloqueo de estos y no podrán ser utilizados hasta que la transacción actual haya finalizado, hasta niveles más bajos en los que varias transacciones pueden trabajar sobre un mismo conjunto de datos a la vez.
4. Durabilidad: Una vez que la transacción fue completada de manera exitosa, los datos deben residir de manera permanente en la base de datos.

2.7.4.1 La anotación `@Transactional`

Esta anotación es utilizada para marcar un método como transaccional, puede ser declarada a nivel interface, en donde cada método implementado será ejecutado bajo la definición transaccional o bien a nivel método, en donde específicamente el método marcado con la anotación será transaccional.

Propiedades:

1. `Propagation`: Es la asociación de la transacción con el método en ejecución. En otras palabras, determina si en el método actual se crea una nueva transacción, se trabaja sobre una ya existente, o no se requiere transacción. Sus opciones son las siguientes:
   - `PROPAGATION_REQUIRED`: Es el valor por defecto, en caso de no ser especificado, indica que el método debe ser ejecutado como parte de una transacción. Si no existe la crea, de lo contrario corre sobre una ya existente.
   - `PROPAGATION_SUPPORTS`: Indica que el método no requiere una transacción, pero en caso de existir, esta será utilizada.
   - `PROPAGATION_MANDATORY`: Indica que el método requiere necesariamente una transacción y en caso de no existir, lanzará una excepción.
   - `PROPAGATION_REQUIRES_NEW`: Una transacción nueva debe comenzar cada que el método marcado con esta opción sea ejecutado. Si existe una, es suspendida.
   - `PROPAGATION_NOT_SUPPORTED`: Indica que el método no debe ser parte de ninguna transacción, de existir, esta es suspendida y retomada una vez que el método finaliza.
   - `PROPAGATION_NEVER`: Indica que el método no debe ser ejecutado como parte de una transacción, si existe, lanzará una excepción.

2. `Isolation`: Cuando se trabaja con transacciones existen 3 tipos de problema que se pueden encontrar debido a la concurrencia de estas sobre los datos.
   - Lectura sucia: Ocurre cuando una transacción lee datos que no han sido confirmados (commit).
   - Lectura no repetitiva: Ocurre si dentro de una transacción se leen datos diferentes para una misma consulta enviada más de una vez, debido a que otra transacción modificó los datos.
   - Lectura fantasma: Ocurre cuando una transacción envía una consulta más de una vez y las filas retornadas no coinciden debido a que otra transacción insertó nuevos datos que coinciden con el criterio de búsqueda de la consulta.

Para evitar estos problemas, se configura esta propiedad de aislamiento con sus siguientes opciones:

- `ISOLATION_READ_UNCOMMITED`: Es el nivel más bajo de aislamiento en el que los problemas previamente descritos pueden ocurrir. No es recomendable para operaciones críticas, sin embargo, con esta opción de aislamiento el rendimiento de la aplicación es mayor.
- `ISOLATION_READ_COMMITED`: Este nivel no permite que se hagan lecturas de datos a los que no se ha hecho commit, pero permite que estos sean modificados por otras transacciones.
- `ISOLATION_REPETABLE_READ`: Este nivel tiene las características de `READ_COMMITED` y además no permite que los datos manejados por una transacción sean modificados.
- `ISOLATION_SERIALIZABLE`: Es el nivel más alto de aislamiento de una transacción, en el que además de las condiciones de `REPETABLE_READ`, no permite que se haga inserción de datos que cumplan con el criterio de búsqueda de la consulta en una transacción en curso.

En la siguiente tabla se muestran los problemas que pueden ocurren por cada grado de aislamiento

	READ_UNCOMMITED	READ_COMMITED	REPETABLE_READ	SERIALIZABLE
Lectura Sucia	✓	×	×	×
Lectura no repetitiva	✓	✓	×	×
Lectura fantasma	✓	✓	✓	×

Tabla 2.18: Comparativa entre los posibles niveles de aislamiento en transacciones

3. `read-only`: Indica si la transacción debe ser manejada como de sólo lectura. Hibernate provee una optimización cuando esta propiedad es marcada como `true`.

4. `rollbackFor`: Indica el tipo de excepción, que al ser lanzada, provocará el rollback de la transacción. Su contraparte es la opción `noRollbackFor`.

2.7.4.2 Configuración

Para comenzar la configuración, se debe elegir la estrategia que utilizará Spring para el manejo de las transacciones.

Cada API de acceso a datos en Java contiene su propia definición para el manejo de transacciones. Spring en su API, cuenta con interfaces que abstraen estas definiciones y las implementa para ser utilizadas en los diferentes casos. Para Hibernate 3 se debe utilizar la clase `org.springframework.orm.hibernate3.HibernateTransactionManager`.
El `transactionManager` debe obtener un objeto `sessionFactory` que le permita realizar la transacción. Por consiguiente, en el archivo de configuración de Spring se inyecta una dependencia de este al `transactionManager`.

```
1    <bean id="transactionManager"
2    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
3             <property name="sessionFactory"  ref="sessionFactory"/>
4    </bean>
```

Código 2.23: Configuración de transacciones.

Finalmente, en caso de emplear anotaciones para el manejo de transacciones, se le debe indicar a Spring su uso por medio de la etiqueta `<tx:annotation-driven>` indicándole la referencia del bean que gestiona las transacciones (`transactionManager`).

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

Código 2.23 (*continuación*)

La interacción de los objetos tras la integración de los 3 frameworks (Struts, Hibernate y Spring) se muestra en la siguiente figura

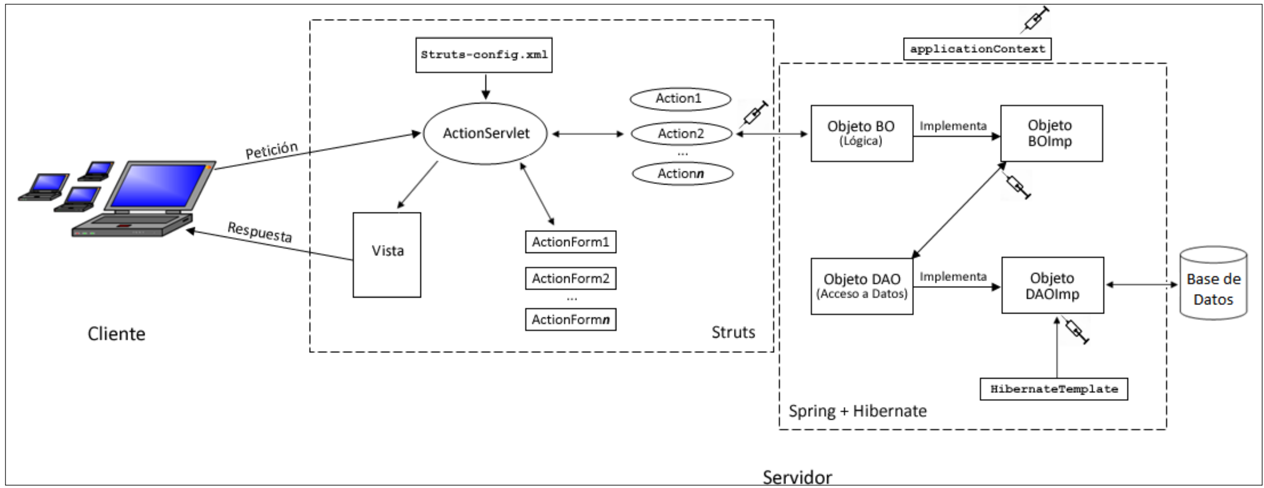


Figura 2.14: Intregración de Struts, Hibernate y Spring.

3

Implementación

3.1 Implementación del Cliente

3.1.1 Hojas de estilo

La definición de la interfaz gráfica del sistema se encuentra centralizada en 2 hojas de estilo: `estilosAdmon.css` y `estilosCliente.css`, las cuales contienen los elementos requeridos para la vista del trabajador y del cliente respectivamente. En la siguiente tabla se muestran estos elementos (similares para ambas hojas pero con definiciones diferentes).

Elemento	Tipo	Descripción
body	Etiqueta (<>)	Cuerpo de las páginas JSP, en él se define la imagen de fondo.
a		Vínculos requeridos en el sistema.
CContenedor	Id (#)	Capa principal de la interfaz que contiene a las demás (contenedor externo).
CTitulo		Capa externa superior que muestra la hora, la sesión y el logo.
CCentro		Capa externa en que se cargan los datos a través de la navegación del sistema.
CMenu		Capa externa que contiene las opciones de menú.
MenuMP		Capa externa que contiene el menú oculto de la Materia Prima.
MenuProd		Capa externa que contiene el menú oculto de las producciones.
Logo		Capa externa que contiene el logo de la PyME.
Sesion		Capa externa que contiene los datos de la sesión actual.
Hora		Capa externa que muestra el reloj.
Contenedor		Capa principal dentro de la navegación en CCentro (contenedor interno).

Tabla 3.1: Elementos de `estilosAdmon.css` y `estilosCliente.css`

Elemento	Tipo	Descripción
Titulo	Id(#)	Capa que contiene el título en el contenedor interno.
Izquierda		Capa que posiciona a la izquierda los datos en el contenedor interno.
Derecha		Capa que posiciona a la derecha los datos en el contenedor interno.
Centro		Capa que centra los datos en el contenedor interno.
Regresar		Capa estática que contiene la redirección a la página anterior (cuando es necesario).
Letra1	Clase (.)	Letra utilizada en las capas externas.
LBotones		Letra utilizada para las tablas.
LTitulo		Letra utilizada en los títulos de cada página.
LSubtitulo		Letra utilizada en caso de ser requeridos subtítulos.
LTexto		Letra utilizada para texto en general.
Campos		Define el estilo de los campos de comunicación con el usuario.
Botones		Define el estilo de los botones.
Botones:hover		Define el estilo de los botones cuando el mouse pasa encima de estos.
BotonDisabled		Define el estilo de los botones están inhabilitados.
LTrans		Letra pequeña utilizada en las tablas que tienen muchos datos.
label.error [1]		Letra que marca algún error en la validación de datos.
input.error, select.error,textarea.error [1]		Define el estilo que se le dará a los campos cuando contengan errores de validación.

Tabla 3.1 (continuación)

1. Elementos necesarios para el plugin `Validate.js`

Elemento	Tipo	Descripción
altaConsulta	Clase (.)	Tabla utilizada para consultas y alta de datos.
tablaEstatica		Tabla que contiene datos de reportes.
tablaEstatica thead		Encabezado de tablas estáticas
tablaEstatica tbody tr:nth-child(even)		Los renglones pares de las tablas.
tablaEstatica tbody tr:nth-child(odd)		Los renglones impares de las tablas.
ul.pageNav li[2]		Estilo de los números en la paginación.
ul.pageNav li a[2]		Estilo de los vínculos en la paginación.
li.currentPage[2]		Estilo del número de la página actual en la paginación.
ul.pageNav li.currentPage a[2]		Estilo del link de la página actual en la paginación.
pager[2]		Capa que contiene los datos de la paginación

