A continuación se utilizan esos métodos para visualizar los mensajes de error:

```
try
   {
      //Código
   }
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e)
{
      System.out.println("HA OCURRIDO UNA EXCEPCIÓN:");
      System.out.println("Mensaje:     "+ e.getMessage());
      System.out.println("SQL estado: "+ e.getSQLState());
      System.out.println("Cód error:  "+ e.getErrorCode());    }
```

El siguiente ejemplo muestra la salida que se produce cuando se intenta hacer SELECT de una tabla que no existe (en MySQL):

```
HA OCURRIDO UNA EXCEPCIÓN:
Mensaje:    Table 'ejemplo.departamentoss' doesn't exist
SQL estado: 42S02
Cód error:  1146
```

En Oracle se visualizaría información diferente:

```
HA OCURRIDO UNA EXCEPCIÓN:
Mensaje:    ORA-00942: la tabla o vista no existe
SQL estado: 42000
Cód error:  942
```

2.12 PATRÓN MODELO-VISTA-CONTROLADOR. ACCESO A DATOS

El patrón MVC (*Model-View-Controller*) es un patrón de diseño que se utiliza como guía para el diseño de arquitecturas software que ofrecen una fuerte interactividad con el usuario y donde se requiere una separación de conceptos para que el desarrollo se realice más eficazmente facilitando la programación en diferentes capas de manera paralela e independiente. Este patrón organiza la aplicación en 3 bloques cada cual especializado en una tarea:

- **El Modelo**: representa los datos de la aplicación y sus reglas de negocio.
- **La Vista:** es la representación del modelo de forma gráfica para interactuar con el usuario, un ejemplo son los formularios de entrada y salida de información o páginas HTML con contenido dinámico.
- **El Controlador:** interpreta los datos que recibe del usuario analizando la petición, coordinando la vista y el modelo para que la aplicación produzca los resultados esperados.

Las ventajas de hacer uso de este patrón son:

- Separación de los datos de la representación visual de los mismos.
- Diseño de aplicaciones modulares.

- Reutilización de código.
- Facilidad para probar las unidades por separado.
- Facilita el mantenimiento y la detección de errores.

Entre las desventajas cabe destacar la complejidad que se agrega al sistema al separar los conceptos en capas o la cantidad de ficheros a desarrollar que se incrementa considerablemente.

La Figura 2.17 describe el flujo general de la solicitud de un usuario construida en una arquitectura MVC, como se puede observar, el controlador es el que dirige la aplicación. Los pasos son los siguientes:

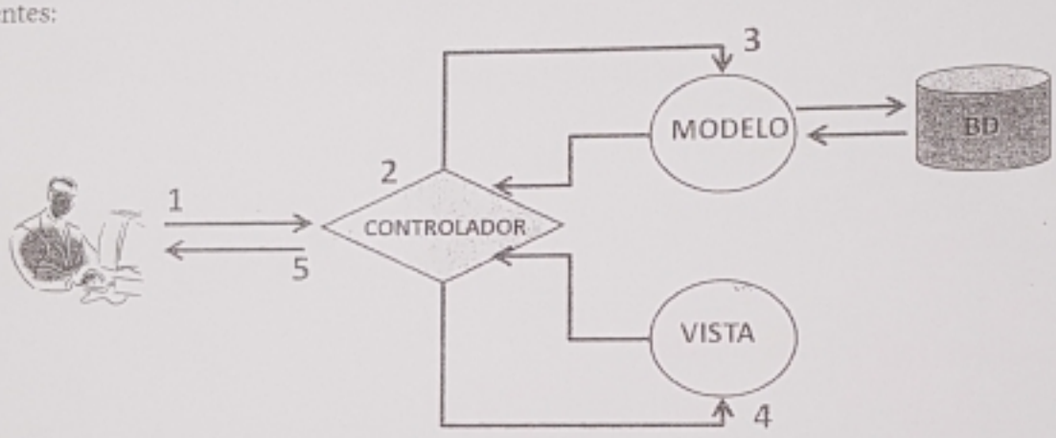

Figura 2.17. Flujo de solicitud en el MVC

1. Un usuario realiza una solicitud a través de la aplicación (pulsa un botón, un enlace,). La solicitud es dirigida al controlador.
2. El controlador examina la solicitud y decide qué regla de negocio aplicar, es decir, determinará el componente de negocio a aplicar para procesar la solicitud, este componente de negocio es el modelo.
3. El modelo contiene las reglas de negocio que procesan la solicitud y que dan lugar al acceso a los datos que necesita el usuario. Estos datos se devuelven al controlador.
4. El controlador toma los datos que devuelve el modelo y selecciona la vista en la que se van a presentar esos datos al usuario.
5. El controlador devuelve los resultados al usuario tras procesar la solicitud.

