CAPÍTULO 4

TECNOLOGÍAS RELACIONADAS

En este capítulo se tratarán varias tecnologías de vanguardia que se usan actualmente para la creación y manipulación de archivos XML, así como del marco de trabajo de formato abierto para implementación de servicios web basado en la arquitectura Modelo-Vista-Controlador más utilizado por la comunidad Java, Struts.

4.1 TECNOLOGÍAS PARA LA MANIPULACIÓN Y DESARROLLO DE XML

El desarrollo y manipulación de documentos en XML no es una tarea trivial, se necesitan aplicaciones que realicen las funciones de análisis, creación, manipulación, entrada y salida entre otros. Está sección explicará las tecnologías más importantes que se han desarrollado para facilitar la creación de aplicaciones XML.

4.1.1 SAX (SIMPLE API FOR XML)

SAX es un API nativo en java para procesar documentos XML. Está basado en eventos y es muy útil para procesar documentos y flujos de datos muy grandes. Dichos eventos son lanzados por el analizador según encuentra diferentes *tokens* del lenguaje de marcas, mientras se analiza como un flujo continuo de retrollamadas e invocaciones a métodos. Los eventos están anidados de la misma forma que los elementos en el documento, por lo tanto, no se crea ningún modelo de documento intermedio. Como el uso de memoria es bajo, el modelo de programación puede ser complejo especialmente si la estructura no corresponde con la estructura de datos de la aplicación. Como genera un flujo temporal de eventos, el API SAX no puede utilizarse cuando un modelo de documento tiene que ser editado o procesado varias veces. SAX está organizado alrededor de interfaces. Está basado sólo en dos interfaces, **XMLReader** que representa al *parser* y **ContentHandler** quien recibe los datos del *parser* y a través del cual se comunica con la aplicación del cliente [Harold, 2002].

SAX procesa la información por conforme ésta es presentada, es decir, evento por evento, manipulando cada elemento a un determinado tiempo, sin incurrir en uso excesivo de memoria. Sin embargo su mayor desventaja es que no se puede manipular la información una vez procesada [Rubio, 2002].

4.1.2 DOM (Document Object Model)

Es una interfaz neutral de plataforma y lenguaje que permite a los programas y *scripts* acceder y actualizar dinámicamente el contenido, estructura y estilo de los documentos. El documento puede ser procesado más a fondo y los resultados de éste proceso se pueden incorporar nuevamente dentro de la página actual [DOM, 2003].