Tabla 3.1 (continuación): elementos de `estilosAdmon.css` y `estilosCliente.css`

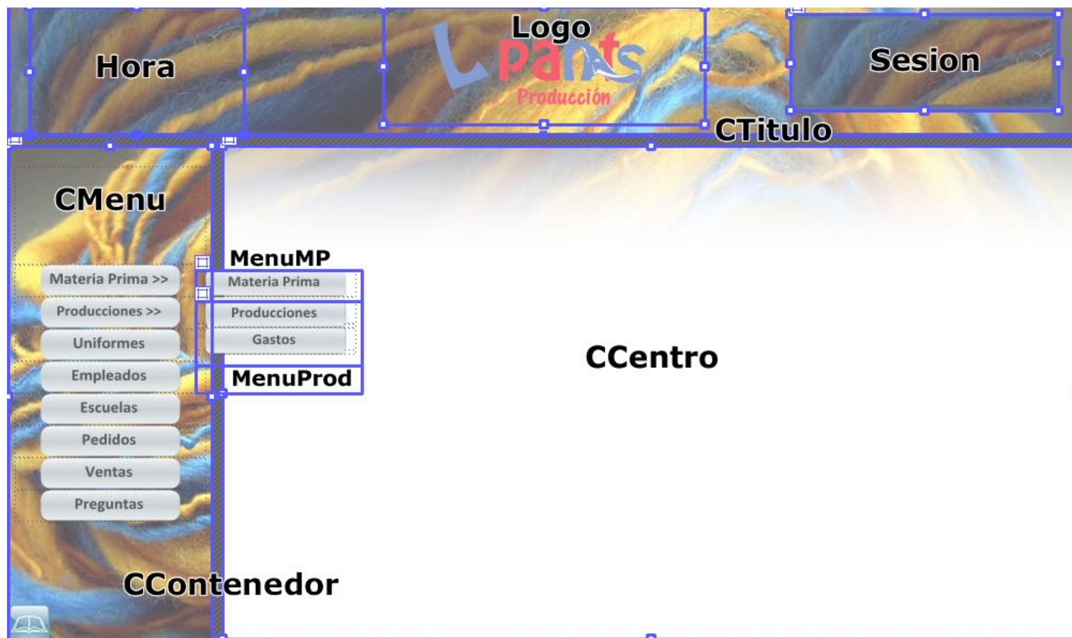


Figura 3.1: Estructura de las capas externas de la interfaz de usuario

2. Elementos necesarios para el plugin `QuickerPage.js`

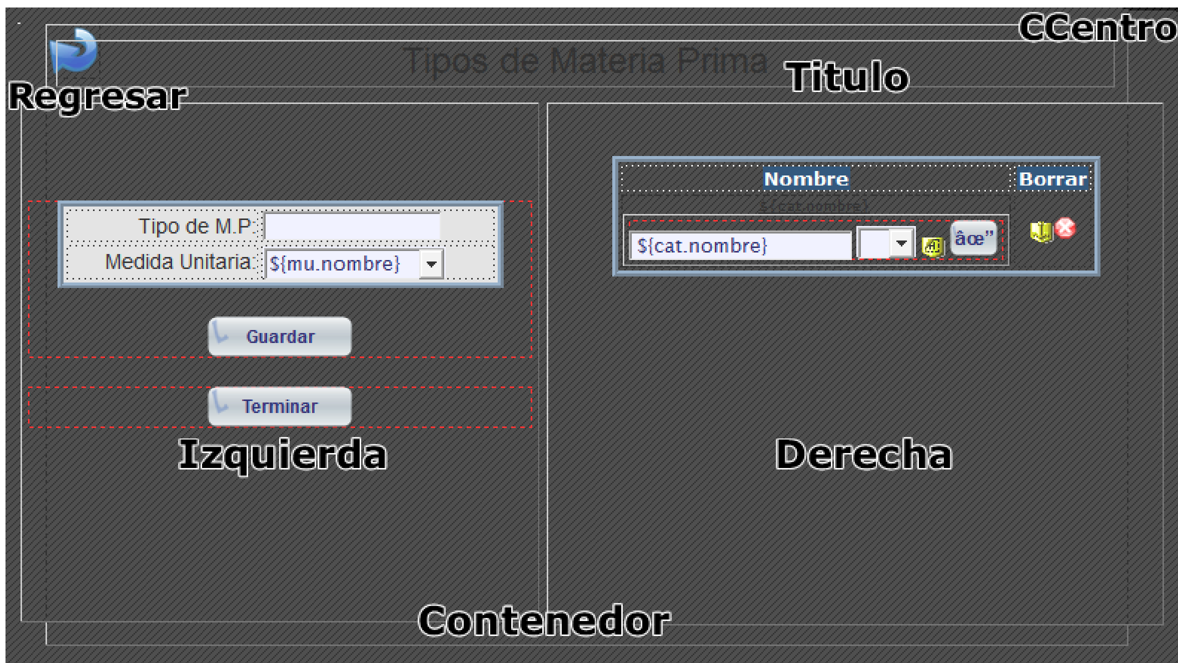


Figura 3.2: Estructura de las capas internas de la interfaz de usuario

3.1.2 jQuery

Las funciones de jQuery fueron divididas en cuanto a su funcionalidad y área de aplicación en diferentes archivos para ser cargadas por las páginas JSP cuando se requiera. A continuación se enlistan los archivos `.js` con su contenido:

1. funcionesP.js y funcionesC.js

Contienen las funciones generales del área de Producción y Ventas así como la del Cliente, con sus respectivas implementaciones. Contiene las siguientes funciones:

Función	Descripción
cargaPaginaLoad()	Carga en CCentro la página solicitada en el enlace linkExt
muestraSubmenuMP()	Muestra el submenú de la Materia Prima cuando el mouse entra al área indicada.

Tabla 3.2: Contenido de `funcionesP.js` y `funcionesC.js`

Función	Descripción
ocultaSubmenuMP()	Oculta el submenú de la Materia Prima cuando el mouse sale del área indicada.
muestraSubmenuP()	Muestra el submenú de las Producciones cuando el mouse entra al área indicada.
ocultaSubmenuP()	Oculta el submenú de las Producciones cuando el mouse sale del área indicada.
atenua()	Atenúa los botones del menú principal cuando entra el mouse al área.
quitaAtenuacion()	Regresa a su color original los botones del menú principal cuando el mouse sale del área.
$(document).bind("contextmenu",function(e){ })	Deshabilita el uso del botón derecho del ratón.

Tabla 3.2 (*continuación*)

2. funcionesAjax.js

Contiene en su mayoría las funciones que hacen uso de ajax tanto para alguna de las áreas de aplicación de la PyME así como para el cliente. Su contenido se muestra a continuación.

Función	Descripción
convierteAMys()	Convierte a mayúscula cada letra introducida en algún campo una vez que la tecla termina de ser presionada.
detallesMostrarOcultar()	Muestra u oculta de las capas la información contenida en algunas tablas.
cargaPaginaLoad()	Carga en CCentro la página solicitada en el enlace linkInt.
cargaPaginaAjax()	Carga en CCentro la información solicitada a través de un formulario.

Tabla 3.3: Contenido de `funcionesAjax.js`

Función	Descripción
obtenMunicipios()	Llena el combo de los municipios de acuerdo al estado seleccionado.
obtenSucursales()	Llena el combo de las sucursales de acuerdo al proveedor seleccionado.
obtenMP()	Llena el combo de la materia prima de acuerdo al tipo seleccionado.
obtenNiveles()	Llena el combo de los niveles de con que cuenta la escuela seleccionada.
obtenTipoUniforme()	Llena el combo del Uniforme para la combinación dada de escuela, nivel y grado.
obtenTallas()	Llena el combo de las tallas de acuerdo al uniforme seleccionado.
obtenPrecioActual()	Obtiene el precio actual de la talla seleccionada
obtenGrados()	Llena el combo de los grados de acuerdo al nivel seleccionado.
cuentaCaracteres()	Limita el número de caracteres introducidos en un área de texto.
confirmaBorrado()	Envía la confirmación de borrado de datos al usuario antes de ser procesada la petición.
confirmaEntrega()	Envía la confirmación de entrega de un uniforme pedido.
confirmacionPositiva(var acepta, var pagina)	Elige la tarea a hacer una vez que se preguntó por la confirmación.
deshabilitarBoton()	Deshabilita un objeto del tipo submit.
habilitarBoton()	Habilita un objeto del tipo submit.
$(".modificar").each(function(){ })	Envía la confirmación de modificación de datos antes de ser procesada por el servidor.

Tabla 3.3 (*continuación*)

Función	Descripción
$(".validar").each(function(){ })	Valida los campos del formulario antes de ser procesados por el servidor.
$(".paginacion tbody.pagina").quickPager({ })	Lleva a cabo la paginación del cuerpo de la tabla clasificada como *paginacion*.
$('#f1,#f2').datepicker({ })	Configura el calendario para los campos con id f1 y f2 que hacen uso de fechas.

Tabla 3.3 (*continuación*)

3. login.js

Debido a que el vínculo de Inicio de sesión y los datos de sesión se encuentran en su propia capa (Sesion), esta funcionalidad debe ser separada de los datos que son manejados en la capa CCentro.

Función	Descripción
loginLoad()	Lleva a cabo algunas tareas de la sesión especificadas en el vínculo seleccionado.
loginAjax()	Lleva a cabo el inicio de sesión y carga los datos en la capa de Sesion.

Tabla 3.4: Contenido de `login.js`

4. autocompletar.js

Con el fin de agilizar el registro de pedidos de usuarios registrados, fue implementado un *auto-completador* que muestra los clientes que coinciden con las letras que se van introduciendo en el campo de búsqueda.

Función	Descripción
controlaLista(e)	Envía al servidor un criterio de búsqueda diferente cada que el usuario introduce una nueva letra y crea una lista con los resultados coincidentes.

Tabla 3.5: Contenido de `autocompletar.js`

Función	Descripción
ocultaLista()	Cuando se da click en el campo de búsqueda se oculta la lista de elementos que coinciden con el criterio.
seleccionaItem()	Una vez que el elemento buscado es mostrado en la lista y se da click sobre este, el valor aparece en el campo de búsqueda para continuar con la tarea.

Tabla 3.5 (*continuación*)

5. reloj.js

Contiene únicamente el método `reloj()` que obtiene la hora del sistema, la muestra y asigna una imagen para cada etapa del día. Esta función es llamada desde `funcionesP.js` y `funcionesC.js` en un intervalo de 30 segundos.

3.1.3 Plugins de jQuery

Si bien jQuery facilita el uso de javascript para un desarrollo rápido y sencillo, las actuales demandas del web2.0 exigen mucho más de lo que este ofrece: Validación de datos sin necesidad de recargar la página, reproducción de audio y video, paginación de datos, subida de archivos al servidor por medio de ajax son sólo algunas de estas necesidades. Sin embargo, existe una infinidad de extensiones o plugins de código abierto que permiten al programador un ahorro considerable de código para implementar estas funcionalidades. Las extensiones utilizadas para el desarrollo del sistema fueron las siguientes:

3.1.3.1 QuickPager.js

Es muy común que al buscar información en una base de datos, esta sea presentada de manera que el usuario pueda verla de una forma cómoda, si el volumen de datos que se regresa es muy extenso, el usuario puede confundirse e incluso enfadarse por la información. Es por ello que la paginación es un recurso bastante utilizado en todos los sistemas que manejan una buena cantidad de datos. Este plugin contiene toda la lógica de la paginación para que sólo sea necesario

indicar la clase de la tabla a paginar y a través del método `quickPager()` se configura el tamaño de la página, el número de la página por defecto en que se comenzará a mostrar la información y la capa en la que se publicará la numeración.

```
1   $(".paginacion tbody.pagina").quickPager( {
2      pageSize: 8,
3      currentPage: 1,
4      holder: ".pager"
5   });
```

Código 3.1: Uso de `QuickPager.js`

3.1.3.2 Validate.js

Es imprescindible garantizar que los datos que llegan al servidor lo hagan en un formato correcto para evitar problemas posteriores en su manipulación y en la seguridad del sistema. Este plugin contiene validaciones para: números, dígitos, fechas, rangos, valores y por ser de código abierto, el programador puede aumentar las necesarias. Su uso resulta bastante sencillo: En el `form` se debe indicar la clase de la validación y colocar en cada campo el(los) criterio(s) de validación necesario(s) mostrados a continuación:

- required: Indica que el campo es obligatorio
- minlength(longitud): Longitud mínima aceptada para el campo.
- maxlength(longitud): Longitud máxima aceptada para el campo.
- min(valor): Valor mínimo aceptada para el campo.
- min(valor): Valor máximo aceptado para el campo.
- date( ): Formato de fecha (originalmente acepta en formato yyyy/MM/dd pero puede ser modificado a dd/MM/yyyy).
- number( ): Campo que acepta únicamente números.
- digits( ): Campo que acepta únicamente dígitos.
- equalTo(campo): El valor del campo debe coincidir con el de otro.
- letras(): Campo que admite únicamente letras (este criterio fue agregado debido a que no se encontraba definido).

En caso de error, crea una etiqueta `<label>` a un costado del campo e imprime el mensaje que debe ser mostrado al usuario (el mensaje es configurado directamente desde código en el plugin) y a su vez modifica el estilo del campo con error. El estilo del `label` y los campos con error debe ser configurado por el programador en una hoja de estilo.

```
1   <form class="validar" method="post"
2    action="/Uniformes/paginasCliente/PersonaAction.do">
3   <input name="nombre" type="text" class="Campos required letras"
4    minlength="10" maxlength="50" />
5   <input name="calle" type="text" class="Campos required letras"
6    minlength="5" maxlength="50" />
7   <input name="cp" type="text" class="Campos required digits"
8    minlength="5" value="${param.cp}"size="5" maxlength="5" />
```

Código 3.2: Validación de datos con `Validate.js`

```
1   $(.validar).validate()
```

Código 3.2 (*continuación*)

3.1.3.3 Datepicker.js

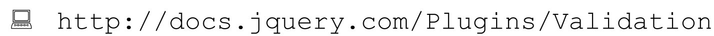
Es un plugin que muestra un calendario una vez que se da click en el campo destinado para las fechas. Contiene múltiples opciones de configuración tales como: El formato de fecha, la forma en qué será abierto el calendario (por medio de un botón extra o al posicionarse en el campo), el cambio del mes y del año, la velocidad en que aparece, el auto-dimensionamiento, la animación que tendrá al ser abierto, el conteo de semanas, el rango de fechas en que el usuario podrá seleccionar, entre otras.

Contiene, además del archivo js, un archivo css que define el estilo del calendario y que puede ser modificado por el programador. Por medio del método `datepicker()` se establecen las opciones de configuración de los campos f1 y f2 que serán los encargados de recoger las fechas.

```
1    $('#f1,#f2').datepicker({
2        dateFormat: 'dd/mm/yy',
3        changeMonth: true,
4        changeYear: true
5    });
```

Código 3.3: Uso de `DatePicker.js`

Para mayor información y documentación de estos plugins se pueden consultar las siguientes páginas:

- http://www.geckonewmedia.com/blog/
-  http://docs.jquery.com/Plugins/Validation
- http://docs.jquery.com/UI/Datepicker

Un problema que se puede observar hasta el momento es que Javascript se convierte en un elemento vital de la aplicación, si este se encuentra desactivado, el sistema no funcionará correctamente y se obtendrán resultados inesperados. Para evitar esto, se revisa que Javascript se encuentre activado con la etiqueta `<noscript>` de HTML, y en caso de no estarlo, se direcciona a una página que da muestra del error para que el usuario lo active (a través de las opciones de configuración de su navegador[3]) o cambie de explorador.

3. Para mozilla: Herramientas>Opciones>Contenido>Habilitar Javascript
   Para IE: Herramientas>Opciones de Internet>Seguridad>Nivel Personalizado>Active Scripting>Habilitar
   Para Opera: Configuración>Opciones>Avanzado>Contenido> Habilitar Javascript
   Para Chrome: Opciones>Avanzadas>Proxy>Seguridad>Nivel Personalizado>ActiveScripting>Habilitar

3.2 Implementación del Servidor

3.2.1 Jerarquía de directorios

Para que la aplicación web pueda funcionar de manera correcta, necesita de una serie de recursos que la complementen: Páginas estáticas HTML, páginas dinámicas JSP, archivos de configuración, hojas de estilo (CSS), servlets, clases compiladas, imágenes, etcétera. La especificación de Sun para las aplicaciones web define por medio de una jerarquía de directorios la forma en que los recursos deben ser situados para ser cargados por el servidor. Este conjunto de directorios puede ser empaquetado en lo que se conoce como un Archivo de Aplicación Web (WAR por sus siglas en inglés) o bien, ubicarlos en la carpeta `webapps` del servidor `tomcat`. A continuación se muestra esta jerarquía.