El patrón MVC es utilizado en múltiples frameworks: Java Enterprise Edition (J2EE), Apache Struts (para aplicaciones web J2EE), Ruby on Rails (para aplicaciones web con Ruby), Google Web Toolkit (GWT, para crear aplicaciones Ajax con Java), ASP.NET MVC Framework (Microsoft), etc. La mayoría de los frameworks para web implementan este patrón. Una aplicación de este patrón en aplicaciones web J2EE es lo que se conoce con el nombre de **Modelo 2**. Esta arquitectura se basa en los siguientes elementos:

- La utilización de Servlets para capturar las peticiones realizadas por el cliente web y redirigirlas a las páginas JSP adecuadas. Estos actúan como controlador.

- Páginas JSP utilizadas para mostrar la interfaz de usuario, implementan la vista. Estas páginas usan los JavaBeans como componente modelo.
- JavaBeans o POJOs (*Plain Old Java Object*) que implementan el modelo.

A continuación se muestra un ejemplo de una aplicación web MVC Java. La arquitectura **Modelo 2** de nuestra aplicación se muestra en la Figura 2.18. Para poder trabajar con ella necesitaremos instalar un contenedor web con soporte para Servlets y JSPs, en este ejemplo se realizará la aplicación sobre Tomcat.

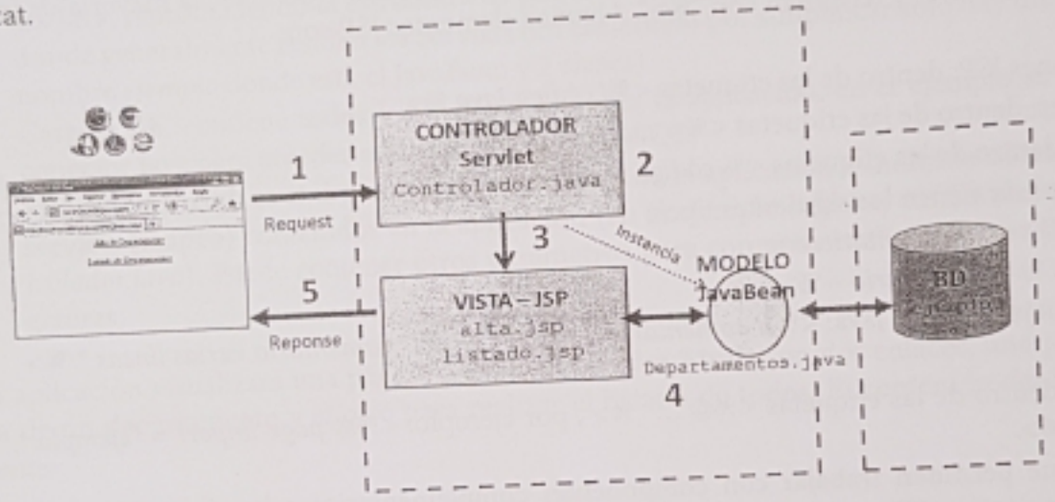

Figura 2.18. Arquitectura Modelo 2.

Se puede descargar la última versión desde la web http://tomcat.apache.org/, para el ejemplo se ha descargado la versión *apache-tomcat-7.0.23.zip* (*apache-tomcat-7.0.23.tar.gz* en Linux). La instalación es sencilla, solo hay que descomprimir el fichero que contiene una única carpeta de nombre *apache-tomcat-7.0.23*. Para iniciar el servidor se ejecuta el fichero *startup.bat* en sistemas Windows o *startup.sh* en Linux que se encuentran en la carpeta *\apache-tomcat-7.0.23\bin*.

