Gestión de tareas de desarrollo de software

Fco. Javier Yáñez Juan

ITIS

Consultor: Javier Ferró Garcia

9 de enero de 2006

Índice de contenido

Plan de trabajo....3
Descripción del TFC....3
Objetivos generales y específicos....3
Planificación....3
Análisis de requisitos....5
Introducción....5
Glosario....6
Modelo de dominio....7
Casos de uso....8
Guiones....9
Descripción textual de los casos de uso....10
Fichas de los casos de uso....11
Futuras mejoras....13
Diseño....14
Arquitectura....15
Capas....15
Implementación de las capas....15
Arquitectura de patrones....17
Diagrama de despliegue....19
Diagrama de componentes....20
Diagrama de clases....21
Diagrama de colaboración....23
Persistencia....24
Detalles de implementación....26
ServiceLocator....26
Typesafe Enumeration....26
Interfaz de usuario....27
Conclusiones....28
Guía de instalación....29
Paso1....29
Paso 2....29
Paso 3....30
Paso 4....30
Paso 5....30

Plan de trabajo

Descripción del TFC

Este trabajo final de carrera consiste en la realización de una aplicación bajo la plataforma J2EE. La realización de esta aplicación permitirá aplicar conocimientos adquiridos en diferentes asignaturas de los estudios de ITIS. Con el desarrollo de la aplicación se adquirirán los conocimientos básicos sobre J2EE.

La aplicación a realizar permitirá gestionar las tareas pendientes en un proyecto de desarrollo de software. Es de utilidad sobre todo en las ultimas fases del proyecto cuando se quiere cerrar y hay múltiples cambios pequeños que hacer (corregir bugs, pequeñas mejoras, etc). También es útil para la fase de mantenimiento en la que los usuarios detectan bugs y piden pequeños cambios.

Objetivos generales y específicos

El desarrollo de la aplicación se basará en patrones de diseño.

La lógica de la aplicación se dividirá en varias partes: lógica de presentación, lógica de negocio y lógica de integración (persistencia en SGBD). Se pretende aplicar los patrones de diseño apropiados para cada tipo de lógica.

Se pretenden los siguientes objetivos funcionales:

1. Introducir tareas pendientes indicando el tipo de tarea (corregir bug, añadir funcionalidad, cambiar funcionalidad, etc), el titulo, la descripción, la prioridad y el proyecto.
2. Cambiar el estado de una tarea (en curso, realizada, rechazada, etc.)
3. Listar las tareas filtradas por estado, y ordenadas por prioridad.
4. Crear, modificar y eliminar proyectos.

Planificación

Fecha	***Hito***
16/10/05	Documento de análisis de requisitos
30/10/05	Documento de diseño
13/11/05	Programación: Crear, modificar y eliminar proyectos
20/11/05	Programación: Introducir tareas
27/11/05	Programación: Listar tareas
4/12/05	Programación: Cambiar el estado de una tarea
18/12/05	Memoria

Fecha	***Hito***
25/12/05	Presentación defensa del TFC

Análisis de requisitos

Introducción

En los proyectos de desarrollo de software se suelen gestionar los *bugs* mediante aplicaciones realizadas para tal fin, como, por ejemplo, *bugzilla.*

Se pretende realizar una aplicación que haga un poco más. Además de los *bugs*, durante el desarrollo de software se suelen acumular pequeñas tareas, tales como variaciones en la funcionalidad, variaciones en el interfaz, etc. Todas estas tareas junto con los *bugs* podrán ser gestionadas por la aplicación.