Directorio	Descripción
/	Se trata del directorio raíz de la aplicación (el nombre es definido por el programador) que contiene todas los subdirectorios con las páginas HTML, JSP, las hojas de estilo, código Javascript, imágenes, etcétera
/WEB-INF	Contiene todos los archivos de configuración utilizados en la aplicación (`web.xml`, `struts-config.xml`, `applicationContext.xml`, etcétera).
/WEB-INF/classes	Contiene las clases y JSP compilados.
/WEB-INF/lib	Contiene los ficheros Java (JAR) de las distintas librerías requeridas para la aplicación.

Tabla 3.6 Directorios estándar de aplicaciones web

3.2.2 Módulos

Struts permite la modularización de la aplicación a través de sus diferentes áreas funcionales para que estas puedan ser tratadas con independencia. Cada módulo requiere de su propio archivo de configuración, sus propios actions, forwards y JSPs. La manera en que se lleva a cabo esta acción es la siguiente:

1. Crear el archivo de configuración del módulo
2. Registrarlo en el descriptor de despliegue `web.xml`.
3. Configurar los accesos de los diferentes módulos en las JSPs.

1. Crear el archivo de configuración para cada módulo

Esta definición de un archivo de configuración por cada módulo permite que trabajen con independencia y los distintos objetos asociados puedan ser agrupados.

La aplicación está compuesta de 3 módulos: Producción, Ventas y Cliente.

2. Registrar cada archivo de configuración en `web.xml`

Los módulos son identificados de la siguiente manera:

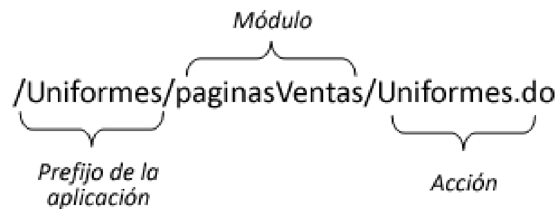


Figura 3.3: Configuración de los diferentes módulos en web.xml.

Y cada uno debe encontrarse debidamente registrado en el descriptor de despliegue como se muestra a continuación:

```
1    <servlet>
2        <servlet-name>action</servlet-name>
3        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
4        <init-param>
5            <param-name>config</param-name>
6            <param-value>/WEB-INF/struts-config.xml</param-value>
7        </init-param>
8        <init-param>
9            <param-name>config/paginasVentas</param-name>
10           <param-value>/WEB-INF/struts-configVentas.xml</param-value>
11       </init-param>
12       <init-param>
13           <param-name>config/paginasCliente</param-name>
14           <param-value>/WEB-INF/struts-configCliente.xml</param-value>
15       </init-param>
16   </servlet>
17   <servlet-mapping>
18       <servlet-name>action</servlet-name>
19       <url-pattern>*.do</url-pattern>
20   </servlet-mapping>
```

Código 3.4: Configuración de los diferentes módulos en `web.xml`.

En el parámetro `config` se registra el módulo por default (en este caso el de producción) junto con su ubicación. Para configurar los módulos restantes se debe especificar en este parámetro el nombre del módulo y el archivo que contiene las definiciones para este.

3. Configurar los accesos de los diferentes módulos en las JSPs.

Para indicar el módulo al que se debe dirigir una acción, este debe ser especificado en la URL (Ver figura 3.3). En caso de que no se detalle entrará a su módulo por defecto.

3.2.3 DispatchAction

Como se mencionó en la sección 2.4 (Struts), cada petición es interceptada por una subclase `Action` y la procesa en su método `execute()` que debe ser sobreescrito por el programador con la funcionalidad específica. Así, para cada petición debe existir un `Action` que sea capaz de atenderla. Por ejemplo:

Dar de alta un nuevo uniforme → NuevoUniforme.do
Ver un uniforme → VerUniforme.do
Borrar un uniforme → BorrarUniforme.do

¿Qué tienen en común estas tres acciones?
La lógica de negocio hacia la que van dirigidas es la misma: Los uniformes.

Struts dispone de una subclase de `Action` que permite procesar peticiones similares dentro de una misma clase en cada uno de sus métodos. Se trata de la clase `DispatchAction` que tiene implementado el método `execute()` por default y se encarga de determinar el método apropiado para procesar la petición por medio de un parámetro cuyo nombre es elegido por el programador y el valor de este debe coincidir con el nombre del método implementado que

deberá tener la misma anatomía del método `execute()`. Su funcionamiento se ejemplifica en la siguiente figura.

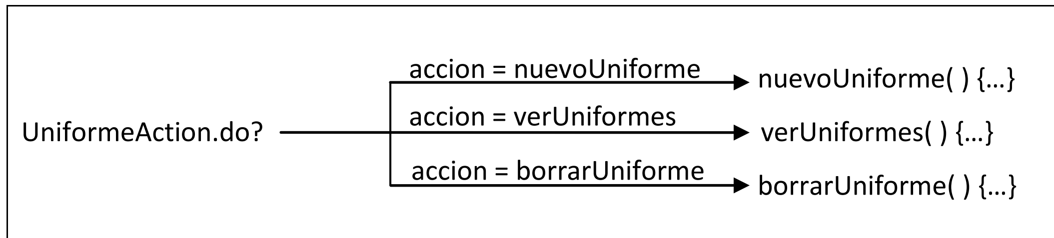


Figura 3.4: Funcionamiento de `DispatchAction`.

Donde:

4 UniformeAction: Es la subclase Action.

5 accion: Es el parámetro en el que se debe indicar el nombre del método que procesará la petición.

De esta forma, se crea una sola clase que maneje las diferentes peticiones en `n` métodos sin la necesidad de crear `n Actions` diferentes. Llevado a código el ejemplo anterior, se vería de la siguiente manera:

```
public class UniformeAction extends org.apache.struts.actions.DispatchAction {

    public ActionForward nuevoUniforme(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
            …
    }

    public ActionForward verUniformes(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
            …
    }

    public ActionForward borrarUniforme(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
            …
    }
}
```

Código 3.5: Ejemplo de subclase `DispatchAction`

La configuración de esta clase en `struts-config.xml` sólo debe contener, como extra, el nombre del parámetro que seleccionará el método indicado. Por lo demás, se hace de manera habitual.

```
1     <action name="UniformeForm" path="/UniformeAction"
2     type="org.springframework.web.struts.DelegatingActionProxy"
3     parameter="accion" scope="request">
4          <!-- Forwards -->
5     </action>
```

Código 3.6: Configuración de la subclase `DispatchAction` en `struts-config.xml`

3.2.4 Arquitectura de la aplicación

Con la finalidad de tener mayor control sobre el contenido de las clases que conforman el sistema. Estas se encuentran distribuidas en 5 paquetes:

1. Action: En este paquete se localizan todas las subclases `Action` (`DispatchAction` hereda a `Action`) que interceptarán la petición del usuario.
2. ActionForm: Contiene las subclases `ActionForm` que transportarán los datos del usuario.
3. Logica: Almacena las clases que se encargan de la lógica de negocio y que son llamadas por las clases contenidas en Action.
4. Persistencia: Contiene las clases que acceden a la información contenida en la base de datos por medio de Hibernate y se comunica con las clases del paquete de la Lógica.
5. Persistencia.Mapeos: Abarca las clases que conforman el modelo de dominio de la aplicación (mapeos de las tablas de la base de datos). Como se muestra en el diagrama de la figura 3.5.

La arquitectura completa de la aplicación se muestra en el diagrama de clases de la figura 3.6.

Figura 3.5: Modelo de Dominio de la aplicación

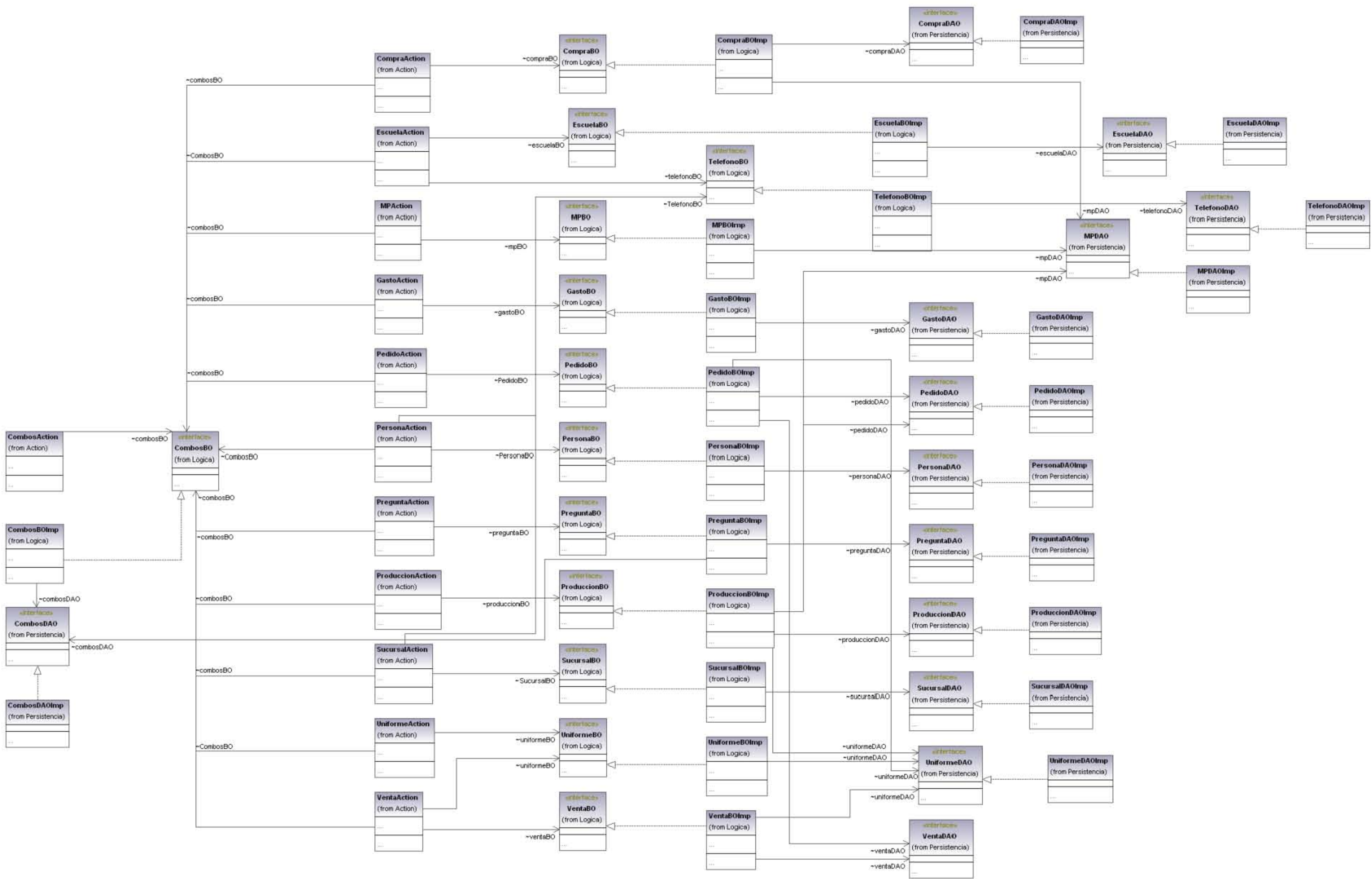


Figura 3.6: Arquitectura de la aplicación

3.2.5 Software involucrado en la construcción

Ambiente de Desarrollo	Herramientas de Diseño	Implementación
– Netbeans 6.9	– ERWin 4.0	– Tomcat
– Windows 7	– Photoshop CS2	– jQuery 1.4.2
– Firefox 3.6	– Visio 2007	– Oracle 11g R2
– Chrome 10	– Umodel 2011	– Struts 1.3.8
– Opera 10	– Fireworks MX	– Hibernate 3
– Internet Explorer 8	– Dreamweaver 8	– Spring 2.5
	– Button Shop 4	– Log4J 1.2
		– Kettle 3.1
		– Pentaho 3.5

Tabla 3.7: Software involucrado en la construcción

4

Toma de Decisiones

¿Pan o cereal?, ¿Jugo o leche?, ¿Corbata azul o verde?, ¿Carro o taxi? Son sólo algunas de las miles de decisiones que como ser humanos tomamos cada día en nuestra vida cotidiana. Muchas de estas las hacemos de manera irracional y otras tantas de manera analítica, pero lo que es común en todas es que, sin importar lo bueno o malo de nuestra elección, ésta influirá a lo largo del día, de la semana o quizá de nuestra vida.

Si lo anterior fuera extrapolado a una empresa, en la que día a día se toman decisiones que trazan el éxito o fracaso de esta en el mercado, la manera "irracional" queda inmediatamente descartada para este proceso, por el contrario, se requiere de un análisis basado en información detallada que ayude a justificar esa *última palabra* y quedar con la certeza de que fue la mejor decisión.

Para el caso de estudio de este trabajo, la mayoría de las decisiones se basan en el área de ventas: Se produce bajo demanda. Sin embargo, debido a que se trata de una pequeña empresa, con un número reducido de trabajadores dedicados a producción y una amplia demanda de uniformes, la producción para la temporada alta (julio-septiembre) se comienza hasta con medio año de anticipación, y debido a esto, no se tiene información precisa acerca de la demanda real de cierto tipo de uniforme por lo que algunos uniformes se agotan rápidamente mientras otros no son vendidos con la misma rapidez. Además de que, en los demás meses del año los uniformes siguen siendo requeridos. Algunas de las cuestiones importantes para la toma de decisiones son las siguientes:

- ¿Qué tallas son las más vendidas?
- ¿Qué tipo de uniforme es el más demandado?
- ¿En qué épocas del año crece y/o decrece la demanda de cierto uniforme?
- ¿En qué escuela se tiene un mayor consumismo?

