Desarrollo de Aplicaciones

Web con Java aplicando el patrón de diseño MVC Sin Utilizar un Framework

Francisco Jacob Ávila Camacho*
Adolfo Meléndez Ramírez*
Valentín Roldán Vázquez**

Resumen

En este trabajo presentamos la forma en que se puede implementar el patrón de diseño MVC (Modelo Vista Controlador) de una manera sencilla, sin necesidad de incorporar un framework como struts o spring, lo cual se vuelve conveniente cuando se requiere desarrollar una aplicación en poco tiempo y no se tienen los conocimientos de un framework que implemente MVC.

El trabajo describe las características del patrón de diseño MVC y la conveniencia de utilizarlo aún en aplicaciones sencillas para generar una solución basada en estándares y en las mejores prácticas del desarrollo de software. El artículo no pretende ser un tutorial, sino más bien una justificación y una guía respecto a la conveniencia de utilizar MVC en el desarrollo de aplicaciones Web. La arquitectura del programa utilizado como ejemplo, pretende servir de base para una aplicación web más elaborada o compleja.
Keywords: Aplicaciones Web, Java, Mejores prácticas, Modelo Vista Controlador MVC, Patrón de Diseño, JSP, Servlets.

Keywords: Aplicaciones Web, Java, Mejores prácticas, Modelo Vista Controlador MVC, Patrón de Diseño, JSP, Servlets.

Acerca de los autores...
*
Profesor en la División de Ingeniería en Sistemas Computacionales, Tecnológico de Estudios Superiores de Ecatepec
** Profesor en el Departamento de Matemáticas, UNAM FES Cuautitlán

Introducción

La plataforma Java Enterprise Edition (JEE) fue concebida para construir presencia en Internet al permitir que los desarrolladores puedan utilizar Java para crear aplicaciones multicapas del lado del servidor [1].
Hoy día las APIs de Java Enterprise Edition se han extendido para abarcar numerosas áreas, como RMI o CORBA para el manejo de objetos remotos, JDBC para la interacción con bases de datos, JNDI para el acceso a servicios de directorio y de nombrado, EJB para la creación de componentes de negocio reutilizables, JMS para la capa orientada a mensajes, JAXP para el procesamiento XML y JTA para la ejecución de transacciones atómicas, entre otras tecnologías de JEE [3]. Adicionalmente JEE también soporta servlets, el substituto en Java para CGI scripts. La combinación de estas tecnologías permite a los programadores crear soluciones distribuidas para una gran variedad de aplicaciones de negocios [1].

En 1999 Sun Microsystems agregó un nuevo elemento a la colección de herramientas Enterprise de Java: JavaServer Pages (JSP). Los JavaServer Pages se construyen sobre la capa de Java Servlets y se diseñaron para incrementar la eficiencia con la cual tanto los programadores como los no programadores pueden crear contenido Web sin necesidad de incrustar HTML dentro del código Java [2].

Mientras que los servlets pueden ser utilizados para extender la funcionalidad de cualquier servidor Java, hoy día son más utilizados para extender la funcionalidad de los servidores Web. Cuando se utiliza un servlet para crear contenido dinámico para una página electrónica, se está creando una aplicación Web. Una aplicación Web ofrece una interactividad mayor que la que ofrece una página Web estática. Una aplicación Web puede ser tan simple como un buscador de palabras clave en un documento, o tan compleja como una tienda en línea de comercio electrónico. Dichas aplicaciones se construyen sobre Internet o en intranets o extranets corporativas, con el potencial de incrementar la productividad y cambiar la forma en que las organizaciones de todo tipo realizan actividades de negocios [2].

Los servlets también son la base de múltiples componentes implementados por los entornos de trabajo o frameworks, como los Action en struts o los Controllers en Spring [4] y de otros componentes de JEE como se puede apreciar en la siguiente figura.

La Figura 1 muestra la relación entre las tecnologías Java desarrolladas a partir de los componentes base: Java Servlets y JavaServer Pages. Los servlets son clases del lenguaje de programación Java que procesan peticiones de forma dinámica y construyen respuestas [3]. Las páginas JSP son documentos de texto que se ejecutan como servlets, pero permiten una aproximación más natural a la utilizada para crear contenido estático [3].