Antes de empezar con la aplicación introducimos una serie de conceptos sobre los Servlets y las páginas JSP.Los Servlets son clases Java que no tienen el método *main()*, en su lugar se invocan otros métodos cuando se reciben las peticiones. Los métodos son: *init()*, *service()* y *destroy()*. El ciclo de vida de un Servlet se divide en varios pasos:

- El cliente solicita una petición a un servidor a través de una URL. El servidor recibe la petición.
- Si es la primera vez, el servidor carga el Servlet y se llama al método *init()* para iniciarlo.
- Se llama al método *service()* para procesar las peticiones de los clientes web.
- Se llama al método *destroy()* para eliminar al Servlet y liberar los recursos. El servidor los destruye porque cesan las llamadas desde el cliente, un temporizador del servidor así lo indica o el propio administrador lo decide.

Las páginas JSP nos permiten generar contenido dinámico en la web, son páginas web con etiquetas especiales y código Java incrustado. La diferencia con el Servlet es que en este el código es Java puro que recibe peticiones y genera una página web a partir de ellas. Una página JSP consta de 2 partes:

- HTML o XML para el contenido estático.
- Etiquetas JSP y scriplets escritos en lenguaje de programación Java para encapsular la lógica que genera el contenido dinámico.

El código de las partes dinámicas se encierra entre unas etiquetas especiales, la mayoría de las cuales empiezan con "<%" y terminan con "%>". Algunos elementos JSP son:

- Declaraciones JSP: dentro de las etiquetas *<%! código Java %>*
- Expresiones: dentro de las etiquetas *<%= código Java %>*
- Scriptlets: dentro de las etiquetas *<% código Java %>*
- Comentarios: se tienen los siguientes tipos:
  - De HTML: *<!-- comentario -->*
  - De JSP: *<%-- comentario --%>*
  - Del lenguaje de script Java: *<%// comentario línea %>* y *<%/* comentario varias líneas */%>*

- Directivas: dentro de las etiquetas <%@ ... %> , por ejemplo: *<%@ page import = "ejemplo.*, java.util.*" %>*.
- Acciones: que permiten trabajar con componentes complementarios a la página JSP como applets, otras páginas JSP, javabeans, etc. Son las siguientes:
  - No asociadas a los javabeans:
    <jsp:include> ... </jsp:include> o *<jsp:include ... />* si solo tiene atributos
    <jsp:plugin> ... </jsp:plugin>
    <jsp:forward> ... </jsp:forward> o *<jsp:forward ... />* si solo tiene atributos
  - Asociadas a los javabeans:
    <jsp:useBean ... /> si solo tiene atributos o *<jsp:useBean ...> ... </jsp:useBean>*
    <jsp:setProperty ... />
    <jsp:getProperty ... />

Nuestra aplicación web estará formada por una serie de ficheros JSP, HTML, Java, XML y librerías AR, la estructura de directorios y ficheros es la siguiente:

```
ejemploMVC
  alta.jsp
  index.html
  listado.jsp
  WEB-INF
    web.xml
    classes
      Controlador.class
      Controlador.java
      OperacionesBD.class
      OperacionesBD.java
      ejemplo
        Departamentos.class
        Departamentos.java
    lib
      mysql-connector-java-5.1.18-bin.jar
```

Esta estructura estará dentro de la carpeta *webapps* (*D:\apache-tomcat-7.0.23\webapps\ejemploMVC*). Hay algunos ficheros y carpetas con un significado especial:

- Carpeta **WEB-INF**: es imprescindible y su nombre debe aparecer siempre en mayúsculas. Contiene las siguientes carpetas y ficheros:
  - Carpeta **classes**: es necesaria si usamos ficheros *.class*. Contiene los paquetes de nuestras clases Java, reproduciendo la estructura de paquetes y subpaquetes. Es dentro de este directorio donde generalmente residen los Servlets (en el ejemplo por simplificar solo hay un paquete de nombre *ejemplo* donde está el JavaBean y 2 clases).
  - Carpeta **lib**: contiene todos los JAR que necesita la aplicación, en el ejemplo se necesita el conector Java para acceder a la base de datos MySQL.
  - Fichero **web.xml**: es el descriptor de despliegue de la aplicación e indica la ubicación de los servlets (mapeo) contenidos en la aplicación (en el ejemplo solo hay un Servlet que es *Controlador.java*). Puede contener otros parámetros para componentes adicionales y manejo de errores.