4.1 Necesidad de un Cubo OLAP

Los sistemas OLTP se encuentran orientados a transacciones que soporten las principales tareas de negocio que se requieren a diario en donde las principales operaciones que se usan son: `INSERT`, `DELETE`, `UPDATE` y son concretadas una vez que se hace `commit` o bien abortadas cuando se hace `rollback`. Este tipo de sistemas no son muy útiles para llevar a cabo las tareas de análisis de datos, debido a que el modelo de datos se encuentra altamente normalizado, generalmente no

cuentan con un registro histórico y la ejecución de consultas que involucren grandes volúmenes de datos podrían entorpecer las transacciones.

Los sistemas OLAP (On-Line Analytical Processing) se encuentran orientados al procesamiento analítico, esto es, que su objetivo principal es la lectura de grandes volúmenes de datos que resulten útiles al usuario y así, con la información recabada, encaminarlo hacia la toma de decisiones inteligente.

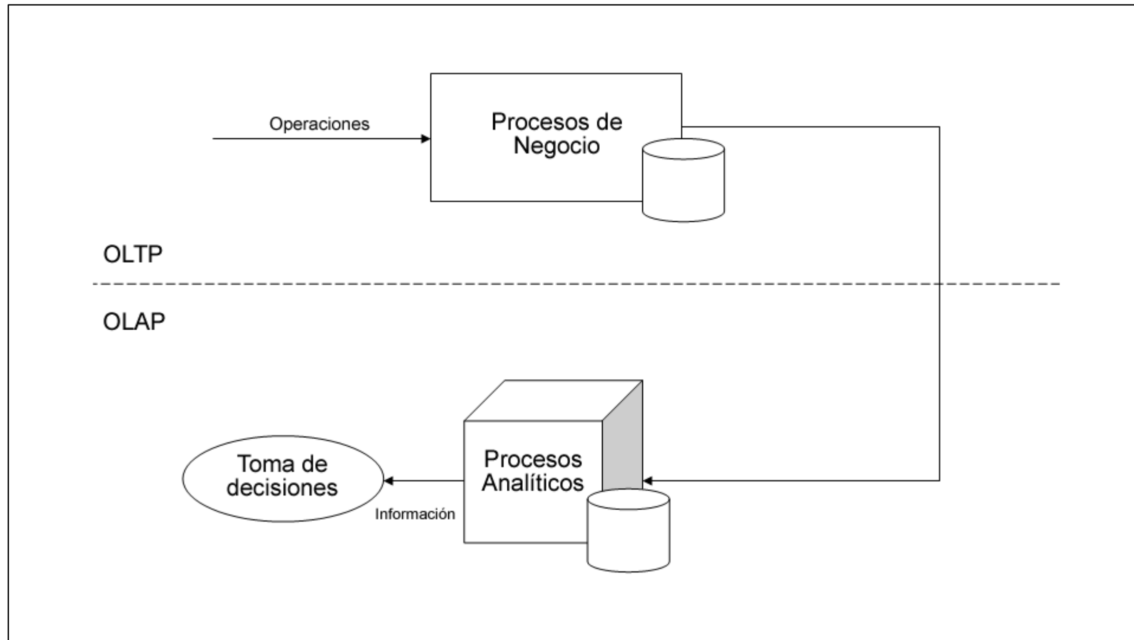


Figura 4.1: Sistemas OLTP y OLAP

Este tipo de sistemas, se basan en cubos, llamados así por la característica multidimensional de la base de datos que es utilizada para su construcción.

Un cubo OLAP, representa un conjunto de hechos relacionados con un área específica de negocio y cuya información se encuentra ordenada en vectores de n dimensiones (generalmente tres). La dimensión es la forma en que el tema objetivo es separado para su análisis, y generalmente responde a las preguntas de ¿Cuándo?, ¿Qué?, ¿Dónde?, etcétera. La intersección de las dimensiones constituyen un hecho.

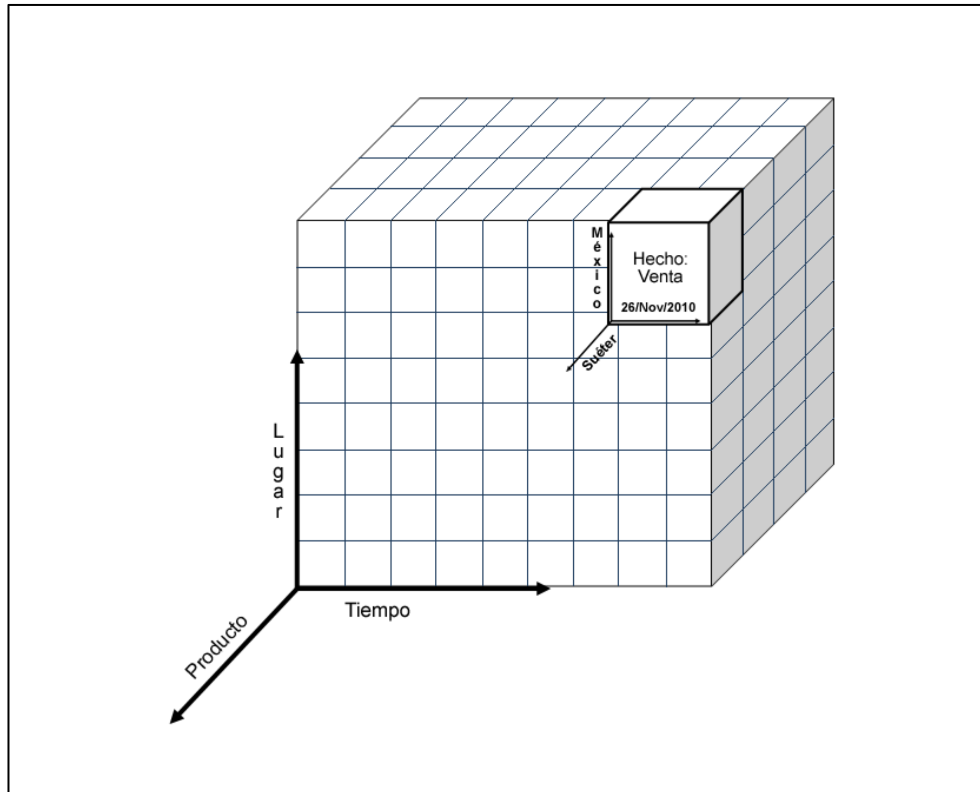


Figura 4.2 Visualización multidimensional del hecho venta

4.2 Componentes y diseño

Antes de diseñar un cubo es importante considerar los diferentes componentes que forman parte de éste:

- Tabla de hechos: Es la tabla central que recoge la información de las dimensiones por medio de las claves foráneas que formarán la llave primaria de esta. Adicionalmente deberá contener atributos, idealmente numéricos, que representen la información de cada hecho. Estos atributos son denominados medidas.
- Tablas de dimensiones: Estas tablas contienen el detalle de cada dimensión útiles para describir los hechos almacenados en el cubo.

Existen 2 maneras de diseñar una base de datos OLAP:

1. Basado en un esquema estrella: Se le llama así debido a que su estructura forma una estrella, con la tabla de hechos en el centro, y cada una de las dimensiones conectada a esta. Existe sólo una conexión por dimensión, no existen extensiones ni caminos alternativos entre estas.
2. Basado en un esquema copo de nieve: Es una variación del esquema anterior en el que las tablas dimensionales se encuentran normalizadas con la finalidad de eliminar la redundancia.

La decisión entre elegir uno y otro se debe basar en la cantidad de datos con que se trabajará y el diseño original de la aplicación OLTP tomando en cuenta las siguientes comparativas entre ambos esquemas:

Esquema estrella	Esquema copo de Nieve
Cada dimensión tiene un solo *join* hacia la tabla de hechos, por lo que las consultas se llevan a cabo de una manera más eficiente.	Incrementa el tiempo de ejecución de las consultas debido a la normalización de las dimensiones.
Este tipo de diagrama es más comprensible por su simplicidad en el diseño.	Generalmente un sistema OLAP nace del diseño OLTP en el que sus tablas se encuentran normalizadas, por lo que su implementación resulta más sencilla.
La migración de datos de OLTP a OLAP podría afectar el rendimiento del sistema OLTP si la transformación de datos resulta ser excesiva.	Las tablas dimensionales son pobladas de manera similar a como se encuentran en OLTP por lo que el rendimiento de este último no se vería muy afectado.

Tabla 4.1: Comparativa entre esquema estrella y copo de nieve

4.3 Implementación

Se pueden identificar 4 etapas principales en la construcción del almacén de datos:

1. Elegir el área de negocio para el cuál será construido
2. Decidir el hecho central
3. Identificar las distintas dimensiones

4. Elegir las medidas de negocio para la tabla de hechos.

Este cubo OLAP estará orientado al área de ventas, cuyo hecho principal es la venta de un uniforme formado por las dimensiones: Escuelas (¿Dónde?), Uniformes (¿Qué?) y Tiempo (¿Cuándo?)

Está diseñado bajo un modelo estrella para la simplificación de consultas OLAP, para obtener un tiempo óptimo respuesta y para que su diseño intuitivo sea más fácil de comprender para el usuario final.

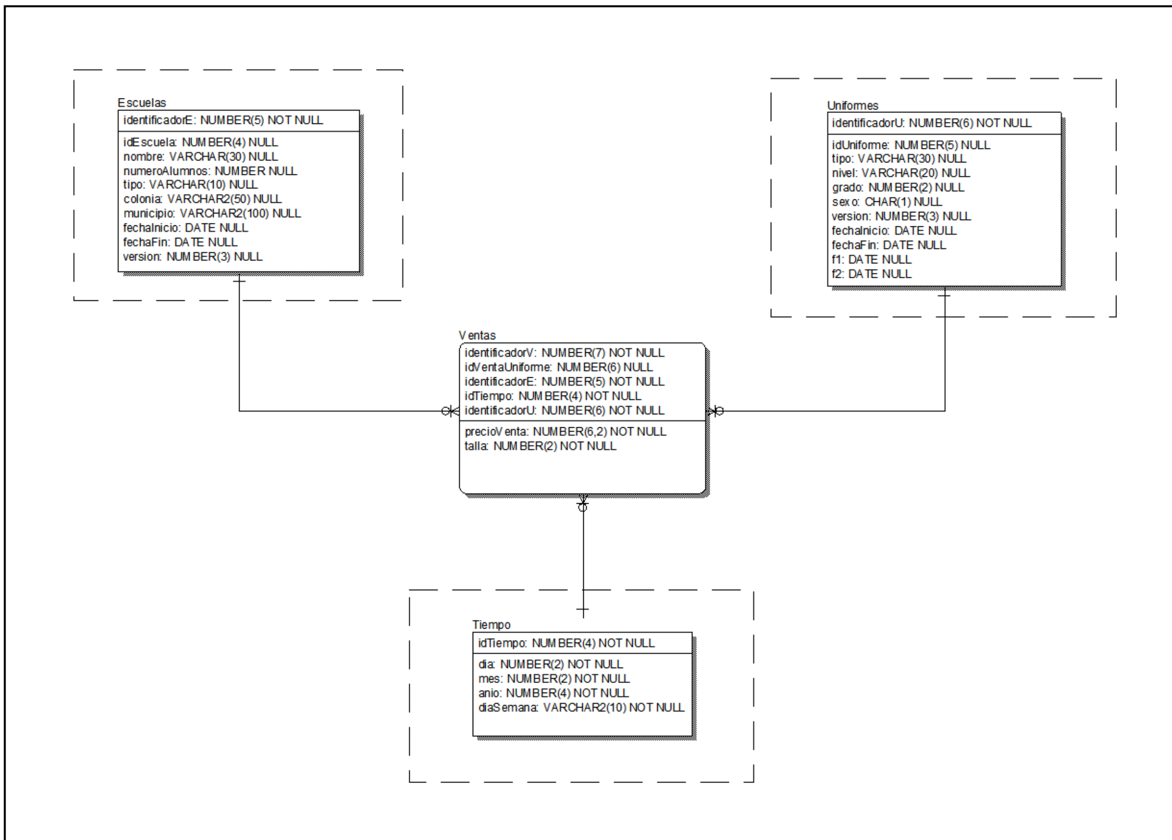


Figura 4.3: Diseño de cubo OLAP orientado a ventas

4.3.1 ETL (Extracción Transformación y carga)

Hasta el momento, sólo se ha diseñado la base de datos OLAP, sin embargo, esta no ha sido alimentada con los datos del negocio. Este proceso de migración de datos se le conoce como ETL (Extract, Transform, Load por sus siglas en inglés) y se divide en cada una de las actividades que conforma su nombre:

- Extracción: Comprende las tareas de obtener la información necesaria de las diferentes fuentes de datos que será integrada en el sistema OLAP, generalmente el origen de datos son sistemas OLTP, sin embargo, pueden provenir también de archivos de texto plano, hojas de cálculo e incluso otros sistemas OLAP.

- Transformación: En esta etapa se realizan las tareas que, independientemente del origen y formato de los datos, se convierten a un estado uniforme con un mismo formato defino. Algunas de estas tareas son:
  - Validación de datos: Se verifica que los datos extraídos cumplan con las restricciones correspondientes.
  - Limpieza de datos: Se realiza la corrección de datos erróneos o incompletos, y en caso de no contar con suficiente información de estos, puedes ser descartados y clasificados para darles tratamiento posterior.
  - Normalización de datos: Convierte los datos extraídos a valores más descriptivos y uniformes. La discretización (convertir un valor numérico a nominal) y numerización (convertir un valor nominal a numérico) son técnicas muy utilizadas en esta fase.
  - Agregación: En ocasiones, debido a las necesidades de negocio, es conveniente almacenar información agregada de ciertos valores numéricos, principalmente útil para la tabla de hechos.

- Carga: En esta etapa los datos que ya han sido extraídos y transformados de acuerdo a los requerimientos, son cargados en las diferentes tablas que forman las dimensiones y el hecho central.

4.3.2 Dimensiones de lenta variación (SCD)

En un sistema OLAP las dimensiones sufren cambios ocasionales a través del tiempo que deben ser actualizados en el sistema para una traducción real de hechos.