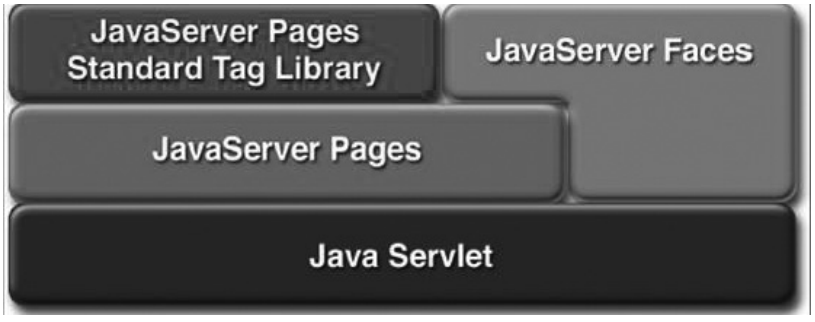


Figura 1.Relación de las tecnologías Java basadas en servlets. Fuente: Java EE 5 Tutorial [3]

La plataforma JEE utiliza un modelo de aplicación multicapa distribuida para aplicaciones empresariales. La lógica de la aplicación se divide en componentes de acuerdo con su función. Estos componentes pueden encontrarse instalados en diferentes equipos dependiendo de la capa a la que pertenecen.

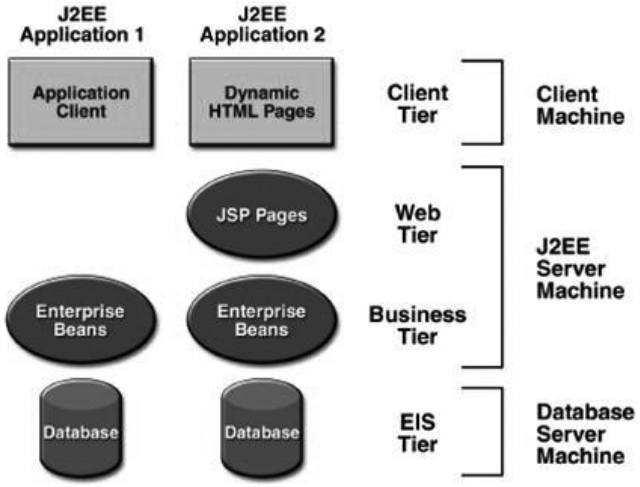


Figura 2.Estructura multicapa para dos aplicaciones web. Fuente: Java EE 5 Tutorial [3]

La Figura 2 muestra dos aplicaciones multicapa JEE divididas en capas, las cuales se describen como:

* Capa Cliente. Componentes corriendo en la máquina del cliente
* Capa Web. Componentes corriendo en el servidor JEE
* Capa de Negocios: Componentes corriendo en el servidor JEE
* Capa de sistemas de información (EIS). Software corriendo en el servidor EIS

Aunque una aplicación JEE puede consistir de tres o cuatro capas, como se muestra en la Figura 2, las aplicaciones multicapa JEE son generalmente consideradas de tres capas, ya que se distribuyen en tres partes: máquina cliente, servidor JEE, y servidores de bases de datos o sistemas empresariales de información.

De esta forma, se extiende el modelo estándar cliente-servidor de dos capas, colocando un servidor de aplicaciones multicapa entre el cliente y el sistema de almacenamiento o bases de datos [4].
La amplia adopción de las tecnologías estratégicas de JEE ha generado estándares abiertos para construir arquitecturas basadas en servicios para las aplicaciones empresariales. Al mismo tiempo, la curva de aprendizaje en las tecnologías JEE es a menudo confundida con el aprendizaje del diseño con tecnologías JEE; muchos de los libros enfocados hacia aspectos específicos de la tecnología, no siempre son claros en cómo aplicarla [8].

Los arquitectos de aplicaciones JEE necesitan entender más allá de los conceptos de las API como:

* ¿Cuáles son las mejores prácticas?
* ¿Cuáles son las malas prácticas?
* ¿Cuáles son los problemas más comunes y las soluciones probadas a estos problemas?
* ¿Cómo se puede reconstruir el código desde un escenario menos óptimo, o desde una mala práctica, hacia una mejor forma, normalmente descrita por un patrón de diseño?

Los buenos diseños se generan con la práctica y la experiencia. Cuando éstos son comunicados como patrones utilizando una plantilla estándar, se convierten en un mecanismo poderoso para intercambiar comunicados y obtener beneficios de la reutilización de componentes, al mismo tiempo se convierten en una influencia para mejorar la forma en que se diseña y se construye software [8].

Los patrones representan soluciones expertas a problemas recurrentes en un contexto y de ahí se capturan en varios niveles de abstracción y en diversos dominios [11]. Existen diversas categorías para clasificar los patrones de software y las más comunes son:

* Patrones de diseño
* Patrones arquitectónicos
* Patrones de análisis
* Patrones de creación
* Patrones estructurales
* Patrones de comportamiento

Aún con esta breve lista de categorías, se aprecian varios niveles de abstracción y esquemas de clasificación ortogonal [8].