La aplicación visualizará una página web inicial (**index.html**) con dos enlaces, uno para realizar el alta de un departamento y el otro para realizar el listado de todos. El contenido del fichero es el siguiente:

```
<html>
<body>
  <p align='center'>
  <a href="/ejemploMVC/controlador?accion=alta">Alta de Departamento</a></p>
  <p align='center'>
  <a href="/ejemploMVC/controlador?accion=listado">Listado de Departamentos</a>
</p>
</body>
</html>
```

Desde los enlaces *href* se llama al controlador con la acción a realizar alta o listado. Para poder llevar a cabo esta acción los contenedores web tienen un fichero llamado **web.xml** que se encarga de mapear la URL. El contenedor web (Tomcat) lee el documento **web.xml** y realiza el mapeo entre el alias encontrado en el path de la URL y el Servlet en cuestión. El contenido del fichero es:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0"
  metadata-complete="true">
 <servlet>
    <servlet-name>Controlador</servlet-name>
    <servlet-class>Controlador</servlet-class>
 </servlet>
 <servlet-mapping>
    <servlet-name>Controlador</servlet-name>
    <url-pattern>/controlador</url-pattern>
 </servlet-mapping>
```

```
<session-config>
   <session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
   <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Nos fijamos en los elementos *<servlet>* y *<servlet-mapping>*:

El elemento *<servlet>* define las características de un Servlet. Está compuesto por los elementos *<servlet-name>* donde se indica el nombre del Servlet (en el ejemplo *Controlador*) y *<servlet-class>* que indica el nombre de la clase Java que contiene el Servlet (*nombredirectorio.ficheroclase*, en el ejemplo el fichero *Controlador.class* no está dentro de ningún directorio (o paquete), por ello se escribe en este elemento solo *Controlador*).

El elemento *<servlet-mapping>* define la ubicación en términos de directorios de un sitio (URL). El Servlet de nombre *Controlador* (*<servlet-name>Controlador</servlet-name>*) será accedido cada vez que se acceda a la URL */controlador* (*<url-pattern>/controlador</url-pattern>*). *<url-pattern>* indica la forma en que se debe invocar al Servlet, en el documento HTML poníamos lo siguiente para invocar al Servlet: *<a href="/ejemploMVC/**controlador?accion=alta**">*, además se le envía el parámetro *accion*.

Controlador.java: es el Servlet controlador que maneja todas las solicitudes entrantes. Recibe un parámetro de nombre *accion*. Dependiendo del valor de este parámetro podrá realizar varias acciones: redirigir la petición a *alta.jsp*, redirigir la respuesta a *listado.jsp* enviando los departamentos a listar e invocar al método para la inserción de un departamento (cuyos datos se reciben de *alta.jsp*) en la base de datos y después redirigir la respuesta a *index.html*. Antes de realizar el listado carga los datos de la tabla DEPARTAMENTOS en un array, para ello utilizan el método *listarDep()* de la clase OperacionesBD. Los datos para insertar un departamento los recibe de la página *alta.jsp* mediante el atributo *depart*, la inserción la realiza invocando al método *insertaDepartamento()* de la clase OperacionesBD:

```
import ejemplo.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Controlador extends HttpServlet {
public void service(HttpServletRequest request, HttpServletResponse response)
                  throws ServletException, IOException {

   //parámetro acción, se obtiene de la URL de index.html, puede ser 'alta' o
'listado'
   String op=request.getParameter("accion");

   //si se ha pulsado en alta de departamento se visualiza la pantalla de alta
   if(op.equals("alta")) response.sendRedirect("alta.jsp");

   //si se ha pulsado en listado, primero se cargan los datos de departamentos en
```

```
//una lista y luego se envían a listado.jsp
if(op.equals("listado")){
    OperacionesBD operBD=new OperacionesBD();
    ArrayList lista=operBD.listarDep(); //se cargan los datos de los dep
    request.setAttribute("departamentos",lista); //se preparan para enviar al jsp
    RequestDispatcher rd=request.getRequestDispatcher("/listado.jsp");
    rd.forward(request,response);
  }

 //se inserta departamento en la tabla y luego se visualiza index.html
 if(op.equals("insertar")){
     ejemplo.Departamentos dep=
            (ejemplo.Departamentos)request.getAttribute("depart");//obtener deps
     OperacionesBD operBD=new OperacionesBD();
     operBD.insertaDepartamento(dep); //se insertan en tabla departamentos
     response.sendRedirect("index.html"); //se muestra la página inicial
    }
}
}//fin de la clase Controlador
```