DOM está organizado en niveles en lugar de versiones. El nivel uno detalla la funcionalidad y navegación del contenido de un documento. El nivel 2 provee módulos y opciones relacionados con modelos de contenido específicos, tal como XML, HTML, CSS (Cascade Style Sheets) entre otros. El nivel 3 resuelve el problema de la entrada y salida (I/O) de los datos (este nivel continúa en desarrollo). A continuación se muestra un ejemplo en JDOM para representar el elemento calle.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance( );
DOMImplementation impl = builder.getDOMImplementation( );
Document doc = impl.createDocument (null, “calles de San Pedro Cholula”,
null);
Element element = doc. createElement(“calle”);
```

4.1.2.1 Estructura de DOM

La base de DOM es un modelo de árbol. Este hace una representación de todo el documento en memoria. El documento se le regresa en forma de árbol, y todo esto es construido sobre la interfaz **Node** del DOM. De esta interfaz se derivan varias interfaces específicas de XML como son **Element**, **Document**, **Attr** y **Text** [McLaughlin, 2001]. En un archivo XML típico se conseguirá una estructura que se ve como la de la siguiente figura:

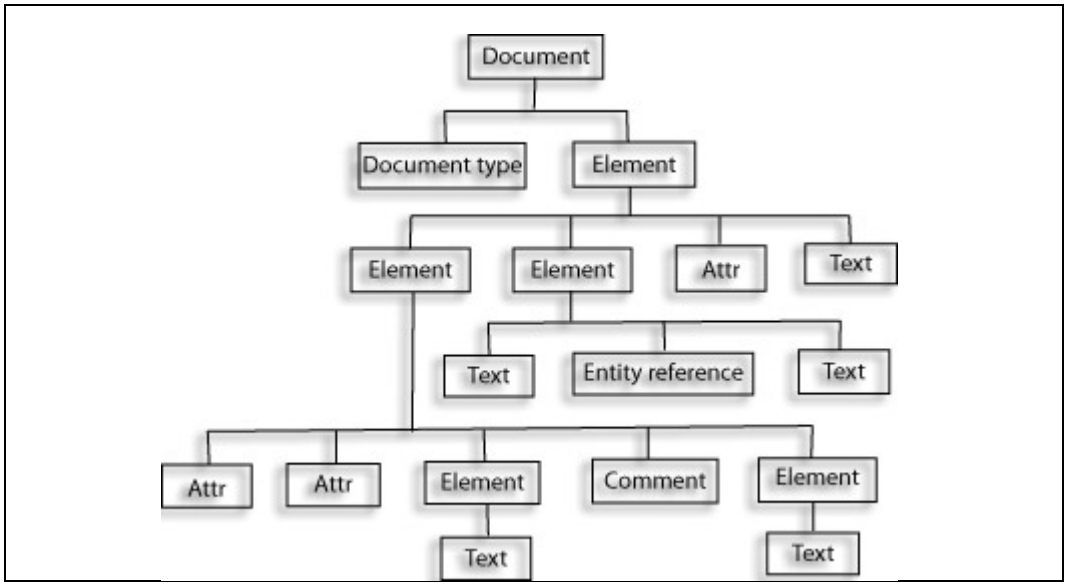

Figura 4.1 Estructura en DOM representando XML [McLaughlin, 2001]

En el caso en que un nodo de tipo **Element** tenga un valor textual, en lugar de obtener el valor textual del nodo a través del nodo **Element**, obtendremos un nodo hijo de tipo **Text**, a su vez el nodo hijo del nodo de tipo **Text** será el valor del elemento. Se puede navegar el árbol hacia arriba o abajo sin preocuparse sobre el tipo de estructura específico.

En DOM el foco de interés está en la salida a partir del proceso de *parseo*. Mientras el documento entero está siendo *parseado* y se va añadiendo a la estructura de árbol de salida, los datos no están en un estado que se pueda usar. La salida esperada del *parseo* para usar con la interfaz de DOM es un objeto **org.w3c.dom.Document**. Este objeto actúa como un manejador del árbol donde los datos XML se encuentran. Este objeto contiene todos y cada uno de los elementos del documento XML. A partir del árbol de DOM se puede convertir a una salida de XML bien formado, HTML o cualquier otro formato [McLaughlin, 2001].

El hecho de que todos los elementos del DOM extiendan de **Node** nos permite manejar la salida de todos los tipos del DOM de manera sencilla y permite la recursividad. Se puede crear y modificar el árbol DOM, se pueden manejar comentarios, *namespaces*, aunque presenta algunos problemas al hacer modificaciones y querer añadir *namespaces* a algún archivo existente. También define módulos (en el nivel dos) para

manejo de XML (extendiendo del nivel 1), Vistas (modelo de *scripts* para dinámicamente actualizar una estructura DOM), Eventos, Estilo (CSS), Travesia/Rango, HTML (tratándose en formato DOM).

El DOM en nivel tres además de lo mencionado anteriormente tiene mejoras a los niveles anteriores como la declaración de XML donde ya están disponibles métodos para manejar la versión, el estado del *standalone* y la codificación. También maneja comparación entre nodos, y define métodos para manejar nodos, y separar cualquier estructura DOM y determinar como se relaciona con otra. La adición más importante en este nivel es el "bootstrapping" para evitar la dependencia de algún vendedor específico.

4.1.2.2 Desventajas.

Debido a que genera un árbol DOM del documento en memoria puede consumir muchos recursos. Aunque algunos API's resuelven esto no expandiendo los nodos que no se están utilizando en memoria, sin embargo, esto hace que se consuma más tiempo de procesador. Otra desventaja es que su utilización es algo compleja y el simple hecho de crear o añadir un elemento en el documento puede requerir de varios pasos.

4.1.3 JDOM

"Creo que JDOM rompe muchas barreras entre Java y XML y lo hace ver más natural" Simon St. Laurent, Autor de XML Elements of Style.

JDOM es una biblioteca *open source* o de formato abierto para manipulación optimizada de datos XML con Java. Es un API puro de Java basado en árbol para crear, *parsear*, manipular y serializar documentos XML. Fue inventado por Brett McLaughlin y Jason Hunter en la primavera del 2000.

JDOM surgió porque DOM y SAX no eran suficientes. SAX no permite modificaciones al documento, acceso aleatorio o capacidades de salida (output) y requiere a menudo construir una maquina de estado. DOM no es muy familiar para el programador en Java, pues está definido en IDL un común denominador a través de lenguajes más bajo [Hunter, 2002].

Así como DOM, JDOM representa un documento XML como un árbol compuesto por elementos, atributos, comentarios, instrucciones de proceso, nodos de texto, secciones CDATA y así sucesivamente. El árbol completo está disponible todo el tiempo. A diferencia de SAX, JDOM puede acceder cualquier parte del árbol en cualquier momento. En contraste a DOM, todos los diferentes tipos de nodos del árbol son representados por clases concretas en lugar de interfaces. Además, no hay una interfaz o clase genérica **Node** (DOM) de la que extiendan o implementen todas las diferentes clases de nodo (Esto hace que las operaciones para navegar el árbol y buscar sean más incómodas de lo que son en DOM) [Harold 2002].

JDOM fue escrito para y con Java. Usa constantemente las convenciones de codificación de Java y la biblioteca de clase, todas las clases primarias de JDOM tienen equals(), toString(), y métodos hashCode(), todas ellas implementan las interfaces **Cloneable** y **Serializable**. Almacenan los hijos de un **Element** o de un objeto **Document** en una **java.util.List** [Harold 2002]. Se hace especial énfasis en este punto ya que a un programador de Java le es mucho más fácil y cómodo trabajar con un API que le es familiar.

JDOM no incluye por si mismo un *parser*. En vez de eso depende de un *parser* de SAX con un manejador de contenido común para *parsear* documentos y construir modelos JDOM a partir de estos. JDOM viene con Xerces 1.4.4, pero puede trabajar de igual manera con cualquier *parser* compatible con SAX2 incluyendo Crimson, AElfred (el parser de Oracle XML para Java), Piccolo, Xerces-2. Cualquiera de estos puede leer un documento XML y ponerlo en JDOM y puede también convertir objetos Documento de DOM en objetos de Documento de JDOM. Es así mismo útil para hacer un pipe de la salida de programas DOM existentes en la entrada de un programa en JDOM. Sin embargo, si se está trabajando con un flujo de datos de XML que se lee de un disco o de la red, es preferible usar SAX para producir el árbol JDOM y así evitar la sobrecarga de construir el árbol en memoria dos veces en dos representaciones diferentes [Harold 2002].

Los datos para construir el árbol en JDOM pueden provenir de una fuente que no sea XML, como una base de datos o literales en un programa de java. JDOM checa todos

los datos para ver que se construya un documento XML bien formado. Una vez que el documento se ha cargado en memoria, JDOM puede modificar el documento y el árbol generado es completamente de lectura-escritura a diferencia de DOM donde si pueden haber secciones de sólo lectura. [Harold, 2002]

Finalmente una vez que se ha terminado de trabajar con el documento en memoria, JDOM permite serializar el documento de nuevo al disco o a un flujo de bytes. JDOM provee numerosas opciones para especificar la codificación, los caracteres de fin de línea, espaciado, y otros detalles de la serialización. Otra alternativa, es producir una secuencia de eventos SAX o un documento DOM de salida en lugar de convertir el documento en un flujo de datos [Harold, 2002].

4.1.3.1 CONSTRUYENDO ELEMENTOS EN JDOM

En JDOM la construcción de elementos es intuitiva, lo cual no siempre es lo mismo en SAX y casi nunca sucede con DOM. En la tabla 3.1 se muestra una tabla de comparación en la complejidad entre JDOM y DOM para la creación del elemento *<calle/>*.

En JDOM se crea de la siguiente manera:	**En DOM de la siguiente manera:**
`Element element = new Element ("calle");`	`DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance( ); DOMImplementation impl = builder.getDOMImplementation( ); Document doc = impl.createDocument (null, "calles de San Pedro Cholula", null); Element element = doc. createElement("calle");`

Tabla 4.1 Comparación de complejidaed entre JDOM y DOM.

Existen algunos limites a la simplicidad, la mayoría establecidos por XML. En general si algo es complejo en JDOM es porque es igualmente complejo en XML [Harold, 2002]. Veamos un ejemplo más complejo, en el cual se manejan elementos hijos y texto:

XML	Creacón del XML en JDOM

<esquina>
<calle>49</calle>
<calle>5</calle>
</esquina>
	Primero se tienen que crear los 3 objetos Element, dos para las calles y uno para la esquina:
	
	`Element esquina = new Element("esquina");`
	`Element calleUno = new Element("calle");`
	`Element calleDos = new Element("calle");`
	
	Después se tienen que agregar los números de las calles en el orden que se quiere que aparezcan:
	
	`calleUno.setText("49");`
	`calleDos.setText("5");`
	`esquina.addContent(calleUno);`
	`esquina.addContent(calleDos);`

Tabla 4.2 Ejemplo de Creación de un fragmento de XML en JDOM

4.1.3.2 Creando Documentos XML con JDOM

Supóngase que se quiere crear a partir de JDOM el siguiente documento XML:

```
<?xml version="1.0">
<calle>59 Oriente </calle>
```

Como todos los documentos en XML tienen un elemento raíz, necesitamos crear el elemento raíz (*root*) calle primero y luego usarlo para crear el documento. Las llamadas a los métodos pueden estar en cualquier orden (sin olvidar el orden lógico de la creación del Elemento antes de aplicar algún método sobre él). Inclusive se puede crear el objeto Documento usando el constructor sin argumentos y luego asignarle su elemento raíz con el método `setRootElement()`. Cabe señalar que de este modo el documento esta temporalmente en un estado de malformación y en caso de no asignarle un elemento raíz o añadir contenido se provocara una excepción en java de estado ilegal (**java.lang.IllegalStateException**).