Para manejar esta actualización de datos existen tres formas principales de registrarlos en la base de datos de acuerdo a las necesidades de negocio:

1. SCD Tipo 1 – Sobreescritura: Cuando se detecta un cambio en alguno de los atributos de la dimensión, este se sobreescribe y no se guardan sus valores históricos.

id_subrogado	id_negocio	columna_1	columna_2
2	5	valor_1	valor_2

id_subrogado	id_negocio	columna_1	columna_2
2	5	valor_3	valor_2

Para que se puedan gestionar de manera correcta las actualizaciones de los datos, se requiere de dos claves en la tabla de la dimensión: El valor original de la clave primaria ubicada en el origen de datos y una clave subrogada que identifique la fila en el sistema OLAP. Esto con el fin de que se logren detectar los cambios por medio del identificador original y a través de la clave sustituta poblar los hechos de la tabla central.

2. SCD Tipo 2 – Nueva fila: Si se detecta un cambio en algún valor se crea una nueva fila con la actualización del atributo y un nuevo id subrogado, conservándose el valor anterior en una fila diferente.

id_subrogado	id_negocio	columna_1	columna_2	fecha_1	fecha_2	version
2	5	valor_1	valor_2	26/11/2010	NULL	1

id_subrogado	id_negocio	columna_1	columna_2	fecha_1	fecha_2	version
2	5	valor_1	valor_2	26/11/2010	24/12/2010	1
3	5	valor_3	valor_2	24/12/2010	NULL	2

Es importante considerar que el histórico se lleva a cabo por medio de 3 columnas extra: la fecha en que se dio de alta el registro, la fecha en que uno nuevo lo sustituyó, y el número de versión de ese registro.

3. SCD Tipo 3 – Nueva Columna: En esta estrategia se requiere una columna extra por cada atributo que se desee mantener su historial. La nueva columna almacenará el valor anterior del atributo en caso de ser actualizado.

id_subrogado	id_negocio	columna_1	columna_1_anterior	columna_2
2	5	-----	valor_1	valor_2

id_subrogado	id_negocio	columna_1	columna_1_anterior	columna_2
2	5	valor_1	valor_3	valor_2

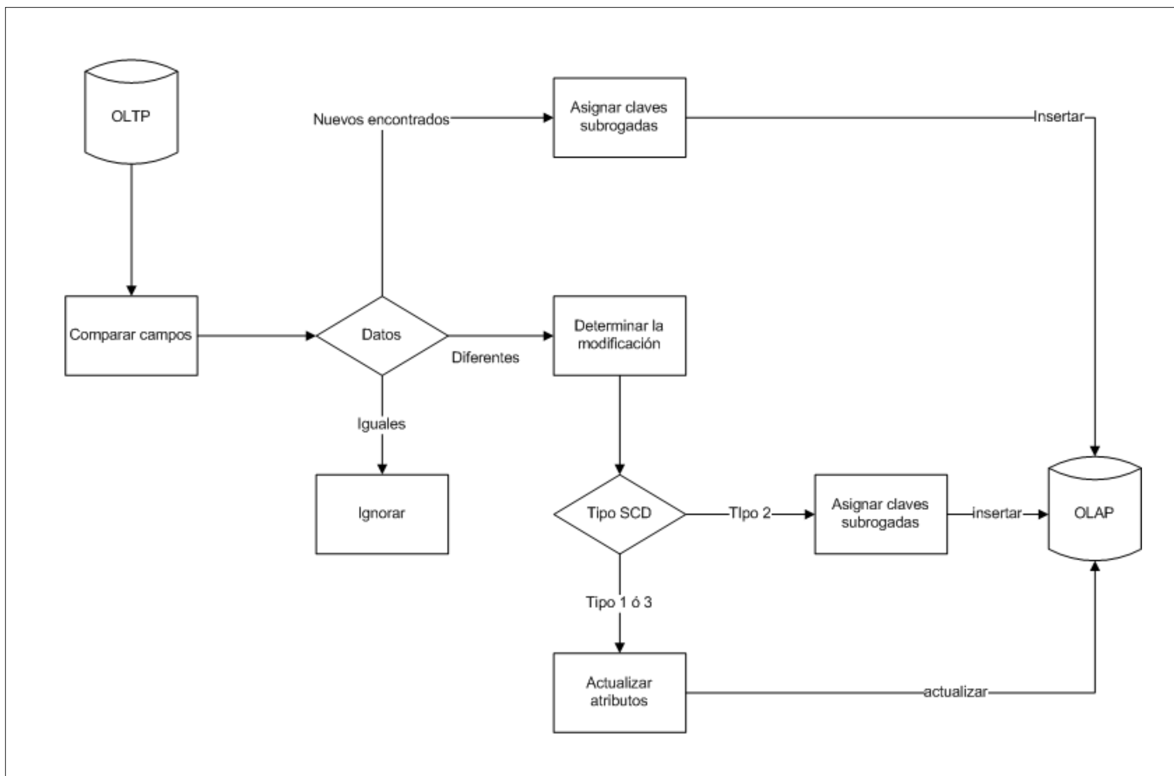


Figura 4.4: Algoritmo para el tratamiento de las SCD

4.3.3 Implementación en Kettle

Kettle (K Extraction, Transformation, Transportation & Load Environment) es una herramienta de código abierto que contiene una amplia gama de utilidades diseñada para ayudar en los procesos ETL. Está dividida en pasos a través de los cuáles se implementan las diferentes tareas. A continuación se muestran los más importantes:

- Input: Contiene las diferentes fuentes de datos con las que se realizará la extracción de la información, tales como: archivos CSV, archivo XML, archivo XLS, archivos genéricos o bien información directa de una tabla por medio de una secuencia SQL.
- Output: Contiene los diferentes destinos que pueden tener los datos una vez que han pasado por el proceso de transformación, tales como: archivos CSV, archivos de propiedades, archivos XML o bien una tabla en la base de datos OLAP.
- Transform: Contiene las tareas básicas de transformación de datos como filtrado de filas, mapeo de valores, agrupación, calculadora, normalización y desnormalización, pivoteo, validación de datos, entre otras.
- Scripting: Puede ser considerado parte del paso anterior debido a que contiene un editor que soporta javascript para que a través de este los datos sean manipulados y transformados por medio de las diferentes funciones del lenguaje de script.
- Data warehouse: Es este paso el que contiene las tareas para la implementación del llenado de las diferentes dimensiones y la tabla de hechos, Kettle se encargará de crear las claves subrogadas y administrarlas, junto con las primarias originales, para la gestión de las tablas.

Cada proceso creado para la manipulación de datos se hace de manera gráfica indicando la interacción de los elementos.

Implementación de Dimensión Escuelas:

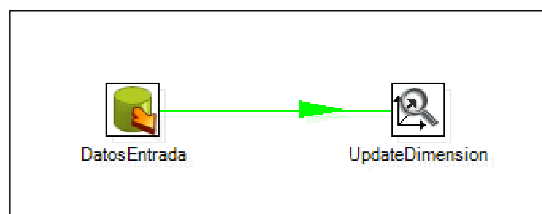


Figura 4.5: ETL de dimensión Escuelas

Esta tarea cuenta con 2 procesos principales, la extracción de datos de una tabla de la base de datos OLTP y la carga hacia una de las dimensiones del cubo OLAP.

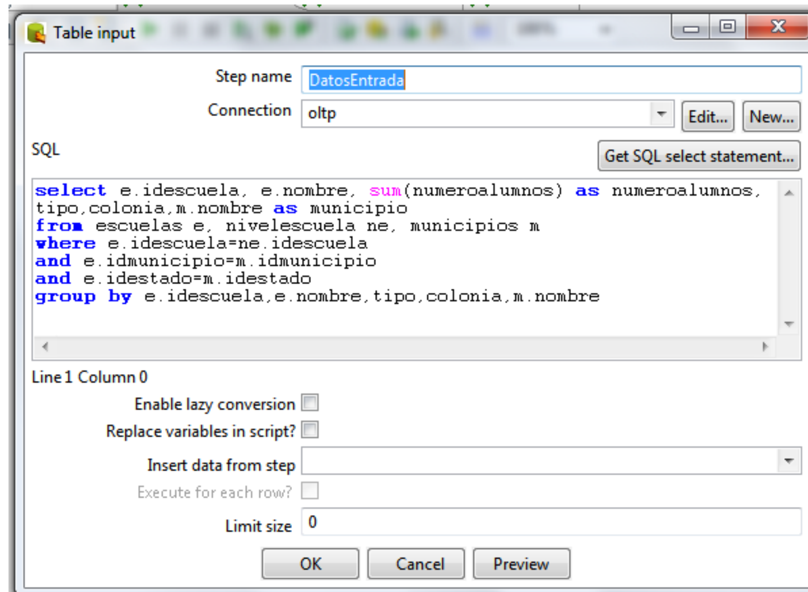


Figura 4.5 (*continuación*)

En el primer proceso se seleccionan, por medio de una sentencia SQL, los atributos del sistema operacional que serán utilizados para alimentar el sistema OLAP así como la conexión del origen de datos.

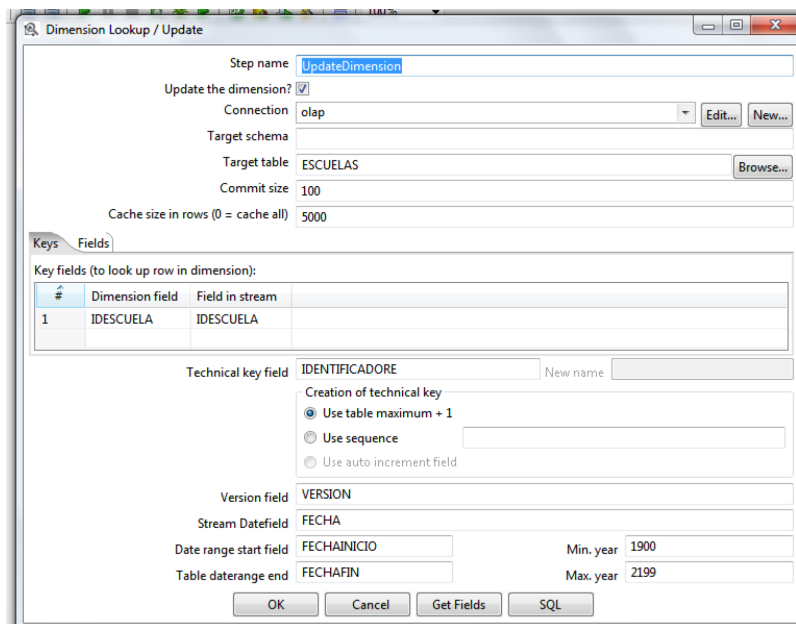


Figura 4.5 (*continuación*)

En el siguiente paso se configuran la conexión hacia la base de datos destino, la tabla a la que llegarán los datos extraídos, la columna que contiene el identificador original de cada fila, la columna que contendrá el identificador subrogado de los registros (*Technical Key Field*), la estrategia que se utilizará para su creación (*máximo + 1, secuencia o autoincremental*), así como las columnas que indicarán el periodo de validez del registro (*Date range start field-Table daterange end*) y la versión de este (*Version Field*) para que se haga uso del algoritmo SCD.

Mediante el check button *Update Dimension?* se le indica si se tratará de una operación de sólo lectura (únicamente obtiene la clave subrogada mediante el identificador original) o de lectura/escritura (obtiene la clave subrogada y en caso de que se detecte una variación en los datos originales se actualizará la dimensión).

La implementación de la dimensión Uniformes es bastante similar a la anterior, por lo que sólo resta implementar la dimensión Tiempo y la conjunción de todas con la tabla de hechos.

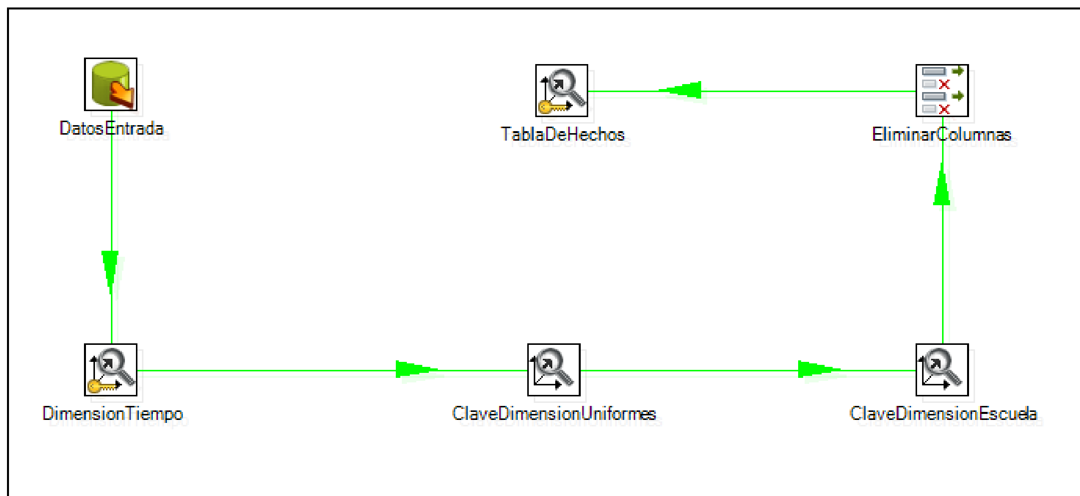


Figura 4.6: ETL de la tabla de Hechos y dimensión Tiempo

Es conveniente unir en una misma tarea la manipulación de la dimensión tiempo y la tabla de hechos debido a que cada hecho sucede en una fecha determinada, partiendo de ese dato se puede poblar de manera automática la dimensión de tiempo.