En el ejemplo solo se ha utilizado el método **service()** del Servlet. Este acepta 2 argumentos:

- *HttpServletRequest request*: en este argumento se recibe la petición del cliente.
- *HttpServletResponse response*: es la respuesta que da el servidor al cliente.

Algunos métodos usados en la petición (objeto *HttpServletRequest*) son:

- *request.getParameter("nombre_de_parametro")*: lee el parámetro enviado por el cliente, devuelve el valor del parámetro especificado o null si no existe.
- *request.setAttribute("nombre", objeto)*: se almacena un objeto en la sesión con el nombre indicado.
- *request.getAttribute("nombre")*: devuelve el atributo asociado al nombre especificado en esta sesión o null si no hay ningún objeto asociado bajo el nombre.

Algunos métodos usados en la respuesta al cliente (objeto *HttpServletResponse*) son:

- *response.sendRedirect("url")*: la respuesta al cliente es la url indicada, puede ser un fichero JSP, un Servlet o una página HTML.

La interfaz *RequestDispatcher* encapsula una referencia a otro recurso web. El método *getRequestDispatcher("url")* acepta una ruta de URL que haga referencia al recurso objetivo, la ruta debe ser absoluta, es decir, el nombre debe empezar por /. El método *forward(request,response)* permite reenviar la solicitud a otro servlet, página JSP o fichero HTML.

alta.jsp: Visualiza un formulario para realizar la entrada de datos de un departamento. Los datos del formulario se enviarán a través del JavaBean (el modelo) *Departamentos.java* al controlador; el nombre del parámetro que contendrá los datos del departamento a insertar es *depart* (*id="depart"*) y se usará en el controlador con el método *getAttribute()* (*request.getAttribute("depart")*) :

```
<html>
<head>
<title>ALTA DE DEPARTAMENTOS</title>
</head>
<!--Form de entrada de datos e Inserción en el JavaBean-clase Departamentos-->
<jsp:useBean id="depart" scope="request" class="ejemplo.Departamentos" />
<jsp:setProperty name="depart" property="*"/>
<%
 if(request.getParameter("deptno")!= null) {%>
<jsp:forward page="/controlador?accion=insertar"/>
<%}%>

<body>
  <center><h2>ENTRADA DE DATOS DE DEPARTAMENTOS</h2>
  </br>
  <form method="post">
        <p>Número de departamento: <input name="deptno" type="text" size="5"></p>
        <p>Nombre: <input name="dnombre" type="text" size="15"> </p>
        <p>Localidad: <input name="loc" type="text" size="15"> </p>
        <input type="submit" name="insertar" value="Insertar departamento.">
        <input type="reset" name="cancelar" value="Cancelar entrada.">
  </form>
  </center>
</body>
</html>
```

Un JavaBean es una clase Java que nos permite ocultar su implementación mostrando al exterior solo los métodos y propiedades públicos. Para acceder al Bean desde una página JSP se utilizan una serie de etiquetas, algunas son: *<jsp:useBean>*, *<jsp:setProperty>* ,*<jsp:getProperty>*, *<jsp:forward>*. En el ejemplo se han usado:

- *<jsp:useBean id="depart" scope="request" class="ejemplo.Departamentos" />*: *id* indica el nombre del Bean, *scope* es el alcance del Bean, un alcance "request" implica que el bean es accesible hasta otra JSP que haya sido invocada por medio de *jsp:forward* o *jsp:include*. El atributo *class* es el nombre de la clase.
- *<jsp:setProperty name="depart" property="*"/>*: el atributo *name* debe ser igual al especificado en *id*, con *property="*"* indicamos que se van a extraer todos los valores de la solicitud (*deptno, dnombre* y *loc*).
- *<jsp:forward page="/controlador?accion=insertar"/>*: permite que la solicitud sea enviada a otra página JSP, a un Servlet o a un recurso estático. En este caso se envia al controlador con el parámetro *accion* con valor *insertar* para que inserte los datos del Bean en la tabla.