```
Element root = new Element ("calle");
```

```
root.setText("59 Oriente");
Document doc = new Document(root);
```

El objeto **Element** no se asocia inicialmente a ningún documento. Es libre, al contrario de DOM donde siempre todos los nodos son parte de algún documento, JDOM permite que los nodos permanezcan aparte si eso es útil. Sin embargo, JDOM no permite que un nodo sea parte de dos documentos al mismo tiempo. Antes de que un elemento pueda ser transferido a un nuevo documento, debe primero ser separado de su anterior documento usando el método `detach()` [Harold 2002].

4.1.3.3 ESCRIBIENDO DOCUMENTOS XML CON JDOM

Una vez creado el documento, tal vez se quiera serializar a un *socket* de red, archivo, cadena, o algún otro flujo de datos. La clase **org.jdom.output.XMLOutputter** de JDOM hace esto de una forma estándar. Se puede crear un objeto **XMLOutputter** con el constructor sin argumentos y después escribir el documento a un **OutputStream** con su método `output()` [Harold, 2002].

Además de flujos de datos, también se puede dar a un documento salida a un **java.io.Writer**. Es recomendado usar un **OutputStream** porque generalmente no se puede determinar la codificación subyacente de un **Writer** y asignar la declaración de codificación de manera acorde. **XMLOutputter** también puede escribir elementos, atributos, secciones CDATA, y todas las demás clases de nodo de JDOM. Finalmente si no se quiere escribir en un flujo de caracteres o en un **Writer** se puede usar el método `outputString()` para almacenar el documento XML o el nodo en una cadena. Esto es muy útil cuando se quieren pasar datos XML a través del sistema o la aplicación que no soporte XML. En la tabla 4.3 se muestra un programa que integra todo esto.

Los documentos creados con JDOM pueden tener DTD's para ser validados. JDOM no ofrece un modelo objeto completo para DTD's, sin embargo permite que se haga referencia a un DTD o que se agregue un subconjunto interno de DTD para los documentos [Harold, 2002]. Cabe aclarar que JDOM no verifica si el documento es válido.

Clase Esquina.java	Corrida

`import org.jdom.*;`
`import org.jdom.output.XMLOutputter;`
`import java.io.IOException;`
 	
`public class Esquina{`
 	
`  public static void main(String[] args) {`
`  Element root = new Element("Esquina");`
 	
`  for (int i = 1; i <= 2; i++) {`
`  Element calle = new Element("calle");`
`  calle.setAttribute("id",`
`String.valueOf(i));`
`  calle.setText("calle "+`
`String.valueOf(i));`
`  root.addContent(calle);`
`  }`
 	
`  Document doc = new Document(root);`
`  // Lo serializa en el System.out`
`  try {`
`  XMLOutputter serializer = new`
`XMLOutputter("  ",  true);  //maneja`
`espaciado y salto de linea`
`  serializer.output(doc, System.out);`
`  }`
`  catch (IOException e) {`
`  System.err.println(e);`
`  }`
`  }`
`}`
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<Esquina>`
	`  <calle id="1">calle 1</calle>`
	`  <calle id="2">calle 2</calle>`
	`</Esquina>`

Tabla 4.3 Programa de Ejemplo de creación de un documento XML con JDOM

JDOM también soporta el uso de *namespaces*. La regla básica de JDOM sobre *namespaces* es que cuando un elemento o atributo está en un *namespace*, se da el nombre local, el prefijo, y el URI, en ese orden. Si el elemento está en el *namespace* por defecto, se omite el prefijo. No se necesitan agregar atributos para las declaraciones de *namespaces*. El *outputter* entenderá los lugares razonables para ponerlos al serializar el documento [Harold, 2002].

Supóngase que se quiere crear el siguiente documento de MathML en JDOM:

```
<?xml version="1.0"?>
<mathml:math xmlns:mathml="http://www.w3.org/1998/Math/MathML">
   <mathml:mrow>
  …
 </mathml:mrow>
</mathml:math>
```

Cada Elemento se tiene que crear de la siguiente manera:

```
Element root = new Element ("math", "mathml",
"http://www.w3.org/1998/Math/MathML");
```

En el elemento **mrow** se pone **mrow** en lugar de **math** y así sucesivamente con los demás elementos.

4.1.3.4 Leyendo Documentos XML con JDOM

JDOM puede leer documentos XML existentes de archivos, *sockets* de la red, cadenas o cualquier cosa que se pueda enganchar a un flujo de datos (*stream*), o lector (*reader*). JDOM no incluye su propio *parser* nativo, en vez de eso depende de *parsers* SAX2 como Xerces y Crimson.

Según Harold (2002) los pasos para trabajar con un documento XML existente usando JDOM son los siguientes:

1. Construir un objeto **org.jdom.input.SAXBuilder** usando un simple constructor sin argumentos.
2. Invocar el método constructor `build()` para construir un objeto Documento de un **Reader**, **InputStream**, **File**, **URL**, o **String** conteniendo un system ID.
3. Si hay algún problema al leer el documento, una **IOException** es lanzada. Si hay algún problema mientras se construye el documento, una **JDOMException** es lanzada.
4. Si no, navegue el documento usando los métodos para la clase **Document**, **Element** y las otras clases de JDOM.

La clase **SAXBuilder** representa el *parser* de XML subyacente. *Parsear* un documento a partir de un URL es simple. Solo se crea un objeto **SAXBuilder** con el constructor sin argumentos y se pasa la cadena del URL a su método `build()`, este retorna un objeto JDOM **Document**. Ejemplo:

```
SAXBuilder parser = new SAXBuilder();
Document doc = parser.build("http://www.udlap.mx/");
```

Si se desea establecer un *parser* distinto al que JDOM tiene por defecto, se tiene que especificar todo el nombre del paquete de la clase **XMLReader** como primer argumento en el constructor, ejemplo:

```
SAXBuilder parser = new SAXBuilder
("org.apache.crimson.parser.XMLReaderImpl");
Document doc = parser.build("http://www.udlap.mx/");
```

4.1.3.5 Navegando árboles JDOM

En JDOM, la navegación en su mayoría es a través de los métodos de la clase **Element**. Todos los hijos de cada **Element** están disponibles como un **java.util.List** regresado por el método `getContent()`. Sólo los elementos hijos de cada **Element** están disponibles en un **java.util.List** regresado por el método `getChildren()`, que solo regresa elementos, para obtener nodos de texto, comentarios e instrucciones de procesamiento se usa *`getContent()`* (JDOM usa la palabra *children* para referirse únicamente a los elementos hijos).

Debido a que JDOM usa el API Java Collections para manejar el árbol, esto es, simultáneamente, muy polimórfico (todo es un objeto y se debe hacer un *cast* al tipo correcto antes de poder usarlo) y no suficientemente polimórfico (no existe una interfaz genérica o superclase para la navegación como la clase **Node** de DOM). En consecuencia se tienen que hacer numerosas pruebas con el instanceof() y haciendo el *casting* al tipo determinado. Existe la interfaz **Filter** que puede simplificar algunos de los problemas del polimorfismo y *casting*, pero esto no seguirá dejando que se avance más de un nivel cada vez [Harold, 2002].

4.1.3.6 Comunicación con programas DOM

JDOM no es un acrónimo y no es directamente compatible con DOM. Un **Element** en JDOM no es un **Element** de DOM. La clase **Element** de JDOM no implementa la Interfaz **Element** de DOM. No se puede pasar un **Element** de JDOM a un método que espera recibir un **Element** de DOM y viceversa, lo mismo pasa con las demás clases de JDOM.

JDOM permite convertir documentos DOM a documentos JDOM y viceversa. Si se tiene un objeto **Document** de DOM, la clase **org.jdom.input.DOMBuilder** se usa para generar el JDOM equivalente. El objeto DOM original no es alterado de ninguna manera, pues los cambios al documento de JDOM no afectan el documento DOM del cual fue construido. De la misma manera cambios futuros al documento DOM original no afectarán al documento JDOM. Para convertir un documento de JDOM a DOM se utiliza la clase **org.jdom.output.DOMOutputter** [Harold, 2002].

4.1.3.7 Comunicación con programas SAX

JDOM trabaja muy bien con *parsers* SAX, que es casi el modelo de eventos ideal para construir un árbol JDOM. Cuando el árbol está completo, JDOM hace fácil su navegación, ejecutando eventos de SAX mientras se avanza. Como SAX es muy rápido y eficiente en memoria, no agrega mucha carga extra a los programas JDOM [Harold, 2002].

Cuando se lee un archivo o flujo de datos a través de un *parser* SAX, se pueden establecer varias propiedades en el *parser* incluyendo **ErrorHandler**, **DTDHandler**, **EntityResolver**, y cualquier característica o propiedad que sean soportadas por el SAX **XMLReader** subyacente. **SAXBuilder** incluye varios métodos que sólo delegan estas configuraciones al **XMLReader** subyacente. Para mayores detalles consultar el API [JDOM API, 2002].

4.1.3.8 Integración con Java

JDOM es de todos los API's para procesar XML, el más enfocado a Java. Los desarrolladores de JDOM han dado muchas ideas para mostrar exactamente como las clases de JDOM se ajustan a las clases comunes de Java como son el API Java Collections, RMI, y el marco de trabajo de entrada/salida. Al contrario de DOM, el comportamiento de los objetos JDOM está muy bien definido con respecto a operaciones de Java como clonar y serializar. [Harold, 2002]

Todas las clases base de JDOM (**Element**, **Attribute**, etc.) implementan el método `equals()`.Como JDOM checa la igualdad basada en la identidad del objeto, la implementación de *`hashCode()`* por defecto, heredada de **java.lang.Object** es suficiente. Sin embargo, para prevenir que las subclases violen este contrato, el método `hashCode()` es implementado como un método final (que ya no se hereda), que llama a `super.hashCode()` [Harold, 2002].

El unir la funcionalidad de un SIG, con las tecnologías XML y el estándar GML, nos da como resultado la generación de información altamente transportable y útil. Al estar disponible en web se hace disponible a muchos usuarios.

Para concluir, cabe señalar que estas tecnologías se dirigen a diferentes niveles de abstracción y proporcionan diferentes niveles de utilización para el programador Java. SAX, DOM y JDOM podrían relacionarse unas con otras. Por ejemplo, un constructor de documentos JDOM podría usar un analizador SAX para generar un árbol JDOM desde un documento XML. Como se apilan, un desarrollador de aplicaciones XML podría verse tentado a elegir una de ellas para conseguir el mismo resultado, pero hacer esto traerá dificultades y pérdidas entre el rendimiento, el uso de memoria y la flexibilidad.

DOM y especialmente JDOM podrían traer simplicidad y eficiencia en la edición de grandes modelos de documentos en memoria. En algunos casos, un modelo de objeto de documento podría ser elegible como la estructura de datos principal de la aplicación. Todos estos APIs junto con tecnologías como XSLT, necesitan ser considerados cuando implementamos una aplicación XML. Además con el API SAX 2.0, y la ínter cambiabilidad (como fuente y resultado) de SAX y JDOM en algunos APIs, se pueden construir complejas tuberías de procesamiento XML combinando implementaciones estándar y personalizadas. A continuación se verá Struts, un marco

de trabajo que implementa la arquitectura Modelo-Vista-Controlador (Ver Apéndice B) y que está siendo cada vez más utilizado para servicios web en el mundo entero.

4.2 JAKARTA STRUTS

Con la llegada de los Servlets muchos programadores se dieron cuenta que eran una buena alternativa, eran más rápidos y potentes que el CGI estándar, además de ser portables e infinitamente extensibles. Sin embargo se tenían que escribir gran cantidad de sentencias `println( )` para enviar HTML al navegador, la respuesta a este problema fueron las Java Server Pages en las cuales se podía mezclar fácilmente HTML con código Java y tener todas las ventajas de los Servlets.

Con el tiempo se dieron cuenta que centralizar las aplicaciones web Java a JSP (Modelo 1) no resolvía problemas como el del control de flujo y otros endémicos de las aplicaciones web. Debido a esto se propuso un nuevo modelo en el cual se utilizan los Servlets para ayudar con el control de flujo y las JSP para enfocarse en escribir HTML esto se dio a conocer como Modelo 2, el cual sigue el clásico patrón de diseño Modelo-Vista-Controlador (ver Apéndice B) de SmallTalk. En la actualidad es común usar los términos Modelo 2 y MVC indistintamente.

El proyecto Struts fue lanzado en mayo del 2000 por Craig R. McClanahan para proporcionar un marco de trabajo (*framework*) MCV estándar a la comunidad de Java. En julio del 2001, se liberó Struts 1.0 [McClanahan, 2002].

4.2.1 DEFINIENDO STRUTS

Struts es un marco de trabajo *open source* para implementar aplicaciones web con Servlets y JSP según el patrón arquitectónico Modelo-Vista-Controlador. Es un subproyecto de Jakarta que funciona sobre cualquier servidor de aplicaciones web que implemente los API's de Servlets y JSP. Actualmente se esta estandarizando JavaServer Faces [ServerFaces, 2003], que formará parte de J2EE. Struts estará parcialmente integrado con JSTL [JSTL, 2003] y JavaServer Faces, y la integración será completa en Struts 2.0 [Bellas, 2002].