Glosario

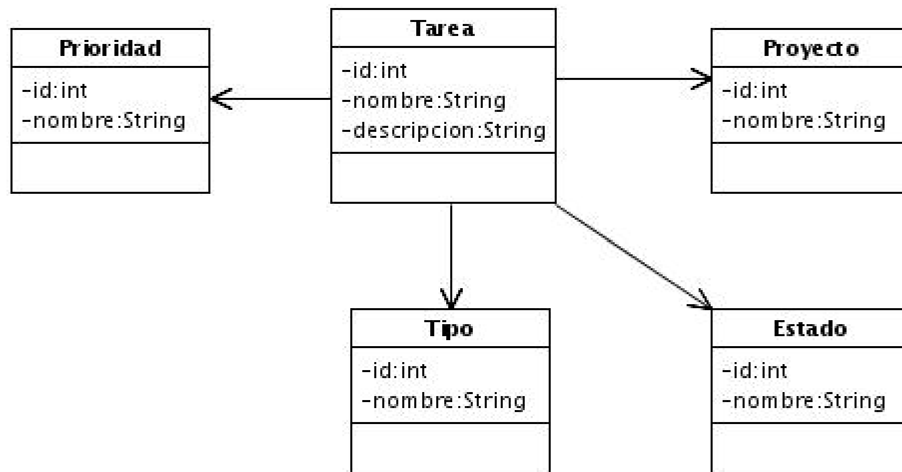
Proyecto: se refiere a un proyecto de desarrollo de software.

Tarea: trabajo pendiente de realizar en un proyecto.

Estado: situación en que se encuentra un tarea.

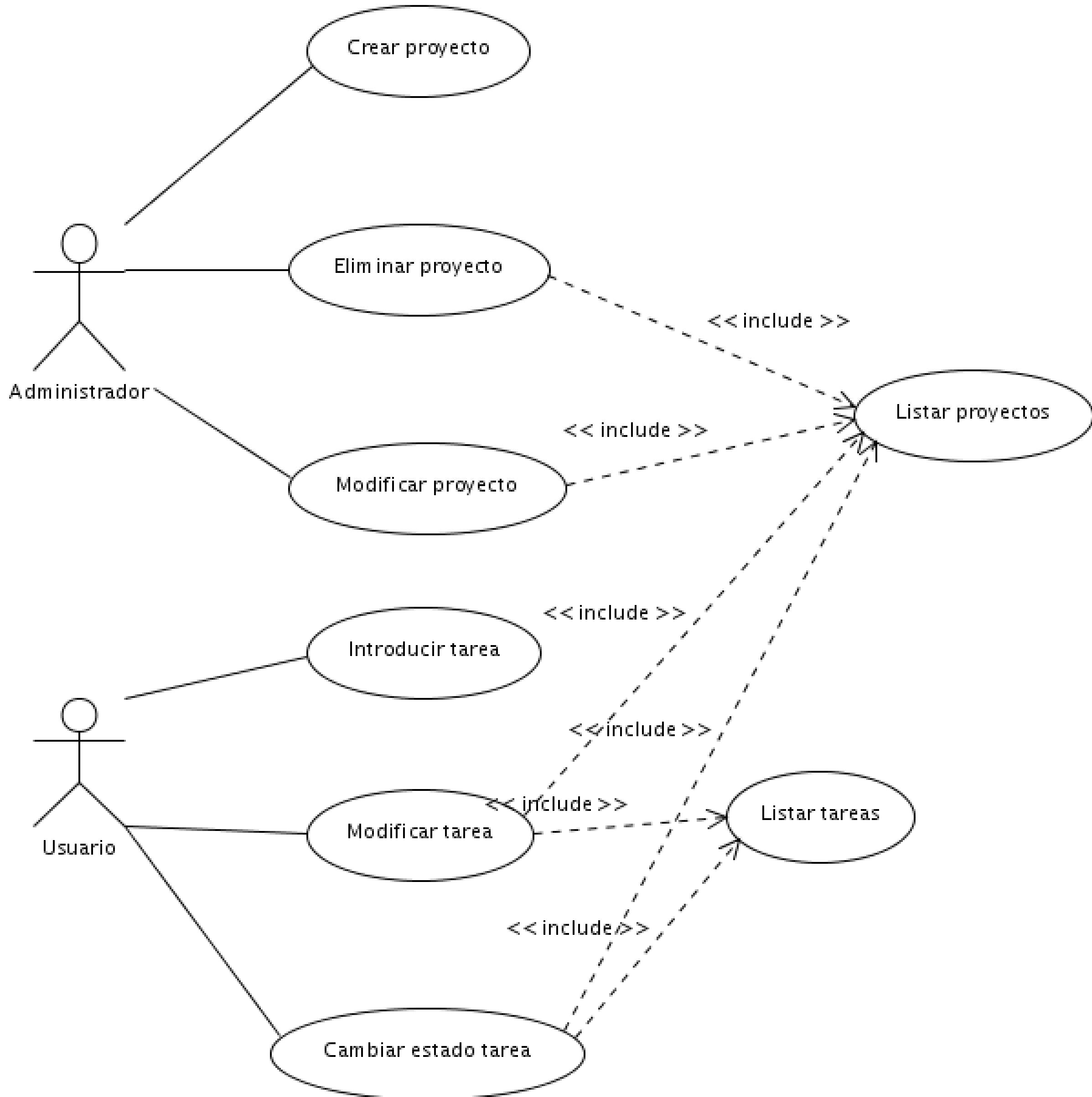
Tipo: naturaleza de una tarea. Los posibles tipos son los siguientes: bug, modificación, nueva funcionalidad.