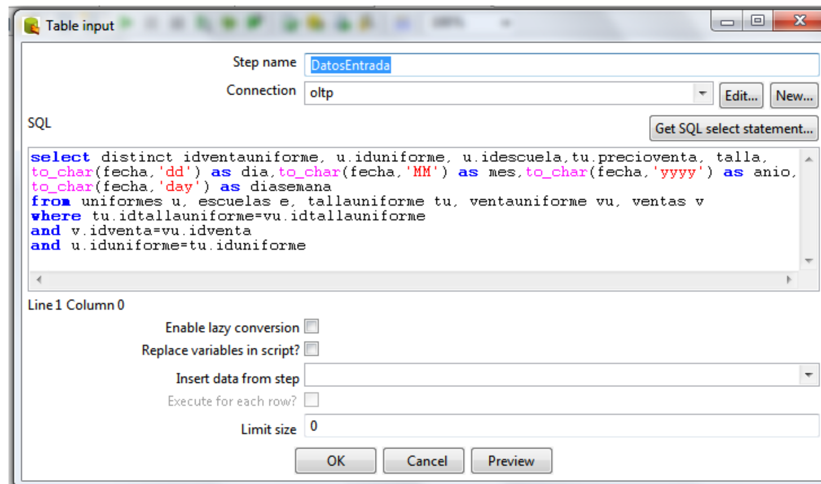


Figura 4.6 (*continuación*)

En el paso inicial se obtienen las columnas requeridas por la tabla de hechos (principalmente identificadores) así como la fecha en que ocurrió tal evento.

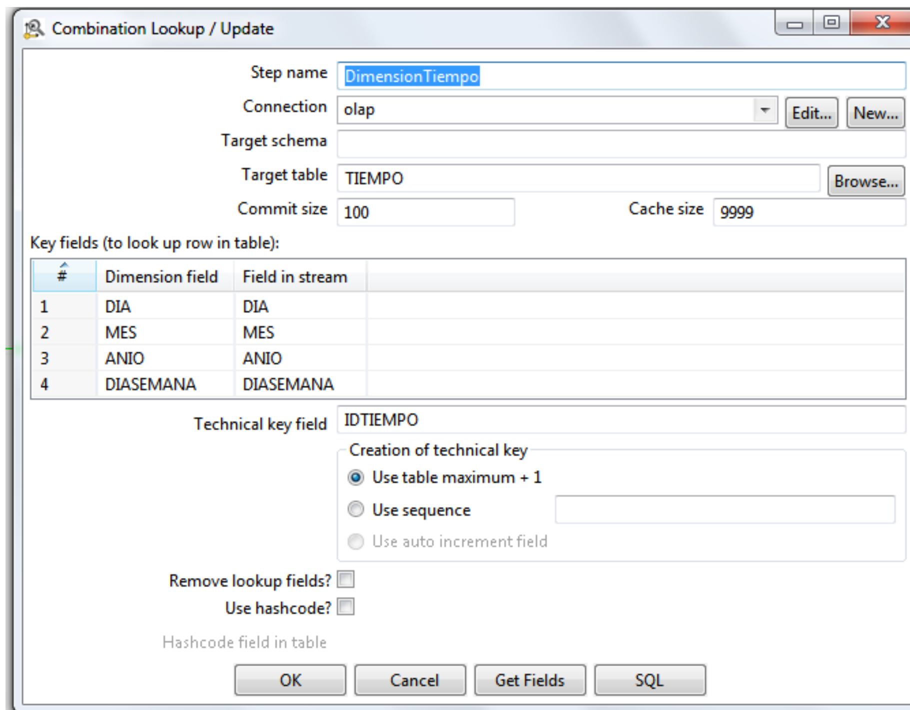


Figura 4.6 (*continuación*)

Se utiliza el paso *Combination Lookup/Update* para llenar la tabla del tiempo. El funcionamiento de este paso es el siguiente: Mediante los atributos seleccionados (dia, mes, anio, diasemana, en este caso) realiza una búsqueda en cada una de las filas de la tabla indicada. Si la combinación de valores de estos atributos (provenientes del paso anterior) existe, entonces se obtiene la clave subrogada que se creó para dicha combinación, de lo contrario se inserta una nueva fila con dicha combinación de valores y retorna la clave subrogada creada.

En los siguientes dos pasos se lleva a cabo la configuración de la obtención de las claves subrogadas de las dimensiones escuelas y uniformes, por lo que deberán ser marcadas como sólo lectura usando el identificador original obtenido en el primer paso.

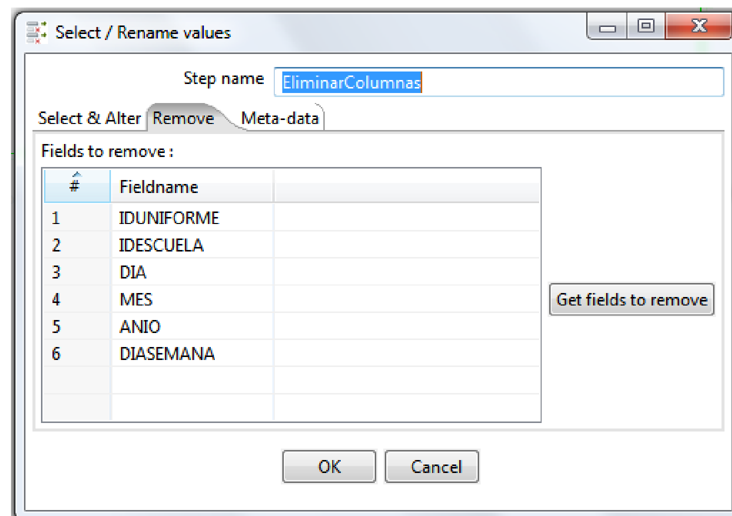


Figura 4.6 (*continuación*)

Debido a que el flujo de datos obtenidos desde el paso 1 hasta el paso actual se conserva, existen atributos que no son necesarios para poblar la tabla de hechos (tales como los identificadores originales útiles para la recuperación de las claves subrogadas y la combinación de valores de la fecha útiles para la dimensión tiempo) por lo que deben ser removidos del flujo y únicamente mantener los que son requeridos por la tabla central.

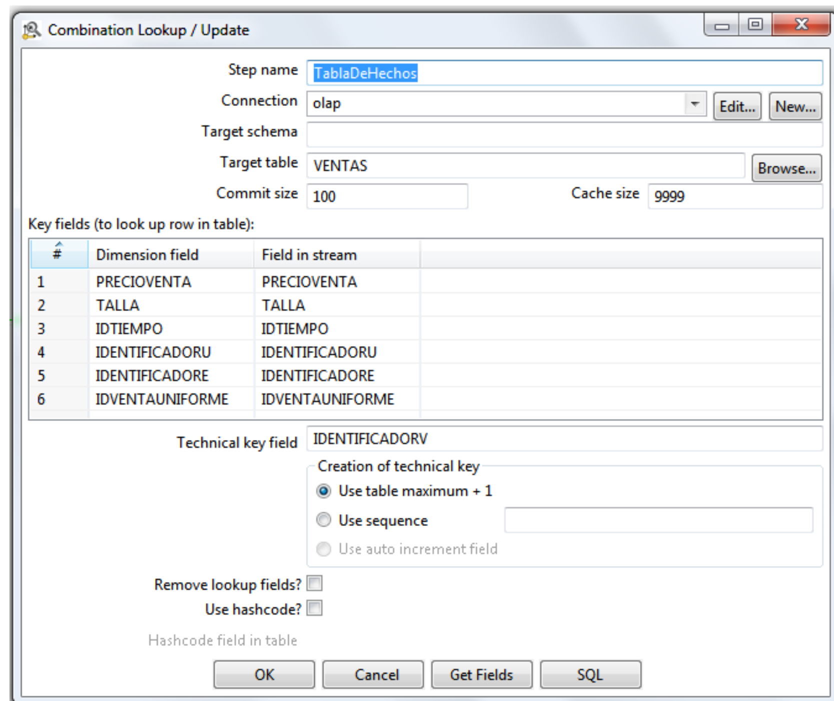


Figura 4.6 (*continuación*)

Finalmente, con la combinación de identificadores obtenidos y las medidas requeridas, se inserta un nuevo registro en la tabla de hechos con su propio identificador sustituto.

Todo este proceso debe realizarse de manera automática en intervalos de tiempo regulares. Kettle cuenta con un programa, llamado Kitchen, que realiza la ejecución automática de los trabajos programados.

Kitchen permite la ejecución de un trabajo por medio de dos opciones:

1. A través de un archivo: Los trabajos realizados en Kettle pueden ser guardados en archivos de extensión .kjb en el que se almacena toda la configuración del trabajo. Para ser ejecutado basta con llamar al programa y la ubicación del archivo por medio de su opción file en línea de comandos.

```
Kitchen.bat /file:"C:\ActualizarCubo.kjb"
```

Código 4.1: Programación automatic de trabajos

2. A través del repositorio: Si se opta por esta opción, se debe especificar el nombre del repositorio, el nombre del trabajo a ejecutar, el nombre de usuario y su contraseña.

```
Kitchen.bat /rep:"OLAP" /job:"TablaHechos" /user:usuario /pass:pass
```

Código 4.1 (*continuación*)

Es de vital importancia conocer lo que sucede con cada una de las ejecuciones automáticas que se llevan a cabo para así comprobar que el sistema OLAP está recibiendo los datos de manera correcta o de lo contrario, diseñar una estrategia para trabajar con los datos erróneos. El resultado de cada una de estas ejecuciones puede ser redirigido hacia un archivo de bitácoras incremental, en el que se indicará la hora de ejecución, el trabajo realizado y si fue finalizado con éxito o no. Para esto, después de especificar el tipo de ejecución (por archivo o repositorio), se indica el archivo al que se mandarán todos estos resultados.

```
Kitchen.bat /file:"C:\ActualizarCubo.kjb" > C:\LOG\trans.log
```

Código 4.1 (*continuación*)

Para la programación automática de estos procesos, las instrucciones deben ser guardadas en un archivo por lotes e indicar el momento en el que deben ser ejecutadas con el comando `at` en Windows o con `crontab` en Linux.

```
Administrador: C:\Windows\system32\cmd.exe
C:\Users\Hugo\Desktop\pentaho>Kitchen.bat /rep:"OLAP" /job:"TablaHechos" /user:a
dmin /pass:admin
INFO  17-03 16:46:03,717 - Kitchen - Start of run.
2011/03/17 16:46:06:183 CST [INFO] DefaultFileReplicator - Using "C:\Users\Hugo\
AppData\Local\Temp\vfs_cache" as temporary files store.
INFO  17-03 16:46:06,369 - RepositoriesMeta - Reading repositories XML file: C:\
Users\Hugo\.kettle\repositories.xml
INFO  17-03 16:46:07,484 - TablaHechos - Starting entry [VENTA]
INFO  17-03 16:46:07,507 - VENTA - Loading transformation from repository [VENTA
] in directory [/]
INFO  17-03 16:46:07,998 - VENTA - Dispatching started for transformation [VENTA
]
INFO  17-03 16:46:08,044 - VENTA - This transformation can be replayed with repl
ay date: 2011/03/17 16:46:08
INFO  17-03 16:46:08,255 - DatosEntrada.0 - Finished reading query, closing conn
ection.
INFO  17-03 16:46:08,268 - DatosEntrada.0 - Finished processing (I=3, O=0, R=0,
W=3, U=3, E=0)
INFO  17-03 16:46:08,317 - org.pentaho.di.trans.steps.combinationlookup.Combinat
ionLookup - Finished processing (I=2, O=0, R=3, W=3, U=3, E=0)
INFO  17-03 16:46:08,340 - ClaveDimensionUniformes.0 - Finished processing (I=1,
 O=0, R=3, W=3, U=3, E=0)
INFO  17-03 16:46:08,359 - ClaveDimensionEscuelas.0 - Finished processing (I=1,
O=0, R=3, W=3, U=3, E=0)
INFO  17-03 16:46:08,369 - EliminarColumnas.0 - Finished processing (I=0, O=0, R
=3, W=3, U=3, E=0)
INFO  17-03 16:46:08,392 - org.pentaho.di.trans.steps.combinationlookup.Combinat
ionLookup - Finished processing (I=3, O=0, R=3, W=3, U=3, E=0)
INFO  17-03 16:46:08,526 - TablaHechos - Starting entry [Success 1]
INFO  17-03 16:46:08,534 - TablaHechos - Finished jobentry [Success 1] (result=[
true])
INFO  17-03 16:46:08,536 - TablaHechos - Finished jobentry [VENTA] (result=[true
])
INFO  17-03 16:46:08,549 - Kitchen - Finished!
INFO  17-03 16:46:08,551 - Kitchen - Start=2011/03/17 16:46:06.347, Stop=2011/03
/17 16:46:08.549
INFO  17-03 16:46:08,554 - Kitchen - Processing ended after 2 seconds.
```

Figura 4.7: Ejecución de Trabajo mediante Kitchen.bat

Apéndice A

El lenguaje EL y las librerías JSTL

Si bien Struts ofrece sus propias librerías de acciones para la manipulación de datos en las páginas JSP, es muy común y en algunas ocasiones conveniente, utilizar estas librerías junto con las que ofrece la especificación estándar JSTL. La principal causa de la mezcla de estas librerías se debe a la complejidad, eficiencia o funcionalidad que algunas funciones JSTL ofrece sobre algunas pertenecientes a Struts. Esta comparativa es totalmente dependiente para cada desarrollo y necesidades del programador.

A.1 Variables implícitas EL

Como fue mencionado en el capítulo 2. Las acciones de estas librerías se encuentra enfocadas al uso del lenguaje de expresiones EL. Este lenguaje permite que el programador pueda acceder a los datos de petición y propiedades de objetos sin la necesidad de utilizar lenguaje Java. Las llamadas variables implícitas son las encargadas de cargar automáticamente los datos de petición del usuario, de su espacio en sesión o de la configuración de la aplicación. En la siguiente tabla se muestran las principales variables implícitas JSTL:

Nombre	Descripción
param	Es una colección que contiene todos los valores de los parámetros de petición.
header	Es una colección que contiene los valores de las cabeceras de la petición.
cookie	Es una colección que contiene todas las cookies que el navegador haya enviado en la petición.
initParam	Es una colección que contiene todas los parámetros de configuración almacenados en el archivo `web.xml`
pageScope	Es una colección que contiene todas las variables que tengan ámbito de página.
requestScope	Es una colección que contiene todas las variables que tengan ámbito de petición.