Departamentos.java: es el JavaBean que facilita el intercambio de datos entre el controlador y el modelo y posteriormente entre el controlador y la vista:

```
package ejemplo;
    public class Departamentos {
          private byte deptno;
          private String dnombre;
          private String loc;
```

```
public Departamentos() {}
public Departamentos(byte deptno, final String dnombre, final String loc) {
        this.deptno = deptno;
        this.dnombre = dnombre;
        this.loc = loc;
          }
 public byte getDeptno() {return this.deptno;}
 public void setDeptno(byte deptno) {this.deptno = deptno;}
 public String getDnombre() {return this.dnombre;}
 public void setDnombre(String dnombre) {this.dnombre = dnombre;}
 public String getLoc() {return this.loc;}
 public void setLoc(String loc) {this.loc = loc;    }
  }//fin clase Departamentos
```

listado.jsp: Obtiene el listado de los departamentos, es llamado por el controlador y recibe el atributo *departamentos* conteniendo un array con los departamentos a listar:

```
<%@ page import="ejemplo.*,java.util.*"%>
<html><head><title>LISTADO DE DEPARTAMENTOS</title></head>
<body>
<center>
<h2>LISTADO DE DEPARTAMENTOS</h2>
<table border='1'>
<tr><th>Departamento</th><th>Nombre</th><th>Localidad</th></tr>
<%
ArrayList listadep=(ArrayList)request.getAttribute("departamentos");
if(listadep!=null)
  for(int i=0;i<listadep.size();i++){
   Departamentos d=(Departamentos)listadep.get(i);%>
   <tr><td><%=d.getDeptno()%></td>
   <td><%=d.getDnombre()%></td>
   <td><%=d.getLoc()%></td>
   </tr>
<%}%>
</table><br/><br/>
<a href="index.html">Inicio</a>
</center>
</body>
</html>
```

OperacionesBD.java: es la clase que realiza las operaciones contra la base de datos, la utiliza el controlador:

```
import ejemplo.*;
import java.sql.*;
import java.util.*;

public class OperacionesBD {
    //OBTENER LA CONEXIÓN
    public Connection getConnection(){
       Connection conexion=null;
       try{
           Class.forName("com.mysql.jdbc.Driver");//Cargar el driver
```

```
            conexion = DriverManager.getConnection
               ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
        }
        catch(Exception e){e.printStackTrace();}
        return conexion;
    }
    //LISTAR - Devuelve un array con la lista de departamentos
    public ArrayList listarDep(){
            ArrayList departamentos = new ArrayList ();
            try{
                  Connection conexion =getConnection();
                  Statement sentencia=conexion.createStatement();
                  String sql= "SELECT * FROM departamentos";
                  ResultSet resul=sentencia.executeQuery(sql);
                  while(resul.next()){ //se crea un array con los datos de los
                                       departamentos
                  ejemplo.Departamentos d=new ejemplo.Departamentos
                    resul.getByte("dept_no"),
                    resul.getString("dnombre"),resul.getString("loc"));

                    departamentos.add(d);//añadir dep al array
                  }
            conexion.close();
            }
            catch(Exception e){e.printStackTrace();}
            return(departamentos);
    }//fin listarDep

    //INSERTAR - Recibe los datos del departamento a insertar en la tabla
    public void insertaDepartamento(ejemplo.Departamentos d){
            try{
                  Connection conexion=getConnection();
                  Statement sentencia=conexion.createStatement();
                  String sql = "INSERT INTO departamentos VALUES('"
                        +d.getDeptno()+"','"+d.getDnombre()+"','"+d.getLoc()+"')";

                  if (d.getDeptno () != 0) sentencia.execute(sql);
                  System.out.println("SQL: "+sql);
                  conexion.close();
            }
            catch(Exception e){e.printStackTrace();}
    }//fin insertaDepartamento
}//fin clase OperacionesBD
```