Struts provee su propio componente Controlador y lo integra con otras tecnologías para proveer el Modelo y la Vista. Struts puede interactuar con cualquier tecnología estándar de acceso a datos, incluyendo Enterprise Java Beans, JDBC, y Object Relational Bridge. Para la Vista, Struts trabaja bien con Java Server Pages, incluyendo JSTL y JSF, así como Templates Velocity, XSLT, y otros sistemas de presentación [Struts, 2003].

Struts nos proporciona un Servlet Front Controller (ver figura 4.5) y clases relacionadas, una librería de *tags* JSP muy completa, un *pool* genérico de conexiones a una BD relacional al implementar **javax.sql.DataSource**, así como soporte a la internacionalización de mensajes [Bellas, 2002]. Las librerías de *tags* que nos proporciona Struts son:

- Bean, la cual tiene la función de imprimir el valor de las propiedades de JavaBeans de manera segura y de soportar internacionalización de mensajes.
- HTML tiene la función de generar código html básico, campos de entrada en formularios y ligas.
- Logic realiza las funciones de control de flujo.
- Template su función es la de ofrecer plantillas de páginas JSP para la Vista.

Es importante señalar que en esta sección y sus subsecciones se explica y define Struts en su versión 1.0.2. Recientemente ha salido una nueva versión (la 1.1) en la cual seguramente cambiarán algunos detalles de diseño e implementación. Si alguno de los lectores de este proyecto de tesis decide usar otra versión de Struts diferente a la aquí mencionada se recomienda revisar los manuales de usuario de la versión elegida.

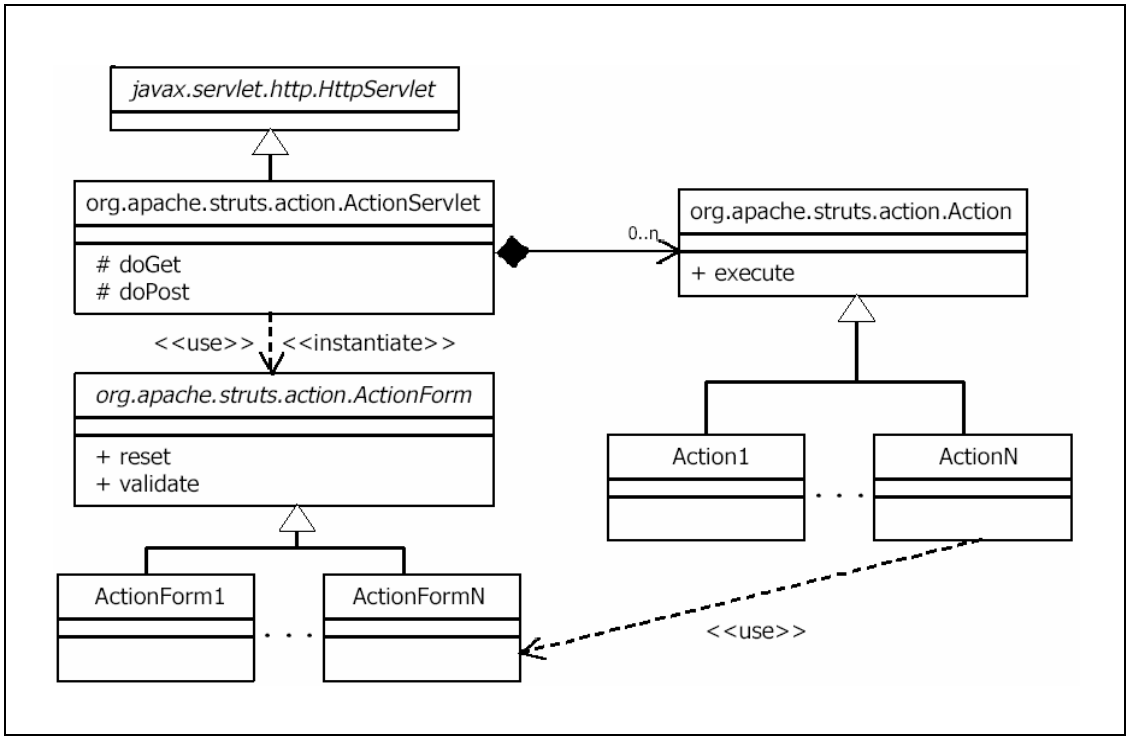

Figura 4.2 Esquema del Patrón Front Controller en Stuts

4.2.2 EL PATRÓN FRONT CONTROLLER EN STUTS

En esta sección se explicará como se implementó el patrón *front controller* en el marco de trabajo de Struts, el diagrama de la figura 4.2 servirá de apoyo para esta sección.

4.2.2.1 ACTION SERVLET

Struts provee un Servlet Front Controller por medio de la clase **org.apache.struts.action.ActionServlet**; este une y enruta solicitudes http a otros objetos del marco de trabajo incluyendo JavaServer Pages y subclases *org.apache.struts.Action* proporcionadas por el desarrollador Struts. Una vez inicializado, el controlador analiza el archivo de configuración de recursos [McClanahan, 2002]. Más adelante se hablará sobre las funciones de este archivo. En

"web.xml" se especifica que todas las URLs que impliquen procesamiento por GET o POST vayan a este *servlet* [Bellas, 2002].

4.2.2.2 Clases ActionForm

Con Struts se pude definir un conjunto de clases *bean* de formulario, extendiéndolas de **org.apache.struts.action.ActionForm**, y almacenar fácilmente los datos de un formulario de entrada en estos *beans*. El *bean* de formulario puede usarlo una JSP para recoger datos del usuario, por un objeto **Action** para validar los datos introducidos por el usuario y luego de nuevo por la JSP para rellenar los campos del formulario. En el caso de validación de errores, Struts tiene un mecanismo compartido para lanzar y mostrar mensajes de error. [McClanahan, 2002].

4.2.2.3 Clases Action

Los objetos *action* tienen acceso al *servlet* controlador de la aplicación y por eso tienen acceso a los métodos del *servlet*, acceden a los parámetros del *request*, directamente o vía el **ActionForm** correspondiente. Un objeto *action* realiza una operación invocando un método de un *bean* de lógica de negocio y deja el resultado devuelto por el método en la *request* o en la sesión. Devuelve un objeto **ActionForward**, que representa la URL que hay que visualizar a continuación (*sendRedirect* o *foward*).

En una aplicación Struts, la mayoría de la lógica del negocio se puede representar usando JavaBeans. Un objeto **Action** puede llamar a las propiedades de un JavaBean sin conocer realmente como funciona. Esto encapsula la lógica del negocio, para que el objeto **Action** pueda enfocarse en el manejo de errores y donde reenviar el control. Para permitir su reutilización en otras plataformas, los JavaBeans de lógica de negocio no deberían referirse a ningún objeto de aplicación web. El objeto **Action** debería traducir los detalles necesarios de la solicitud http y pasarlos a los *beans* de la lógica del negocio como variables normales de Java [McClanahan, 2002].

4.2.2.4 El archivo de configuración

En este archivo (normalmente se llama "struts-config.xml") se definen:

- Las clases **ActionForm** que usa nuestra aplicación por su nombre lógico (ej: *gisForm*) y el nombre completo de la clase (ej: pruebaStruts.GISForm).
- Los nombres lógicos de URLs de *forwards* globales por el nombre que usan las acciones cuando devuelven un *action forward* (ej: conexion) y el URL a invocar (ej: /conexión.jsp).
- Los URLs que implican procesamiento (ActionMappings) por el nombre lógico de la ActionForm asociada (ej: gisForm), el URL de tipo *path* relativo al contexto (no llevan el ".do" final ej: /logout), el nombre completo de la clase **Action** (ej:pruebaStruts.LogoutAction).

Cuando el *servlet* **ActionServlet** arranca (*init*), lee el archivo de configuración, crea una instancia única de cada clase **Action**, es decir, no se crea una instancia de una clase **Action** por cada petición que se recibe, tienen que ser *thread-safe* (misma situación que cuando se trabaja con *servlets*) [Bellas, 2002].

4.2.3 LA ARQUITECTURA MODELO-VISTA-CONTROLADOR CON STRUTS

Siguiendo el patrón de diseño Modelo-Vista Controlador, las aplicaciones Struts tienen 3 componentes principales: un *servlet* controlador, proporcionado por el propio Struts, páginas JSP (la vista), y la lógica de negocio de la aplicación (modelo). A continuación se explicará esto más a detalle.

4.2.3.1 EL MODELO

En general, el desarrollador de componentes del Modelo se enfocará en la creación de JavaBeans que soporten todos los requerimientos de funcionalidad. Estos *beans* generalmente pueden clasificarse en 2 categorías, que se explicarán más adelante:

- Beans **ActionForm** o de manejo de Formularios.
- Beans de estado del Sistema y Beans de Lógica de Negocio.

4.2.3.1.1 BEANS ACTIONFORM

Cabe señalar que mientras que los *beans* de formulario muy a menudo tienen propiedades que corresponden a propiedades en los *beans* de "modelo", los *beans* de formulario por si mismos deben ser considerados un componente del "controlador", debido a que a estos se les permite transferir datos entre las capas del Modelo y la Vista [McClanahan, 2002].

El marco de trabajo Struts generalmente asume que hemos definido un *bean* de formulario (que extiende de la clase **ActionForm**), por cada formulario de entrada necesario en nuestra aplicación. Si se declaran dichos *beans* en el archivo de configuración (esto se explicará más a detalle en la sección referente al controlador), el **ActionServlet** de Struts realizará automáticamente los siguientes servicios antes de llamar al método **Action** apropiado:

- Checa en la sesión de usuario si hay un ejemplar de un *bean* de la clase apropiada bajo la clave apropiada. Si no está disponible dicho *bean* en el scope de la sesión, se crea uno nuevo automáticamente y se añade a la sesión del usuario.