I Modelo Vista Controlador - MVC

El MVC (por sus siglas en inglés) es un patrón de diseño de arquitectura de software usado principalmente en aplicaciones que manejan gran cantidad de datos y transacciones complejas, y en donde se requiere de una mejor separación de conceptos para estructurar el desarrollo de una mejor manera y facilitar la programación en diferentes capas de forma independiente [7]. MVC sugiere la separación en tres capas:

Modelo: Consisten en la representación de la información que maneja la aplicación. El modelo representa los datos puros que, dentro del contexto del sistema, proveen de información al usuario o a la aplicación misma, donde también se encarga de las reglas de negocio [6].

Vista: Consisten en la representación del modelo en formato gráfico, disponible para la interacción con el usuario. En el caso de una aplicación Web, la vista es una página HTML con contenido dinámico sobre el cual el usuario puede realizar operaciones [6].

Controlador: Es la capa encargada de atender las peticiones del usuario, procesando la información y modificando el modelo cuando así se requiere [6].

El ciclo de vida de MVC se representa por la interacción de las tres capas descritas anteriormente con el cliente o usuario.

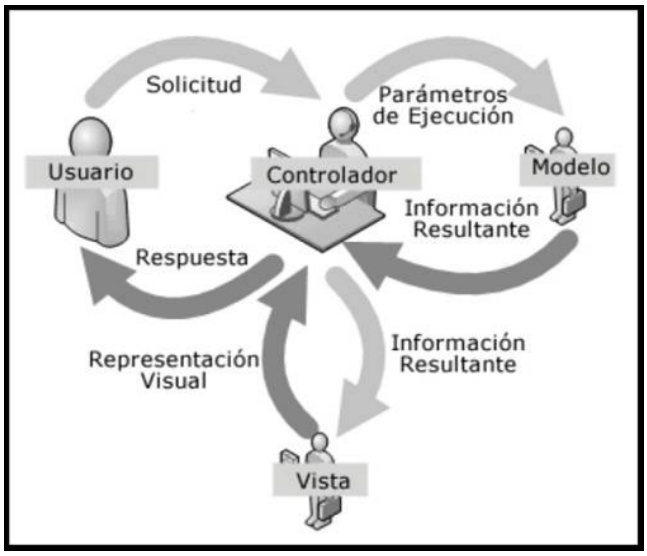


Figura 2.Ciclo de Vida MVC. Fuente: [6]

El ciclo inicia cuando el usuario hace una petición al controlador con información sobre lo que el usuario desea realizar. Entonces el controlador decide a quién debe delegar la tarea y es aquí donde el modelo comienza su trabajo. El modelo se encarga de realizar operaciones sobre la información que maneja y así cumplir con lo solicitado por el controlador. Luego de que termina su trabajo, le regresa al controlador la información resultante, la cual a su vez se re-direcciona a la vista. La vista se encarga de transformar los datos en información entendible para el usuario. Esta información formateada se envía de regreso al controlador, quién se encarga de transmitirla al usuario. El ciclo inicia nuevamente cuando el usuario realiza una nueva petición.

1.1 Ventajas y desventajas de MVC

Las principales ventajas del patrón MVC son:

* La separación del modelo de la vista, con ello separa los datos de la representación visual de los mismos.
* Es más sencillo incorporar múltiples representaciones de los mismos datos.
* Facilidad para agregar nuevos tipos de datos que se requiera por la aplicación, ya que son independientes del funcionamiento de las otras capas.
* Independencia de funcionamiento.
* Facilidad para el manejo de errores.
* Opciones sencillas para probar el funcionamiento del sistema.
* Facilidad para el escalamiento de la aplicación en caso de ser requerido.

Las desventajas del patrón MVC son:

* La separación en capas incorpora complejidad al sistema.
* La cantidad de archivos a manejar y desarrollar se incrementa considerablemente.
* La curva de aprendizaje del patrón es más alta que usando otros modelos más simples.

La comparación de las ventajas y desventajas de MVC puede ser muy subjetiva, e incluso ser parte de un tema a debate, sin embargo, para el objetivo de este trabajo, la balanza se inclina a favor de MVC.

Actualmente existen diversos frameworks que implementan MVC en el desarrollo de aplicaciones, pero su uso requiere de una curva de aprendizaje mucho más amplia y completa, que incluye el funcionamiento de dicho framework, por lo que, para el desarrollo de este trabajo se decidió seguir el esquema que plantea MVC sin hacer uso de frameworks externos.