Modelo de dominio



Para simplificar se han omitido los métodos de acceso a los atributos de las clases.

Casos de uso



Guiones

Administrador: el administrador es el responsable de crear, modificar y eliminar los proyectos a los que se les asignarán tareas.

Usuario: el usuario será quien asigne tareas a los proyectos. También se encargará de modificar el estado de dichas tareas.

Descripción textual de los casos de uso

CU-01 Crear proyecto: crea un nuevo proyecto en el sistema.

CU-02 Modificar proyecto: modifica las propiedades de un proyecto existente en el sistema.

CU-03 Eliminar proyecto: elimina un proyecto existente en el sistema. Si existen tareas relacionadas con el proyecto también son eliminadas.

CU-04 Listar proyectos: lista todos los proyectos existentes en el sistema.

CU-05 Introducir tarea: introduce una nueva tarea asociada a un determinado proyecto.

CU-06 Modificar tarea: modifica las propiedades de una tarea existente en el sistema.

CU-07 Cambiar estado tarea: cambia el estado de una tarea existente en el sistema.

CU-08 Listar tareas: lista las tareas pertenecientes a un determinado proyecto.

Fichas de los casos de uso

Caso de uso: crear proyecto	**Código:** CU-01
Descripción: crea un nuevo proyecto en el sistema	**Actores:** administrador **Prioridad:** alta
Pre-condición:	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** baja

Caso de uso: modificar proyecto	**Código:** CU-02
Descripción: modifica las propiedades de un proyecto existente en el sistema	**Actores:** administrador **Prioridad:** baja
Pre-condición: el proyecto tiene que haber sido creado	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** baja

Caso de uso: Eliminar proyecto	**Código:** CU-03
Descripción: elimina un proyecto existente en el sistema. Si existen tareas relacionadas con el proyecto también son eliminadas.	**Actores:** administrador **Prioridad:** baja
Pre-condición: el proyecto tiene que haber sido creado	**Post-condición:** las tareas relacionados con el proyecto no debe existir
Casos de uso relacionados:	**Frecuencia de uso:** baja

Caso de uso: Listar proyectos	**Código:** CU-04
Descripción: lista todos los proyectos existentes en el sistema	**Actores:** administrador, usuario **Prioridad:** alta
Pre-condición:	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** alta

Caso de uso: Introducir tarea	**Código:** CU-05
Descripción: introduce una nueva tarea asociada a un determinado proyecto	**Actores:** usuario

	Prioridad: media
Pre-condición: el proyecto relacionado tiene que existir	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** alta

Caso de uso: Modificar tarea	**Código:** CU-06
Descripción: modifica las propiedades de una tarea existente en el sistema	**Actores:** usuario
	Prioridad: baja
Pre-condición: la tarea tiene que existir	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** baja