- Por cada parámetro de la solicitud cuyo nombre corresponda con el nombre de una propiedad del *bean*, se llamará al método set() correspondiente.
- El *bean* de formulario actualizado será pasado al método perform() de la clase **Action** cuando es llamado, haciendo que estos valores estén disponibles inmediatamente.

El objeto **ActionForm** también ofrece un mecanismo de validación estándar, implementando el método validate(). Con esto se asegura que están presentes todas las propiedades requeridas y que contienen valores válidos.

Cabe señalar que un "formulario", en el sentido discutido aquí, no corresponde necesariamente con una sola página JSP en la interfaz de usuario. Es común en muchas aplicaciones tener un "formulario" (desde la perspectiva del usuario) que se extienda sobre múltiples páginas. Un ejemplo de esto, son las interfaces de tipo "wizard" que se utilizan comúnmente cuando se instalan nuevas aplicaciones.

[McClanaham, 2002] aconseja definir un sólo **ActionForm** que contenga las propiedades de todos los campos, sin importar que página de campo se está mostrando actualmente. De igual forma, las distintas páginas del mismo formulario deberían ser reenviadas a la misma clase **Action**. Si se siguen estas sugerencias, los diseñadores de páginas podrán reordenar los campos entre varias páginas, frecuentemente sin requerir que se cambie la lógica de procesamiento.

4.2.3.1.2 Beans de estado del sistema y beans de lógica de negocio

En términos gramáticos se puede pensar en la información de estado como nombres (cosas) y las acciones como verbos (cambios del estado de estas cosas). El estado real de un sistema normalmente está representado por un conjunto de una o más clases de Java Beans, cuyas propiedades definen el estado actual.

Las aplicaciones de gran escala normalmente representarán un conjunto de posibles acciones lógicas de negocio con métodos que pueden ser llamados sobre los *beans* que mantienen su información de estado. Se debería encapsular la lógica funcional de la aplicación como llamadas a métodos en JavaBeans diseñados para este propósito. Estos métodos pueden ser parte de las mismas clases usadas para los *beans* de estado del sistema, o podrían estar en clases separadas dedicadas a realizar la lógica.

Para sistemas de pequeña escala, o para información de estado que no necesita guardarse durante mucho tiempo, un conjunto de *beans* de estado del sistema podría contener todos los conocimientos que el sistema tiene sobre esos detalles particulares, o como sucede con frecuencia, los *beans* de estado del sistema representarán información que está almacenada permanentemente en alguna base de datos externa y son creados o eliminados de la memoria del servidor cuando se necesita. Los JavaBeans Enterprise de Entidad también se usan para esto en aplicaciones de gran escala [McClanaham, 2002].

Para una reutilización máxima del código, los beans de la lógica del negocio deberían ser diseñados e implementados para que no sepan que están siendo ejecutados en un entorno de aplicación Web. Se debe diseñar la aplicación de tal manera que las clases **Action** (parte del rol del Controlador), traduzcan toda la información requerida desde

la solicitud HTTP que está siendo procesada en llamadas a métodos setXxx() de propiedades de nuestros *beans* de lógica de negocio, después de que se pueda hacer una llamada a un método execute(). Dicha clase de lógica de negocio podría reutilizarse en entornos distintos al de la aplicación Web para el que fue construida en un principio [McClanaham, 2002].

Dependiendo de la complejidad y del ámbito de la aplicación, los *beans* de lógica de negocio podrían ser JavaBeans ordinarios que interactúan con *beans* de estado del sistema que son pasados como argumentos, o JavaBeans ordinarios que acceden a una base de datos usando llamadas JDBC. Para grandes aplicaciones, estos *beans* frecuentemente ofrecerán JavaBeans Enterprise (EJBs) con o sin estado en su lugar.

4.2.3.2 La Vista

Esta capa está creada usando tecnología JavaServer Pages (JSP) principalmente. En particular Struts proporciona soporte para construir aplicaciones internacionalizadas, así como para interactuar con formularios de entrada.

4.2.3.2.1 Mensajes Internacionalizados

La explosión del desarrollo de aplicaciones basadas en tecnologías Web, así como el despliegue de dichas aplicaciones sobre Internet y otras redes accesibles, han hecho que los límites nacionales sean invisibles en muchos casos. Esto se ha traducido en la necesidad de que las aplicaciones soporten la internacionalización (frecuentemente llamada "i18n" porque 18 es el número de letras entre la "i" y la "n") y localization. Struts se construye sobre la plataforma Java proporcionada para construir aplicaciones internacionalizadas y localizadas [McClanaham, 2002].

La parte de la Vista de una aplicación basada en Struts generalmente está construida usando tecnología JavaServer Pages (JSP). Las páginas JSP pueden contener texto HTML estático (o XML) llamado "plantilla de texto", además de la habilidad de insertar contenido dinámico basado en la interpretación (en el momento de solicitud de la página) de etiquetas de acción especiales. Además, hay una facilidad estándar para definir etiquetas propias, que están organizadas en "librerías de etiquetas

personalizadas". Struts incluye una extensa librería de etiquetas personalizadas que facilitan la creación de interfaces de usuario que están completamente internacionalizados, y que interactúan amigablemente con *beans* **ActionForm** que son parte del Modelo del sistema [McClanaham, 2002].

Para una aplicación internacionalizada, se tienen que seguir los pasos descritos en el documento *Internationalization* del paquete de documentación del JDK de nuestra plataforma para crear un archivo de propiedades que contenga los mensajes para cada idioma [McClanaham, 2002]. Por ejemplo si nuestro código fuente se crea en el paquete *gisudla.mipaquete*, para crear un paquete de recursos llamado *gisudla.mipaquete.Misrecursos,* creamos los siguientes archivos en el directorio *gisudla/mipaquete*:

- *Misrecursos.properties*: Este archivo contiene los mensajes del idioma por defecto de nuestro servidor. Por ejemplo si el idioma es español podríamos definir una entrada como esta: "button.reset = Limpiar".
- *Misrecursos_xx.properties*: Contiene los mismos mensajes del idioma cuyo código ISO es "xx". Así para un navegador en inglés tendríamos esta entrada: "button.reset= Reset". Se pueden tener archivos de recursos para tantos idiomas como se necesite.

Cuando configuramos el *servlet* controlador en el descriptor de despliegue de la aplicación Web ("web.xml"), una de las cosas que necesitamos definir en un parámetro de inicialización es el nombre base del paquete de recursos para la aplicación. En el caso descrito arriba, sería *gisudla.mypaquete.Misrecursos.* A continuación en la figura 4.3 se mostrarán las líneas necesarias para definir en el archivo "web.xml" donde encontrar el paquete de recursos de la aplicación.

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>application</param-name>
    <param-value>gisudla.mypaquete.Misrecursos</param-value>
  </init-param>
  <.../>
```

```
</servlet>
```