2 Implementación del patrón MVC

Con la finalidad de implementar el patrón MVC sin ningún framwork externo, se utilizará la tecnología Java Servlets, a fin de implementar las funciones del controlador y la tecnología JSP para las funciones de la capa de vista. Para la capa del modelo y las reglas de negocio se utilizan clases que ejecutan las funcionalidades del ejemplo; dichas clases en algún momento podrían interactuar con la base de datos, ya sea aplicando un patrón como DAO o utilizando simplemente JDBC; en el ejemplo no mostramos esta parte y nos concretamos en explicar el funcionamiento del modelo.

Para el ejemplo de implementación del patrón MVC se utilizará la siguiente arquitectura, la cual se muestra en la Figura 4, donde se aprecian los componentes que forman parte de esta aplicación de ejemplo.

Tomando en cuenta que la arquitectura de la aplicación debe permitir el desarrollo y mantenimiento independiente de sus capas (Modelo, Vista y Controlador), centrando la atención en el flujo de operación de la capa de negocio (Modelo) y su interrelación con la capa de presentación (Vista), la arquitectura queda de la siguiente manera:

En la Figura 4 observamos que el punto de entrada del modelo es un servlet que llamamos SPrincipal. Este componente recibe las peticiones iniciales que llegan al servidor Web e interpreta el request para obtener los parámetros asociados a cada petición.

Con esta información el servlet pasa el control a la clase AccionProxy, donde, a través del archivo de configuración AccionConf obtiene el nombre de la clase que modela la acción para la solicitud recibida.

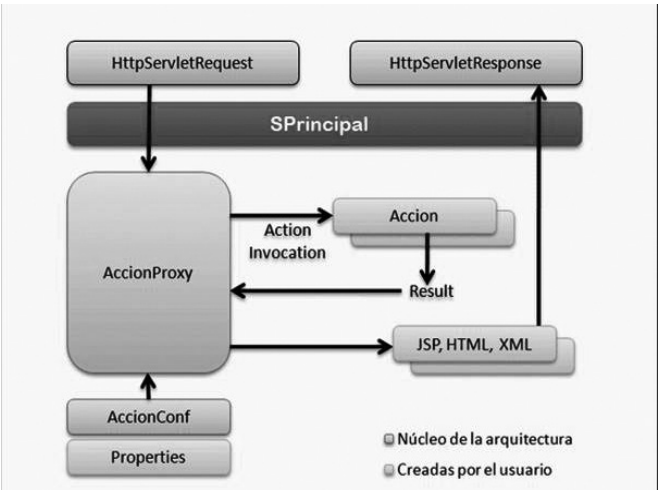


Figura 4. Arquitectura para la implementación de la aplicación Web de ejemplo MVC.

AccionProxy instancia la clase concreta y ejecuta el método que encapsula la funcionalidad. Una vez ejecutada la acción, evaluará el resultado y en función de ello, determinará si la petición debe ser redireccionada a otra acción o se debe generar una vista que será enviada de regreso al usuario.

Cabe mencionar que las variantes a esta arquitectura pueden ser muchas, sin embargo ésta es la que nos permitirá explicar la estrategia de implementación del patrón MVC, la cual puede servir de base para la creación de otro tipo de aplicaciones.

La Vista del modelo MVC está compuesta por la página Web a través de la cual el usuario realizará la petición de la acción, así como la página que resultará de la ejecución de la petición.

"solicita_libro_cuento.html" es una página HTML que solicita la ejecución de la acción listarLibrosDeCuentoDisponibles al pulsar el botón del formulario y cuyo código se muestra a continuación.

```
<form method="post" action="SPrincipal">

<input type="hidden" name="pAccion" value="lis
tarLibrosDeCuentoDisponibles" />

 <input type="submit" value="Solicitar libros de
cuento disponibles" />

</form>
```

En el formulario podemos apreciar que la petición se realiza a un servlet llamado "SPrincipal", el cual recibe un parámetro llamado "pAccion" con el valor "listarLibrosDeCuentoDisponibles".
El resultado de la ejecución de esta acción será visualizado por la página: resultado.jsp, la cual se describe más adelante.

El Control. El servlet “SPrincipal” será el encargado de recibir la petición realizada desde la página “solicita_libro_cuento.html”. En el servlet, el método doPost tratará la petición recibida desde el Web con el objeto request, como se muestra en el siguiente código

```
<protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String parametro, accion = null;
parametro = request.getParameter(“pAccion”);

if(parametro != null && parametro.length() > 0){

accion = parametro;
AccionProxy accProxy = AccionProxy.getInstance();
accProxy.creaAction(request, response, accion);

}
}
```

Observamos en el código cómo se recupera el parámetro “pAccion” desde el objeto request para obtener su valor: “listarLibrosDeCuentoDisponibles”. Si el parámetro es válido, obtenemos la instancia existente de la clase ActionProxy a la que pasamos el control invocando al método “creaAction” con los parámetros de la conexión y el valor de la acción solicitada.