Tabla A.1: Variables implícitas de JSTL

Nombre	Descripción
sessionScope	Es una colección que contiene todas las variables que tengan ámbito de sesión.
applicationScope	Es una colección que contiene todas las variables que tengan ámbito de aplicación.

Tabla A.1 (*continuación*)

A.2 Ámbito de datos

En algunas variables implícitas se mencionan los distintos ámbitos que puede tener una variable en la aplicación. El ámbito es el alcance que una variable tiene definido a lo largo de la aplicación. Existen 4 tipos de ámbito:

1. Ámbito de página: Es el ámbito por defecto. La variables con este tipo de alcance sólo se encuentran disponibles en la página actual (en la que está siendo procesada) una vez que hay redirección a otra o se sale de esta la variable deja de estar disponible.

2. Ámbito de petición: La disponibilidad de una variable con este ámbito se da a lo largo de la resolución de una petición HTTP.

3. Ámbito de sesión: Las variables con este ámbito se encuentran disponibles durante el tiempo que el usuario utilice la aplicación. La variable se almacena en una cookie en memoria que estará disponible mientras el navegador se encuentre abierto o el tiempo de disponibilidad de este no sea rebasado.

4. Ámbito de aplicación: Este es el alcance máximo que puede tener una variable y su valor puede ser accedido por todos los usuario (independientemente del navegador).

A.3 Funciones de EL

El lenguaje EL dispone de distintas funciones que permiten un mejor manejo de los datos como se muestra a continuación:

Función	Retorno	Descripción
fn:contains(cadena,subcadena)	boolean	Verifica si la subcadena se encuentra contenida en la cadena.
fn:containdIgnoreCase(cadena,subcadena)	boolean	Verifica si la subcadena se encuentra contenida en la cadena sin importar el uso de mayúsculas y minúsculas.
fn:endsWith(cadena,sufijo)	boolean	Verifica que la cadena termine con el sufijo indicado.
fn:startsWith(cadena,prefijo)	boolean	Verifica que la cadena comience con el prefijo indicado.
fn:indexOf(cadena,subcadena)	int	Devuelve la posición en la que la cadena contiene a la subcadena o -1 si no se encuentra.
fn:length(cadena)	int	Devuelve la longitud de la cadena.
fn:Split(cadena,separador)	String[]	Divide la cadena original por cada ocurrencia que se encuentre del separador.
fn:toLowerCase(cadena)	String	Devuelve la cadena transformando cada uno de sus caracteres en mayúsculas a minúsculas.
fn:toUpperCase(cadena)	String	Devuelve la cadena transformando cada uno de sus caracteres en minúsculas a mayúsculas.
fn:trim(cadena)	String	Devuelve la cadena limpia de espacios iniciales y finales.

Tabla A.2: Funciones EL

A.4 La librería Core

Esta librería se encarga de las condiciones, bucles, asignación de variables y redirecciones. Sus acciones principales son:

<c:if>

Se trata de una etiqueta condicional simple en la que se evalúa el contenido de la variable comparándolo con la condición indicada y así realizar cierta acción. Sus atributos principales son:

Atributo	Tipo	Descripción
test	Boolean	Contiene la expresión a evaluarse. Es el único atributo obligatorio.
var	String	El nombre de la variable donde se guardará, de ser necesario, el resultado de la evaluación.
scope	String	El ámbito que se dará a la variable `var`. Por defecto se le asigna `page`.

Tabla A.3: Atributos de la etiqueta `<c:if>`

<c:choose>, <c:when>, <c:otherwise>

Estas etiquetas conforman entre sí una condición más compleja para ser evaluada. La etiqueta `<c:when>` presenta el mismo comportamiento que `<c:if>`, pero siempre es utilizada dentro de un bloque definido por `<c:choose>`, el cuál controla el procesamiento de los diferentes `<c:when>` anidados, y así comprobar los diferentes valores que puede tomar la condición. Análogamente a la programación estructurada, su uso es similar al `switch`.

<c:choose> → switch

<c:when> → case

<c:otherwise> → default

El único atributo requerido por estas etiquetas es `test` (de funcionamiento idéntico que en `<c:if>`) y es soportado por la etiqueta `<c:when>`

<c:forEach>

Permite la iteración sobre las colecciones para obtener los valores contenidos en estas. Sus atributos son los siguientes:

Atributo	Tipo	Descripción
begin	int	Indica el índice en el que iniciarán las iteraciones.
end	int	Indica el último índice de la colección
items	array	La colección de objetos sobre la que se está iterando.
step	int	Indica el incremento de cada iteración. Por defecto su valor es 1.
var	String	El nombre de la variable que se usará como referencia para el elemento actual de la colección.

Tabla A.4: Atributos de la etiqueta `<c:forEach>`

<c:set>

Útil para la creación de variables requeridas para el procesamiento de la página JSP. Sus atributos son los siguientes:

Atributo	Tipo	Descripción
value	Object	El valor que será asignado a la variable.
var	Object	Nombre de la variable que será creada.
scope	String	Ámbito de dato que será asignado a la variable.

Tabla A.5: Atributos de la etiqueta `<c:set>`

A.5 La librería Format

Esta librería contiene funcionalidades útiles para dar formato a la información usando los patrones dados. Sus etiquetas principales son:

<fmt:formatDate>

Se utiliza para dar un formato específico a las fechas independientemente del formato que tengan en el servidor. Sus atributos principales son:

Atributo	Tipo	Descripción
value	Date	La fecha a la que se dará formato.
pattern	String	El patrón en que debe mostrarse la fecha dada.
type	String	Indica si se desea presentar sólo la fecha, la hora o ambos.
timeZone	String	Especifica la zona horaria
var	String	Indica la variable en que se guarda la fecha una vez que ha sido formateada.
scope	String	Especifica el ámbito de la variable anterior.

Tabla A.6: Atributos de la etiqueta `<ftm:formatDate>`

<fmt:formatNumber>

Su uso es muy parecido a la etiqueta anterior pero aplicada al formato de los números. Sus atributos principales son:

Atributo	Tipo	Descripción
value	String / Número	Especifica el número a formatear.
pattern	String	Indica el patrón que debe seguir el número dado.
type	String	Indica de qué tipo de número de trata: cantidad, moneda, o porcentaje.
currencyCode	String	Indica el código ISO-4217 de la moneda (define todas las monedas del mundo por medio de un código. MXN es para el peso mexicano).
currencySymbol	String	Indica el símbolo de la moneda

Tabla A.7: Atributos de la etiqueta `<ftm:formatNumber>`

Atributo	Tipo	Descripción
maxIntegerDigits	int	Especifica el número máximo de cifras en la parte entera del número.
maxFractionDigits	int	Especifica el número máximo de cifras en la parte decimal del número.
minFractionDigits	int	Especifica el número mínimo de cifras en la parte decimal del número.
var	String	Indica la variable en que se guarda la fecha una vez que ha sido formateada.
scope	String	Especifica el ámbito de la variable anterior.

Tabla A.7 (*continuación*)

A.6 La librería SQL

Esta librería contiene las diferentes tareas requeridas para el acceso a bases de datos realizado directamente en las páginas JSP. Esta acción, como se ha mencionado y demostrado, no es recomendable debido a la mezcla de código de presentación, lógica de negocio y acceso a datos en una JSP. Dando como resultado la nula reutilización de elementos programados, código altamente acoplado, nula escalabilidad y un mantenimiento deficiente. Sus principales etiquetas (para que queden en conocimiento) son:

<sql:query>
Utilizada para definir las consultas encargadas de leer información de la base datos.

<sql:transaction>
Dentro de esta etiqueta se marcan las diferentes sentencias que forman parte de una transacción.

<sql:update>

Utilizada para realizar tareas de actualización tales como `INSERT`, `DELETE`, y `UPDATE`.

Conclusiones

Hasta el año 2009, según los datos de la Secretaría de Economía, se contaba con un total de 5’144,056 de empresas en el país, de las cuáles, el 98% entraban en la clasificación de PyME. Con esta cifra no es de sorprender que el 78.5% del personal ocupado se encuentre laborando en alguna de estas pequeñas empresas, convirtiéndolas así en la columna de vertebral del sistema económico del país.

El problema real con este tipo de empresas tan importantes para el desarrollo del país, es que muchas de estas nacen de la búsqueda de ciertas oportunidades económicas a través de la oferta de cierto producto o servicio, por lo que comienzan laborando de manera improvisada y sin una visualización concreta hacia el futuro. Si a lo anterior se le suma que muchas veces el objetivo económico no es alcanzado, se llega a la cifra de que el 50% de las PyMEs que comienzan su ciclo de vida, lo terminan el primer año de su existencia, el 75% de las restantes tendrán el mismo desenlace para el siguiente año. Las “sobrevivientes” tendrán que mejorar la administración de su empresa para no correr con la misma suerte en años posteriores.

De lo que estas empresas adolecen es de una intrusión al mundo tecnológico, con esto, no hablo de robots trabajando en lugar de personas, sino de una automatización de todos esos procesos que puedan sustituir al papel y al lápiz para una gestión óptima. A veces esto no se hace por la falta de conocimiento, falta de recursos que cubran los gastos o bien la resistencia al cambio.

Gracias al enorme paso que decidieron dar estos pequeños empresarios y a quitarse el estigma de que un cambio nunca es bueno, las oportunidades de crecimiento de esta PyME serán expandidas y notarán cómo el crecimiento de la base de los clientes y las ganancias obtenidas será inversamente proporcional al tiempo invertido en sus procesos de negocio.

En cuanto a la construcción de la aplicación se refiere, se logró conjuntar diferentes tecnologías que brindan por sí solas eficiencia en el desarrollo y ejecución de ciertas áreas de la aplicación. Estando en la era del web 2.0 en el que la apariencia visual y el tiempo de respuesta juegan una parte fundamental en el éxito de una aplicación, con CSS, jQuery y ajax se lograron diferentes efectos que permiten una navegación más intuitiva, cómoda y veloz en las pantallas del sistema.

Por otro lado, los frameworks que ofrece Java para la programación del servidor valieron para modularizar la aplicación haciendo uso del patrón MVC que permite programar un código más limpio y cuyo mantenimiento sea rápido y fácil. Struts como framework de presentación de datos hoy por hoy sigue siendo el líder en el mercado del MVC (a pesar de tener poco más de 10 años de existencia) debido a que es sencillo, robusto y muy escalable. Hibernate , por su lado, resultó ser muy eficiente para el acceso a datos debido a la casi nula programación que se requiere para recuperar información de la base de datos y siempre conservando la portabilidad que ofrece JDBC pues finalmente se encuentra basado en este último. Y finalmente Spring, un framework orientado más a los aspectos de la programación (que quizá el usuario no entendería) que a los del negocio, pero demasiado útil para lograr que la consistencia de la información siempre se mantenga (por medio de las transacciones) y de paso, optimizar la ejecución de los objetos involucrados en esta así como mejorar la interacción entre los diferentes módulos y frameworks.

Si bien el uso de las tecnologías expuestas en este trabajo no son del todo imprescindibles, se logró demostrar el porqué son tan necesarias para un desarrollo óptimo y eficaz, en el que el tiempo siempre es un aspecto crítico. De igual manera, si en un futuro se desea modificar la implementación de algún módulo, se podrá realizar sin la necesidad de hacer cambios importantes que comprometan el funcionamiento de los demás.

Para finalizar, quisiera dar por terminada la percepción de un mito que puede llegar a modificar, como estudiantes, nuestra forma de pensar acerca de que las ciencias exactas no se encuentran concebidas para la interacción con temas relativos a la sociedad. En este trabajo se está haciendo uso de estas ciencias “anti-sociales” para mejorar la calidad de una sociedad que se encuentra en búsqueda de crecimiento y mejores oportunidades.

Bibliografía y mesografía

Bibliografía

- PÉREZ, César
  Oracle 10g: Administración y análisis de Bases de Datos
  2ª Edición
  México
  Alfaomega, 2008, 712 p

- FOWLER, MARTIN
  UML gota a gota
  1ª Edición
  México
  Addison Wesley, 1999, 224 p

- EGUÍLUZ, Javier
  CSS Avanzado
  España
  Libros web, 2009, 151 p

- BIBEAULT, Bear, KATZ, Yehuda
  jQuery In Action
  1ª Edición
  USA
  Manning, 2008, 377 p

- FLANAGAN, David
  jQuery Pocket Reference
  1ª Edición
  USA
  O'Reilly, 2010, 158 p

- DEITEL, Hervey, DEITEL, Paul
  Cómo programar en Java
  5ª Edición
  México
  Pearson, 2004, 1268 p

- MENON, R. M.
  Expert Oracle JDBC Programming
  1ª Edición
  USA
  Apress, 2005, 722 p

- HUNTER, Jason
  Java Servlet Programming
  1ª Edición
  USA
  O'Reilly, 1998, 408 p

- URBANEJA, Javier
  JSP, Guía Práctica
  1ª Edición
  España
  Anaya Multimedia, 2008, 319 p

- BAYERN, Shawn
  JSTL In Action
  1ª Edición
  USA
  Manning, 2003, 480 p

- MARTÍN, Antonio
  Struts
  1ª Edición
  México
  Alfaomega, 2008, 300 p

- CAVANESS, Chuck
  Programming Jakarta Struts
  2ª Edición
  USA
  O'Reilly, 2004, 470 p

- DORAY, Arnold;
  Beginning Apache Struts: From Novice to Professional
  1ª Edición
  USA
  Apress, 2006, 536 p

- HOLMES, James
  Struts: The Complete Reference
  2ª Edición
  USA
  McGraw-Hill, 2007, 800 p

- WALLS, Craig, BREIDENBACH, Ryan
  Spring In Action
  1ª Edición
  USA
  Manning, 2005, 472 p

- VAN, Thomas, SNYDER, Bruce, DUPUIS, Christian, LI, Sing, HORTON, Anne, BALANI, Naveen
  Beginning Spring Framework 2
  1ª Edición
  USA
  Wrox, 2008, 507 p