Para compilar las clases seguimos estos pasos:

1.Nos dirigimos a la carpeta donde están las clases:

```
CD D:\apache-tomcat-7.0.23\webapps\ejemploMVC\WEB-INF\classes
```

2.Definimos la variable CLASSPATH:

```
SET CLASSPATH=D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/classes/
```

```
ejemplo/;D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/lib/mysql-connector-java
5.1.18-bin.jar;D:/apache-tomcat-7.0.23/lib/servlet-api.jar;D:/apache-tomcat-7.0.23/
webapps/ejemploMVC/WEB-INF/classes/;
```

3.Compilamos la clase Departamentos que está en la carpeta de nombre *ejemplo*:

```
javac ejemplo/Departamentos.java
```

4.Compilamos las otras dos clases:

```
javac -Xlint OperacionesBD.java

javac Controlador.java
```

Una vez compiladas será necesario recargar la aplicación desde Tomcat. Para ello abrimos el
gador y escribimos la URL: http://localhost:8080/. Puede ocurrir que el servidor no se inicie (a
cutar el fichero *startup.bat* o *startup.sh*) porque el puerto 8080 esté ocupado, en este caso cambi
el puerto 8080 por otro que no se esté utilizando, por ejemplo 8081. Este cambio se hace en el fi
D:\apache-tomcat-7.0.23\conf\server.xml (línea 70). También será necesario crear un usuario
manejar el entorno web de Tomcat. Añadimos al fichero *D:\apache-tomcat-7.0.23\conf\tomcat-
xml* las siguientes líneas entre las etiquetas <*tomcat-users*> </*tomcat-users*>:

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-gui"/>
```

Donde indicamos que el usuario *admin* con clave *admin* tendrá acceso a la gestión
aplicaciones web de Tomcat. Hecho esto, iniciamos de nuevo el servidor y escribimos la URL
localhost:8081/, a continuación pulsamos el botón *Manager App;* nos pedirá un nombre de u
y su clave, escribimos *admin*. Desde la pantalla de gestión de aplicaciones de Tomcat recar
nuestra aplicación */ejemploMVC*, véase Figura 2.19.

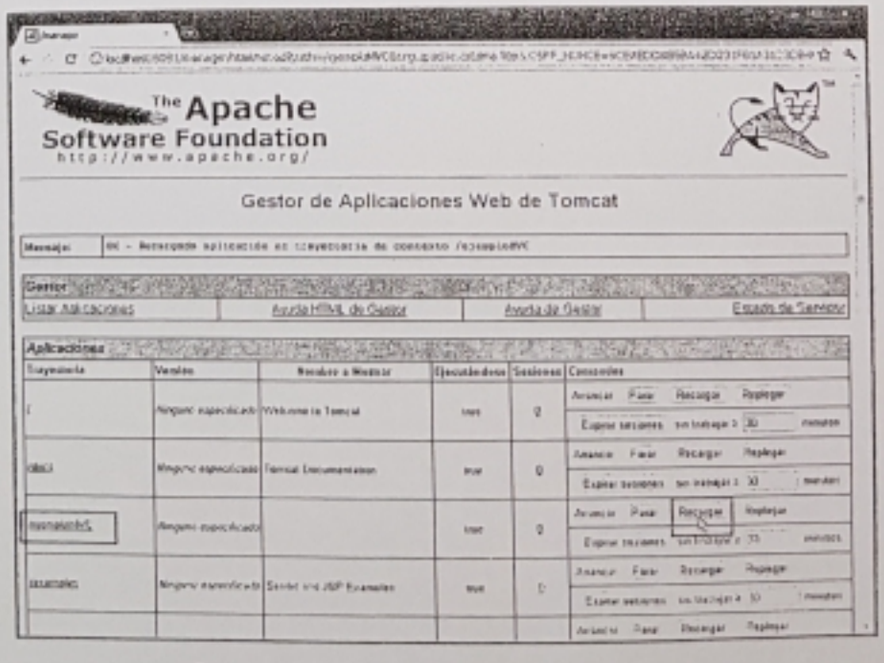

Figura 2.19. Recarga de la aplicación desde Tomcat.