Como se puede apreciar, “SPrincipal” se limita a ser el punto de entrada unificado para todas las posibles peticiones pasando el control al ActionProxy, desacoplando de esta forma la capa de Vista con la capa de Control.

Una vez transferido el control al ActionProxy, este objeto se encarga de crear una instancia en tiempo de ejecución de la clase que modela la acción que se solicita desde la vista y que representa las reglas de negocio.

Para saber qué clase se debe instanciar, el ActionProxy utiliza un archivo de propiedades con los valores correspondientes a cada acción. El archivo se llama: “acciones.properties”, el cual es un archivo de propiedades con la estructura clave=valor donde se declara, para cada acción definida en la aplicación, la clase Java que modela la funcionalidad asociada a cada petición. También se declara que se debe hacer cuando termine la ejecución de la funcionalidad.

Las variables que utilizamos dentro del archivo de propiedades para el ActionProxy tienen las siguientes características:

• "nombre de la acción".srcAction.Ruta completa de la clase que modela la acción y encapsula el código a ejecutar.

• "nombre de la acción".true.resultType. Describe, en caso de que la acción se ejecute satisfactoriamente, si a continuación se debe ejecutar otra acción o re-direccionar a la vista. Los posibles valores son: action para ejecutar una acción, html y jsp para vistas.

• "nombre de la acción".true.resultValue. Describe, en caso de que la acción se ejecute satisfactoriamente, el nombre de la nueva acción a ejecutar o del contenido Web a generar.

• "nombre de la acción".false.resultType. Describe, en caso que la acción no sea satisfactoria o produzca un resultado inesperado, si a continuación se debe ejecutar otra acción o re-direccionar a la vista. Los posibles valores son: action para ejecutar una acción, html y jsp para vistas.

• "nombre de la acción".false.resultValue. Describe, en caso que la acción no se satisfactoria o produzca un resultado inesperado, el nombre de la nueva acción a ejecutar o del contenido Web a generar.

El archivo de propiedades para el ejemplo es el siguiente:

```
listarLibrosDeCuentoDisponibles.srcAction = com.
ro.ejercicioMVC.acciones.ListarLibrosCuento
listarLibrosDeCuentoDisponibles.true.resultType =
action
listarLibrosDeCuentoDisponibles.true.resultValue =
listarCualquierLibroDisponible
listarLibrosDeCuentoDisponibles.false.resultType =
html
listarLibrosDeCuentoDisponibles.false.resultValue =
respuestaInesperada.html
listarCualquierLibroDisponible.srcAction = com.
ro.ejercicioMVC.acciones.Listar0trosLibros
listarCualquierLibroDisponible.true.resultType = jsp
listarCualquierLibroDisponible.true.resultValue =
respuesta.jsp
listarCualquierLibroDisponible.false.resultType =
html
listarCualquierLibroDisponible.false.resultValue =
respuestaInesperada.html
```

En el archivo se observa que la clase "ListarLibrosCuent" que se encuentra en el paquete "com.ro.ejercicioMVC.acciones." es la clase que modela la funcionalidad solicitada por la acción llamada "listarLibrosDeCuentoDisponibles" y que una vez que el objeto ejecuta esta funcionalidad correctamente, se invoca a una nueva acción encadenada, la cual se llama: "listarCualquierLibroDisponible".

Por su parte, se declara que la acción “listarCualquierLibroDisponible” está modelada por la clase “ListarOtrosLibros” también en el paquete: “com.ro.ejercicioMVC.acciones” y que al finalizar satisfactoriamente su ejecución se re-direccione a la página: “respuesta.jsp” donde se mostrarán los resultados de ambas acciones.

Si la ejecución de alguna de estas acciones diera un resultado inesperado, se re-direcciona al usuario a la página: “respuestaInesperada.html”.

Controlando estas propiedades, se puede cambiar el flujo de ejecución de la aplicación de forma sencilla, así como agregar nuevas funcionalidades e integrarlas con las existentes de forma clara y sin tener que recompilar la aplicación.