Figura 4.3 Código del archivo web.xml para iniciar el servlet controlador de Struts y definir la ubicación del paquete de recursos de la aplicación

4.2.3.2.2 CONSTRUYENDO FORMULARIOS CON STRUTS

Como se sabe la tendencia hoy en día es el desarrollo de aplicaciones interactivas con interfaces amigables, fáciles de entender y que capten el interés del usuario, por lo tanto es necesario que estas aplicaciones tengan ciertos comportamientos, y uno de estos está relacionado con el manejo de errores. Si el usuario comete un error, la aplicación debería permitirle corregir sólo lo que necesita ser modificado, sin tener que re-introducir cualquier parte del resto de la información de la página o formulario actual.

Struts proporciona una facilidad comprensiva para construir formularios, basada en la facilidad de las Librerías de Etiquetas Personalizadas de JSP 1.1, sin la necesidad de referirnos explícitamente al formulario JavaBean del que se recupera el valor inicial. Esto lo maneja automáticamente el marco de trabajo [McClanaham, 2002].

En la figura 4.4, se muestra un ejemplo completo de un formulario de login el cual ilustrará cómo Struts trata con los formularios de una forma más sencilla que usando sólo HTML y JSP estándar. Consideremos la siguiente página (basada en la aplicación de ejemplo incluida con Struts) llamada "logon.jsp":

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld"
        prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld"
        prefix="bean" %>
<html:html>
<head>
<title>
  <bean:message key="logon.title"/>
```