- SMEETS, Bram, LADD, Seth
  Building Spring 2 Enterprise Applications
  1ª Edición
  USA
  Apress, 2007, 343 p

- BAUER , Christian, KING, Gavin
  Java Persistence with Hibernate
  1ª Edición
  USA
  Manning, 2007, 876 p

- MINTER, Dave, LINWOOD, Jeff
  Beginning Hibernate: From Novice to Professional
  1ª Edición
  USA
  Apress, 2006, 359 p

- BAUER , Christian, KING, Gavin, ANDERSEN, Rydahl, BERNARD, Emmanuel, EBERSOLE, Steve
  Hibernate - Relational Persistence for Idiomatic Java
  2009, 342 p

- BERNARD, Emmanuel
  Hibernate Annotations - Reference Guide
  2010, 49 p

Fuentes

- MORENO, Luciano
  El Color en la Web
  http://www.terra.es/personal6/morenocerro2/diseno/color/color_1.html
  (Consultado: 5 - julio - 2010)

- **Nociones básicas de diseño: Teoría del color**
  http://www.weblogicnet.com/descargas/teoria-del-color.pdf
  (Consultado: 5 - julio - 2010)

- **Cultura del color**
  http://estocolmo.se/cultura/color_oktub23.htm
  (Consultado: 5 - julio - 2010)

- **Hypertext Transfer Protocol**
  http://www.w3.org/Protocols/rfc2616/rfc2616.html
  (Consultado: 21 - noviembre - 2010)

- **Servlet API Documentation**
  http://tomcat.apache.org/tomcat-5.5-doc/servletapi/index.html
  (Consultado: 26 - noviembre - 2010)

- **Struts 1.3.10 API**
  http://struts.apache.org/1.x/apidocs/index.html
  (Consultado: 28 - noviembre - 2010)

- **What Is in the JDBCTM 2.0 API**
  http://download.oracle.com/javase/1.3/docs/api/java/sql/package-summary.html#2.0%20API
  (Consultado: 5 - diciembre - 2010)

- **Hibernate JavaDoc (3.5.6-Final)**
  http://docs.jboss.org/hibernate/stable/core/api/overview-summary.html
  (Consultado: 7 - diciembre - 2010)

- **Spring Framework**
  http://static.springsource.org/spring/docs/1.1.5/api/index.html
  (Consultado: 15 - diciembre - 2010)

- **Plugins/Validation**
  http://docs.jquery.com/Plugins/Validation
  (Consultado: 1 - febrero - 2011)

- **UI/API/1.8/Datepicker**
  http://docs.jquery.com/UI/Datepicker
  (Consultado: 1 - febrero - 2011)

- **jQuery Documentation**
  http://docs.jquery.com/Main_Page
  (Consultado: 11 - febrero - 2011)

- **OLTP vs. OLAP**
  http://datawarehouse4u.info/OLTP-vs-OLAP.html
  (Consultado: 20 - febrero - 2011)

- **Dimensiones lentamente cambiantes**
  http://www.dataprix.com/ca/node/2369
  (Consultado: 20 - febrero - 2011)

- **Contacto PyME**
  http://www.economia.gob.mx/swb/es/economia/p_Contacto_PyME
  Consultado: 1 - marzo - 2011)

- **Cierran en el primer año la mitad de las PyMEs**
  http://www.oem.com.mx/oem/notas/n1594296.htm
  Consultado: 1 - marzo - 2011)

Glosario

1. **API (Application Programming Interface):**

   Conjunto de clases que forman parte de una librería o framework.

2. **CSS (Cascade Style Sheet):**

   Lenguaje que define la presentación de un documento HTML.

3. **DI (Dependency Injection):**

   Patrón de diseño propio de la programación orientada a objetos en el que las dependencias de los objetos son suministradas en vez de instanciadas por estos.

4. **EL (Expression Language):**

   Lenguaje de expresiones utilizado por JSTL para acceder a las propiedades de los objetos.

5. **GUI (Graphical User Interface):**

   Componentes gráficos que ayudan a la interacción de un sistema con el usuario.

6. **HTTP (Hypertext Transfer Protocol):**

   Protocolo de petición/respuesta que comunica al cliente con el servidor.

7. **JDBC (Java DataBase Conectivity):**

   API estándar de Java que permite la conectividad con una base de datos.

8. **JSP (Java Server Pages):**

   Tecnología Java que permite la creación de contenido dinámico para la creación de documentos HTML.

9. **JSTL (JSP Standard Tag Library):**

   Librería estándar de JSP para el manejo de contenido dinámico a través de etiquetas de acciones en sustitución de código Java embebido.

10. **MIME (Multipurpose Internet Mail Extensions):**
    Estándar utilizado por el navegador para comprender y manejar los contenidos incluidos en él (image/jpeg, text/html, video/mpeg…)

11. **MVC (Model, View, Controller):**
    Patrón de diseño que divide la lógica de negocio, el acceso a datos (Modelo), la presentación de información (Vista) y el manejo de la petición (Controlador) con la finalidad de desacoplar los componentes.

12. **OLAP (On-Line Analytical Processing):**
    Base de datos aplicada al procesamiento analítico para la toma de decisiones.

13. **OLTP (On-Line Transaction Processing):**
    Base de datos aplicada para las operaciones transaccionales.

14. **ORM (Object Relational Management):**
    Paradigma de conversión de datos relacionales a objetos (y viceversa).

15. **POJO (Plain Old Text Object):**
    Clases que definen atributos con sus métodos getter y setter para el transporte de datos.

16. **RDBMS (Relational DataBase Management System):**
    Sistema gestor de base de datos que proporciona el ambiente para el manejo de datos relacionales.

17. **SCD (Slow Changing Dimension):**
    Dimensiones cuyos datos tienden a modificarse ocasionalmente en el tiempo.

18. **W3C (World Wide Web Consortium):**
    Comunidad internacional encargada de definir los estándares del mundo web.

Índice de Figuras

Figura 1.1: Diagrama Entidad Relación…… 11
Figura 1.2: Diagrama Jerárquico Funcional…… 12
Figura 1.3: Diagrama de Flujo de Datos…… 13
Figura 1.4: Almacenamiento físico en la Base de Datos…… 18

Figura 2.1: Funcionamiento básico del protocolo HTTP…… 39
Figura 2.2: Juego hecho con la tecnología applet de Java y embebido en una página web…… 41
Figura 2.3: Funcionamiento del servlet…… 42
Figura 2.4: API general de los Servlets…… 43
Figura 2.5: Petición de cliente…… 47
Figura 2.6: Respuesta del servidor…… 47
Figura 2.7: Funcionamiento básico del patrón MVC…… 51
Figura 2.8: API general de Struts…… 53
Figura 2.9: Funcionamiento de Struts…… 54
Figura 2.10: API general de JDBC…… 62
Figura 2.11: Componentes generales de Hibernate…… 69
Figura 2.12: Mapeo de una relación @OneToMany bidireccional…… 85
Figura 2.13: Arquitectura general de Spring…… 87
Figura 2.14: Intregración de Struts, Hibernate y Spring…… 102

Figura 3.1: Estructura de las capas externas de la interfaz de usuario…… 106
Figura 3.2: Estructura de las capas internas de la interfaz de usuario…… 107
Figura 3.3: Configuración de los diferentes módulos en web.xml …… 116
Figura 3.4: Funcionamiento de DispatchAction…… 118
Figura 3.5: Modelo de Dominio de la aplicación…… 120
Figura 3.6: Arquitectura de la aplicación…… 121

Figura 4.1: Sistemas OLTP y OLAP…… 125
Figura 4.2 Visualización multidimensional del hecho venta…… 126
Figura 4.3: Diseño de cubo OLAP orientado a ventas…… 128
Figura 4.4: Algoritmo para el tratamiento de las SCD…… 131
Figura 4.5: ETL de dimensión Escuelas…… 132
Figura 4.6: ETL de la tabla de Hechos y dimensión Tiempo…… 134
Figura 4.7: Ejecución de Trabajo mediante Kitchen.bat……138

Índice de Tablas

Tabla 1.1: Tamaño total de la Base de Datos…… 22
Tabla 1.2: Funciones principales de jQuery para la manipulación de elementos en una página…… 32
Tabla 1.3: Funciones principales de jQuery para dar efectos visuales a los elementos de una página…… 33
Tabla 1.4: Funciones principales de jQuery para el manejo de eventos…… 34
Tabla 1.5: Funciones principales de jQuery para la navegación de elementos de una página…… 34
Tabla 1.5: Parámetros de ajax()…… 35

Tabla 2.1: Métodos de la interface Servlet…… 44
Tabla 2.2: Métodos de la clase abstracta GenericServlet (implementa interface Servlet)…… 44
Tabla 2.3: Métodos de la clase abstracta HttpServlet (hereda a la clase abstracta GenericServlet)…… 45
Tabla 2.4: Algunos métodos de la interface HttpServletRequest…… 45
Tabla 2.5: Algunos métodos de la interface HttpServletResponse…… 45
Tabla 2.6: Funciones de JTSL. Ver apéndice A para mayor información…… 49
Tabla 2.7: Método principal de la clase Action…… 54
Tabla 2.8: Métodos de la clase ActionMapping…… 55
Tabla 2.9: Métodos principales de la clase DriverManager…… 63
Tabla 2.10: Métodos principales de la Interface Connection…… 64
Tabla 2.11: Métodos principales de la Interface Statement…… 64
Tabla 2.12: Métodos principales de la Interface PreparedStatement…… 65
Tabla 2.13: Métodos principales de la Interface ResultSet…… 65
Tabla 2.14: Métodos principales de Interface Configuration…… 70
Tabla 2.15: Métodos principales de Interface SessionFactory…… 70
Tabla 2.16: Métodos principales de Interface Session…… 71
Tabla 2.17: Métodos principales de Interface Query…… 72
Tabla 2.18: Comparativa entre los posibles niveles de aislamiento en transacciones…… 100

Tabla 3.1: Elementos de estilosAdmon.css y estilosCliente.css…… 104
Tabla 3.2: Contenido de funcionesP.js y funcionesC.js…… 107
Tabla 3.3: Contenido de funcionesAjax.js…… 108
Tabla 3.4: Contenido de login.js…… 110
Tabla 3.5: Contenido de autocompletar.js…… 110
Tabla 3.6: Directorios estándar de aplicaciones web…… 115
Tabla 3.7: Software involucrado en la construcción…… 122

Tabla 4.1: Comparativa entre esquema estrella y copo de nieve…… 127

Tabla A.1: Variables implícitas de JSTL…… 140

Tabla A.2: Funciones EL...... 142
Tabla A.3: Atributos de la etiqueta <c:if>...... 143
Tabla A.4: Atributos de la etiqueta <c:forEach>...... 144
Tabla A.5: Atributos de la etiqueta <c:set>...... 144
Tabla A.6: Atributos de la etiqueta <ftm:formatDate>...... 145
Tabla A.7: Atributos de la etiqueta <ftm:formatNumber>...... 145

Índice de Códigos

Código 1.1: Imagen de fondo en página HTML sin CSS........ 27
Código 1.2: Imagen de fondo en página HTML con CSS y etiquetas........ 28
Código 1.3: Tipos de letra en página HTML sin CSS........ 29
Código 1.4: Tipos de letra en página HTML con CSS y clases........ 29
Código 1.5: Capas en página HTML con CSS e ids........ 31
Código 1.6: Funcionamiento de load()........ 35
Código 1.7: Funcionamiento de ajax()........ 36
Código 1.8: Funcionamiento de post()........ 37

Código 2.1: Servlet que procesa una petición del cliente........ 46
Código 2.2: JSP que procesa la petición, usando scriptlets........ 48
Código 2.3: JSP que procesa la petición, usando la librería JSTL y lenguaje EL........ 50
Código 2.4: Estructura básica del archivo de configuración de Struts........ 55
Código 2.5: Registro de objetos ActionForm en struts-config.xml........ 56
Código 2.6: Registro de objetos Action, ActionMapping y ActionForward en struts-config.xml........ 56
Código 2.7: Registro de struts-config.xml en el descriptor de despliegue web.xml........ 57
Código 2.8: Conexión y consulta a una base de datos por medio de JDBC........ 66
Código 2.9: Configuración de hibernate mediante el archivo hibernate.cfg.xml........ 73
Código 2.10: Clase HibernateUtil encargada de crear el objeto SessionFactory........ 74
Código 2.11: Configuración del mapeo de la clase Proveedores en un archivo xml externo........ 76
Código 2.12: Configuración del mapeo de la clase Proveedores usando anotaciones........ 77
Código 2.13: Persistencia de los objetos Proveedores desde la base de datos con Hibernate........ 78
Código 2.14: Persistencia de los objetos Proveedores hacia la base de datos con Hibernate........ 79
Código 2.15: Mapeo entre Proveedores y Sucursales........ 85
Código 2.16: Suma de 2 números........ 89
Código 2.17: Suma de 2 números usando Interfaces........ 90
Código 2.18: Suma de 2 números en Spring........ 91
Código 2.19: Integración de Struts con Spring........ 94
Código 2.20: Integración de Hibernate con Spring........ 95
Código 2.21: Persistencia de los objetos Proveedores desde la base de datos con Hibernate y Spring........ 97
Código 2.22: Persistencia de los objetos Proveedores hacia la base de datos con Hibernate y Spring........ 97
Código 2.23: Configuración de transacciones........ 101

Código 3.1: Uso de QuickPager.js........ 112
Código 3.2: Validación de datos con Validate.js........ 113
Código 3.3: Uso de DatePicker.js........ 114
Código 3.4: Configuración de los diferentes módulos en web.xml........ 116

Código 3.5: Ejemplo de subclase DispatchAction…………………………………………………………………………………………… 118
Código 3.6: Configuración de la subclase DispatchAction en struts-config.xml……………………………………………… 119

Código 4.1: Programación automatica de trabajos…………………………………………………………………………………………… 138