Como se observa en el código del servlet “SPrincipal”, se crea un objeto de la clase ActionProxy y se invoca al método “creaAction”, al cual se le envían los parámetros de la conexión y la acción solicitada desde la Web. El siguiente código muestra los detalles del método “creaAction”.

```
public void creaAction (HttpServletRequest request, HttpServletResponse
response, String actionName){
try{
Boolean otraAcc = false;
String nombAcc = actionName;
String tipoRespAccExe,valorRespAccExe = null;
AccionConf confAcc = new AccionConf();
confAcc.setPropertiesPath(“com.ro.ejercicioMVC.acciones”);
do {
String nuevaAcc = confAcc.getProperty(nombAcc + “.srcAction”);
if ( nuevaAcc != null ){ nuevaAcc = nuevaAcc.trim(); }

Accion accion;
accion = (Accion) Class.forName(nuevaAcc).newInstance();
accion.setActionParams(request, response);
Boolean RespAccExe = accion.executeAction();

if (RespAccExe.equals(true))
{
...
} else if (RespAccExe.equals(false)) {
...
}
} while( otraAcc.equals(true) ) ;

} catch (Exception ex) {System.out.println(“ Exception: “ + ex.getMessage() );
}

}
```

Una vez que se recupera el archivo de propiedades, se obtiene la propiedad relacionada con la acción solicitada. Para ello, la clase ActionProxy se apoya del método “getProperties(String clave)” de la clase ActionConf, al cual se le pasa la clave para obtener el valor asociado.

La ruta de la clase que modela la acción almacenada en la variable “nuevaAcc”, se crea con el objeto de la clase de acción correspondiente en tiempo de ejecución utilizando el método “forName(nuevaAcc)” para cargar la clase y newInstance() para crear el objeto.

Una vez creado el objeto y con el uso del polimorfismo a través del método “executeAction” de la super clase “Accion” se ejecuta el método “execute()” de la clase correspondiente de acción. Es en este método donde realmente se implementa el código específico que se debe ejecutar para la funcionalidad correspondiente.

Una vez que la acción se ha realizado, la respuesta de la ejecución se almacena en la variable “RespAccExe” y en función de si la acción se ha ejecutado correctamente o no, se recuperan los parámetros correspondientes (resultType y resultValue) asociados al true o al false y se crea un nuevo ciclo, en caso de que el resultado sea una acción o se sale del ciclo principal despachando a la página que corresponda.

El bloque del código con la evaluación de RespAccExe es el siguiente:

```
if (RespAccExe.equals(true))
{
tipoRespAccExe =confAcc.
getProperty(nombAcc + ".true.
resultType");
if (tipoRespAccExe != null )
{ tipoRespAccExe = tipoRespAccExe.
trim(); }

valorRespAccExe = confAcc.
getProperty(nombAcc+".true.
resultValue");
if ( valorRespAccExe != null ) {
valorRespAccExe = valorRespAccExe.
trim(); }

if (tipoRespAccExe.equals("action"))
{

otraAcc = true;
nombAcc = valorRespAccExe;

} else {
try {
otraAcc = false;
request.getRequestDispatcher("/"+
valorRespAccExe).forward(request,
response);
break;
} catch (Exception e) {System.out.
println(" error en forward: " +
e.getMessage()); }
}

} else if (RespAccExe.equals(false))
{

tipoRespAccExe =confAcc.
getProperty(nombAcc + ".false.
resultType");
if (tipoRespAccExe != null )
{ tipoRespAccExe = tipoRespAccExe.
trim(); }

valorRespAccExe = confAcc.
getProperty(nombAcc+".false.
resultValue");
if ( valorRespAccExe != null ) {
valorRespAccExe = valorRespAccExe.
trim(); }

if (tipoRespAccExe.equals("action"))
{
 otraAcc = true;
nombAcc = valorRespAccExe;

} else {
try {
otraAcc = false;
request.getRequestDispatcher("/"+
valorRespAccExe).forward(request,
response);
break;
} catch (Exception e) {System.out.
println(" error en forward: " +
e.getMessage()); }
}

}
```

Al finalizar su ejecución la instancia ListarLibrosCuento y ser el resultado de la misma positivo/verdadero (en RespAccExe), se evalúan los parámetros de configuración:

- listarLibrosDeCuentoDisponibles.true.resultType y
- listarLibrosDeCuentoDisponibles.true.resultValue,

que en este caso indican que el resultado de esta acción es otra acción de nombre “listarCualquierLibroDisponible”.

Así que la variable local nombAcc se carga con el nuevo valor (“listarCualquierLibroDisponible”) y se asigna a la variable lógica local otraAcc el valor “verdadero”, lo que implicará que ciclo do-while se reinicie, esta vez para la nueva acción.

Una vez ejecutada la nueva acción, cuando lleguemos al mismo punto donde se evalúan resultType y resultValue encontraremos que la configuración indica que el resultado de esta acción es una página jsp con el nombre “respuesta.jsp”, con lo que a otraAcc se le da el valor “false”, se hace un forward de la vista asociada y se interrumpe el ciclo.
La clase abstracta “Accion” tiene básicamente tres métodos:

- setActionParams: A través del cual las clases que heredan de Accion podrán tener acceso a la request/response.
- executeAction: Que es le método que será invocado por AccionProxy sobre la clase de acción concreta haciendo uso del polimorfismo.
- execute: Que es el método abstracto que tendrán que implementar todas las acciones “hijas” de esta acción genérica.