```
</title>
<body bgcolor="white">
<html:errors/>
<html:form action="/logon" focus="username">
<table border="0" width="100%"><tr>
    <th align="right">
      <html:message key="prompt.username"/></th>
    <td align="left">
      <html:text property="username" size="16"/>
    </td></tr><tr>
    <th align="right">
      <html:message key="prompt.password"/>
    </th>
     .......
</table>
</html:form>
</body>
</html:html>
```

Figura 4.4 Código de logon.jsp [McClanaham, 2002]

Los siguientes puntos ilustran las características clave del manejo de formularios en Struts, basadas en el ejemplo de la figura 4.4:

- La directiva *taglib* le dice al compilador de la página JSP donde encontrar el *tag library* descriptor para la librería de etiquetas Struts. En este caso, se esta usando "bean" como el prefijo que identifica las etiquetas de la librería struts-bean, y "html" como el prefijo que identifica las etiquetas de la librería struts-html. Se puede usar cualquier prefijo que se desee.
- Esta página usa varias ocurrencias de la librería de mensajes para buscar los strings de mensajes internacionalizados desde un objeto MessageResources que contiene todos los recursos de esta aplicación. Para que esta página funcione, se deben definir las siguientes claves de mensajes en estos recursos:
    - o logon.title - Título de la página de login.
    - o prompt.username - Un string para pedir el "Username:"
    - o prompt.password - Un string para pedir la "Password:"
    - o button.submit - Etiqueta para el botón "Submit"
    - o button.reset - Etiqueta para el botón "Reset"

Cuando el usuario entra, la aplicación puede almacenar un objeto **java.util.Locale** en la sesión de este usuario. Este **Locale** se usará para seleccionar mensajes en el idioma apropiado. Esto se hace sencillo de implementar dando al usuario una opción para elegir el idioma , simplemente cambiamos el objeto **Locale** almacenado, y todos los mensajes se modificarán automáticamente.

- Las banderas de error muestran cualquier mensaje de error que haya sido almacenado por un componente de lógica de negocio, o ninguna si no se ha almacenado ningún error. Esta etiqueta se describirá más adelante.
- La etiqueta *form* renderiza un elemento `<form>` HTML, basándose en los atributos especificados. También asocia todos los campos que hay dentro del formulario con un **FormBean** con ámbito de sesión que se almacena bajo la clave logonForm. El desarrollador de Struts proporciona la implementación Java para este *bean* de formulario, extendiendo la clase **ActionForm** de Struts. Este *bean* se usa para proporcionar valores iniciales para todos los campos de entrada que tienen nombres que corresponden con nombres de propiedades del *bean*. Si no se encuentra un *bean* apropiado, se creará uno nuevo automáticamente, usando el nombre de la clase Java especificado.
- La etiqueta *text* se renderiza como un elemento `<input>` de HTML del tipo "text". En este caso también se han especificado el número de caracteres y la posición a ocupar en la ventana del navegador. Cuando se ejecuta esta página, el valor actual es el de la propiedad username del bean correspondiente (es decir, el valor devuelto por `getUsername( )`).
- La etiqueta *password* se usa de forma similar. La diferencia está en que el navegador mostrará asteriscos, en lugar del valor de entrada, mientras el usuario teclea su password.
- Las etiquetas *submit* y reset generan los botones correspondientes en la parte inferior del formulario. Las etiquetas de texto para cada botón se crean usando la librería de mensajes, como las peticiones, para que estos valores sean internacionalizados.

Para entender como implementar las clases **ActionForm** y **Action** en Struts se recomienda ver el código fuente de las clases proporcionadas en la aplicación de ejemplo de Struts. El nombre de las clases de tipo **ActionForm** siempre serán del tipo

"nombre"+Form (ej:LoginForm) y las de tipo **Action** siempre serán del tipo "nombre"+Action (ej: **LoginAction**). Para fines de orden se recomienda usar el mismo "nombre" para relacionar a los *beans* de formulario con sus respectivas clases **Action**.

Struts tiene varias librerías de etiquetas (*tags*) útiles para crear presentaciones como lo son la librería de etiquetas "html", la cual contiene etiquetas para crear formularios de entrada struts así como otras etiquetas html útiles en la creación de interfaces de usuario. La librería "logic" la cual contiene etiquetas útiles para manejar la generación condicional de salida de texto, la librería "templates" la cual contiene etiquetas que definen un mecanismo de plantillas para crear las JSP's y la librería "bean" la cual realiza operaciones de soporte para los mensajes internacionalizados y propiedades de los JavaBeans. También se pueden crear y definir etiquetas personalizadas específicas a una aplicación determinada. Se recomienda consultar [Struts, 2003], para consultar más a fondo las funcionalidades que ofrecen estas librerías de etiquetas.

4.2.3.3 EL CONTROLADOR

La parte Controlador de la aplicación está enfocada en las solicitudes recibidas desde el cliente (normalmente un usuario ejecutando un navegador Web), decidiendo qué función de la lógica de negocio se va a realizar, y luego seleccionando el siguiente componente Vista apropiado. En Struts, el componente principal del Controlador es un *servlet* de la clase **ActionServlet**. Este *servlet* está configurado definiendo un conjunto de **ActionMappings**. Un **ActionMapping** define un *path* que se compara contra la URI solicitada de la solicitud entrante, y normalmente especifica el nombre totalmente calificado de una clase Action. Todas las **Actions** son subclases de **org.apache.struts.action.Action**. Las acciones encapsulan la lógica del negocio, interpretan la salida, y por último despachan el control al componente Vista apropiado para la respuesta creada [McClanaham, 2002].

Según McClanaham las principales responsabilidades con el controlador son:

- Escribir una clase **Action** por cada solicitud lógica que podría ser recibida (extendida desde **org.apache.action.Action**).
- Configurar un **ActionMapping** (en XML) por cada solicitud lógica que podría ser enviada, en el archivo "struts-config.xml".

- Actualizar el archivo del descriptor de despliegue de la aplicación Web (en XML) para nuestra aplicación para que incluya los componentes Struts necesarios.
- Añadir los componentes Struts apropiados a nuestra aplicación.

4.2.3.3.1 Clases Action

El objetivo de una clase **Action** es procesar una solicitud, mediante su método perform(), y devolver un objeto **ActionForward** que identifica dónde se debería reenviar el control (por ejemplo a una JSP) para proporcionar la respuesta apropiada. Antes de codificar las clases **Action** debemos tomar en cuenta algunos problemas de diseño que se pudiera presentar, [McClanaham, 2002] recomienda tomar en cuenta los siguientes:

- Necesitamos codificar la clase **Action** para que opere correctamente en un entorno *multi-thread*, como si se estuviera codificando un método `service`() de un *servlet*.
- El principio más importante que ayuda en la codificación de *threads* seguros es usar sólo variables locales en las clases **Action**. Las variables locales se crean en una pila que es asignada (por el JVM) a cada *thread* solicitado, por eso no es necesario preocuparse por compartirlas.
- Los *beans* que representan el Modelo del sistema podrían lanzar excepciones debido a problemas de acceso a bases de datos o a otros recursos. Se debería atrapar dichas excepciones en la lógica del método `perform()`, y guardarlas en el archivo de logeo de la aplicación (junto con el seguimiento de pila correspondiente) llamando a: "servlet.log("Error message text", exception);"
- Como regla general, asignar recursos y mantenerlos a través de las solicitudes del mismo usuario (en la misma sesión de usuario) puede causar problemas de escalabilidad. Se debería pensar en liberar esos recursos (como las conexiones a una base de datos) antes de reenviar el control al componente de la Vista apropiado -- incluso si un método del *bean* que se ha llamado lanza una excepción.

4.2.3.3.2 Implementación de ActionMapping

Para poder operar satisfactoriamente, el *servlet* controlador Struts necesita conocer varias cosas sobre como se debería mapear toda URI solicitada a una clase **Action** apropiada. El conocimiento requerido ha sido encapsulado en un interfaz Java, llamado **ActionMapping**, estas son las propiedades más importantes [McClanahan, 2002]:

- *type* - nombre totalmente calificado de la clase Java que implementa la clase **Action** usada por este mapeo.
- *name* - El nombre del *bean* de formulario definido en el archivo de configuración que usará este *action*.
- *path* - El *path* de la URI solicitada que corresponden con la selección de este mapeo.
- *unknown* - Seleccionado a *true* si este *action* debería ser configurado como por defecto para esta aplicación, para manejar todas las solicitudes no manejadas por otros *action*. Sólo un **Action** puede estar definido como por defecto dentro de una sola aplicación.
- *validate* - Seleccionado a *true* si se debería llamar al método validate() de la *action* asociada con este mapeo.
- *forward* - El *path* de la URI solicitada a la que se pasa el control cuando se ha invocado su mapeo. Esto es una alternativa a declarar una propiedad *type*.

En este capítulo se analizaron las tecnologías más importantes para el desarrollo de esta tesis, tanto para la manipulación, lectura y creación de XML como para la implementación del patrón de diseño Modelo-Vista-Controlador (Ver Apéndice B), por medio del marco de trabajo Struts, un desarrollo del proyecto Jakarta de la compañía Apache que está revolucionando el desarrollo de Web Services. El uso de dichas tecnologías servirá de soporte para construir el sistema propuesto. Todas las tecnologías mencionadas en este capítulo así como el marco teórico y los proyectos relacionados, mencionados en los capítulos anteriores permiten situarnos en el contexto de este proyecto de tesis, para así dar paso al capítulo siguiente, el cual presenta el análisis y diseño del proyecto GISonline.