El código correspondiente queda de la siguiente manera:

```
public void setActionParams (HttpServletRequest request, HttpServletResponse
response ){ this.request = request; this.response = response; }
public abstract void execute() throws Exception;
public boolean executeAction() {
boolean resp = true;
try{
execute();
} catch (Throwable ex)
{
System.out.println(" Exception: " + ex.toString() + " en " + getClass() );
resp = false;
}
return resp;
}
```

Las clases que implementan las acciones concretas ListarLibrosCuento y ListarOtrosLibros implementarán el método execute y en nuestro ejemplo simplemente van a enviar un mensaje de salida a la Web: un texto indicando el resultado de la búsqueda de los libros.

El objetivo de este trabajo está centrado exclusivamente en el flujo de navegación a través de las capas del modelo MVC.

El código de esta clase es:

```
package com.ro.ejercicioMVC.acciones;

public class ListarLibrosCuento
extends Accion {
public void execute () {
String informe = " No hay libros de
cuento disponibles, no se quedarán de
otros géneros";
request.setAttribute("LibrosEncontrad
os", informe);
}
}
public class ListarOtrosLibros extends
Accion {
public void execute () {
String informe = " No hay libros
disponibles de ningún tipo, todos
están prestados";
request.setAttribute("LibrosEncontrad
os", informe);
}
}
```

Para la implementación del ejemplo, se creó un Dynamic Web Project utilizando Eclipse como entorno de desarrollo. La estructura del proyecto es la siguiente:

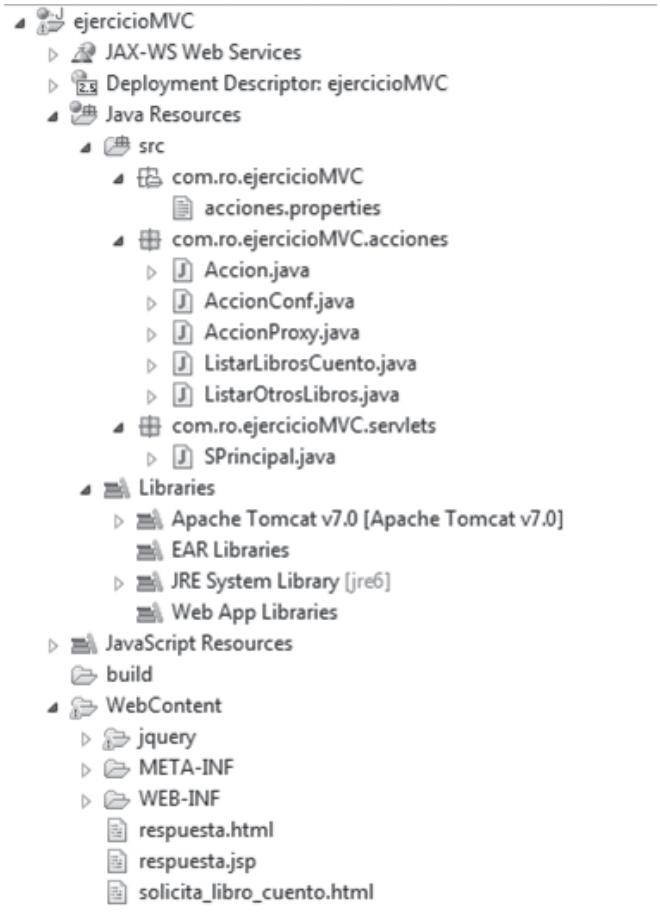
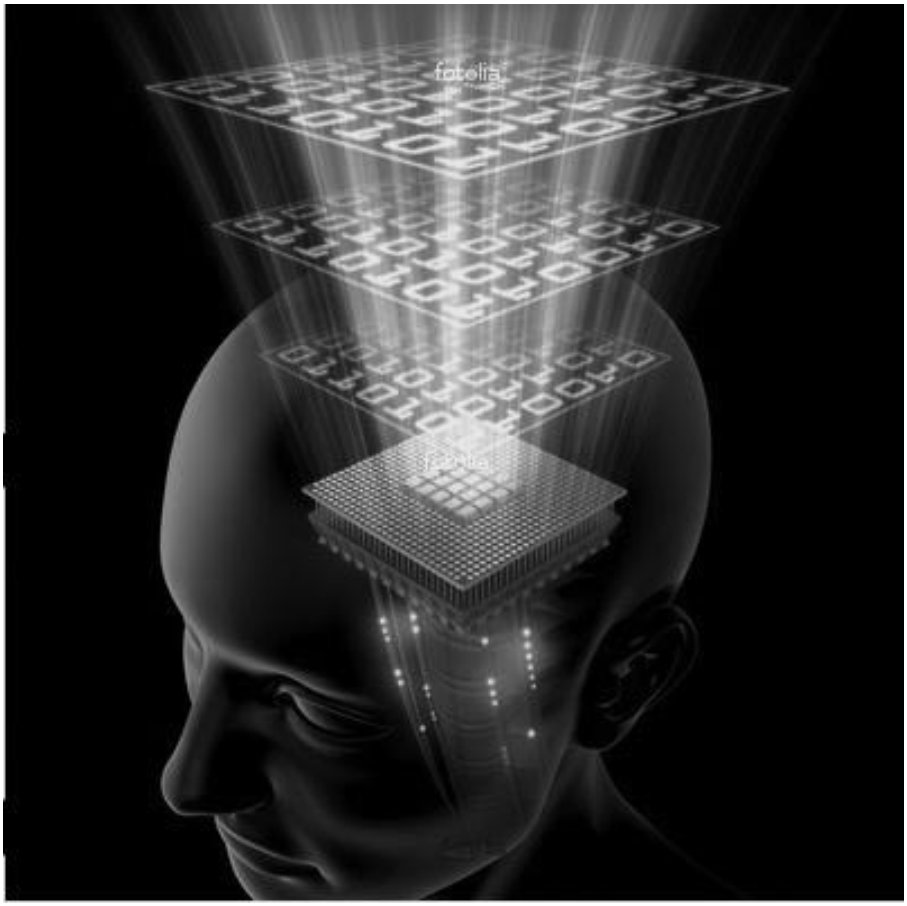


Figura 5. Estructura del ejemplo implementado utilizando MVC.



Conclusiones

En el presente trabajo hemos explicado las ventajas de utilizar un patrón de diseño en el desarrollo de aplicaciones web, como el patrón MVC, sin la necesidad de conocer un framework externo y con la finalidad de utilizar las mejores prácticas para el desarrollo de software, aun con aplicaciones basadas en Servlets y JSP. El ejemplo descrito, nos deja ver unas estrategias de implementación del patrón MVC bajo un esquema sencillo, utilizando las ventajas de la programación orientada a objetos y la estrategia de desarrollo en capas totalmente independientes y desacopladas.

La arquitectura propuesta nos sirve de guía para el desarrollo de múltiples aplicaciones Web, las cuales se podrán elaborar aprovechando las ventajas de MVC y de la estrategia de desarrollo en capas independientes.

El ejemplo presentado muestra una opción rápida para la aplicación de los principios y la implementación de ciertos patrones que, sin duda, ofrecen una base sólida para el diseño y la arquitectura de aplicaciones confiables creadas en forma rápida y con bajo riesgo.

Referencias

1. Bergsten, H.: Java Server Pages, 2nd Edition. O'Relly, Sebastopol (2002).
2. Hunter, J., Crawford, W.: Java Servlet Programming. O'Relly, Sebastopol (1998).
3. Bodof, S. et al.: Java EE 5 Tutorial. Addison-Wesley, Palo Alto (2010).
4. Green, D. et al.: Java EE 6 Tutorial. Addison-Wesley, Palo Alto (2011).
5. Modelo Vista Controlador, http://es.wikipedia.org/wiki/Modelo_Vista_Controlador (Recuperado el 12 de marzo de 2012).
6. Pavón, J.: Estructura de las Aplicaciones Orientadas a Objetos: El Patrón Modelo-Vista-Controlador (MVC). In: Dep. Ingeniería de Software e Inteligencia Artificial, Universidad Complutense Madrid, Madrid (2008). http://www.fdi.ucm.es/profesor/jpavon/poo/2.14.MVC.pdf (Recuperado el 7 de marzo de 2012).
7. Oracle, Model-View-Controller Also known as MVC, http://www.oracle.com/technetwork/java/mvc-140477.html (Recuperado el 8 de marzo de 2012).
8. Aulur, D., Crupi, J., Malks, D.: J2EE Patterns: Best Practice and Design Strategies. Sun Microsystems, Cupertino (2005).
9. Stelting, S., Masen, O.: Patrones de Diseño Aplicados a Java. Pearson Education, Santa Clara (2010).
10. Diaz, M.: Ingeniería de la Web y Patrones de Diseño. Pearson Education, Santa Clara (2011).
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (2000).
12. Ladd, S., Davison, D., Devijiver, S., Yates, C.: Expert Spring MVC and Web Flow. Apres, New York (2006).