Caso de uso: Cambiar estado tarea	**Código:** CU-07
Descripción: cambia el estado de una tarea existente en el sistema	**Actores:** usuario
	Prioridad: alta
Pre-condición: la tarea debe existir	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** alta

Caso de uso: Listar tareas	**Código:** CU-08
Descripción: lista las tareas pertenecientes a un determinado proyecto	**Actores:** usuario
	Prioridad: media
Pre-condición:	**Post-condición:**
Casos de uso relacionados:	**Frecuencia de uso:** alta

Futuras mejoras

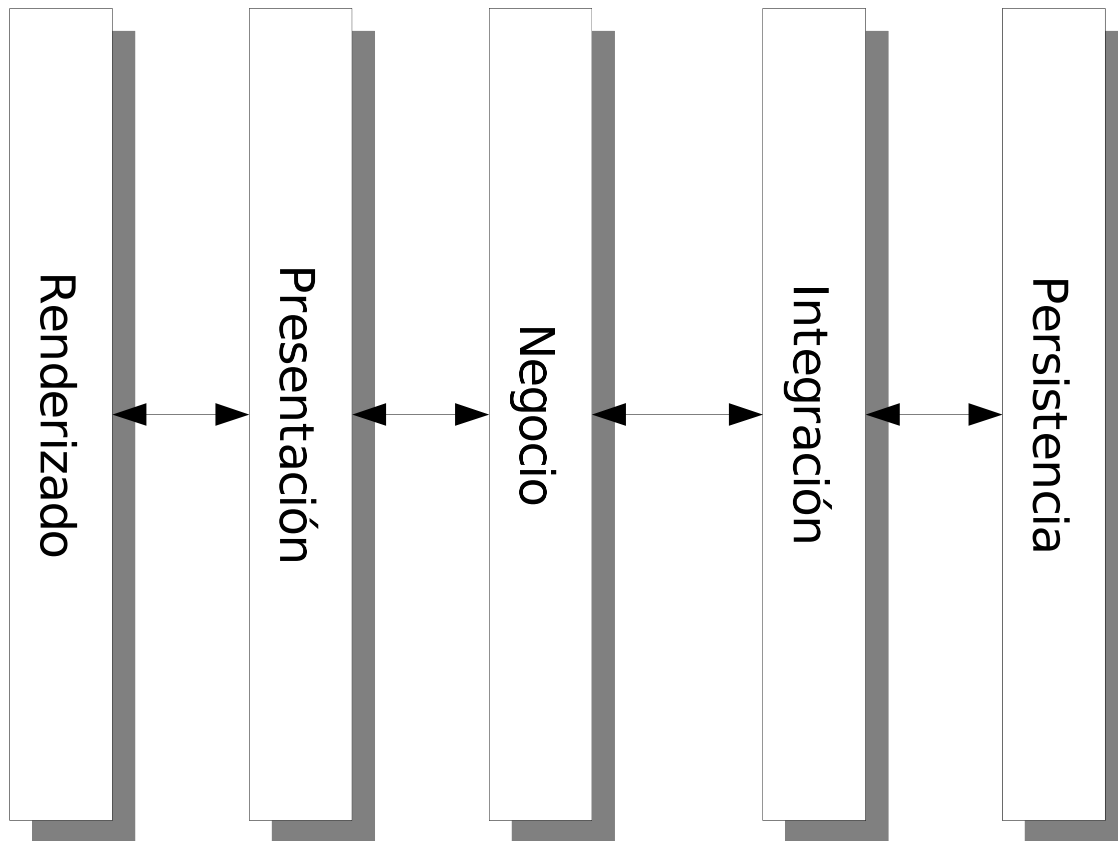
Debido a limitaciones de tiempo en este proyecto no se han podido incluir algunas funcionalidades.

Estas funcionalidades son las siguientes:

- **Seguridad:** creación de una gestión de usuarios basada en roles. Cada usuario deberá autenticarse para acceder al sistema y la funcionalidad que tendrá disponible variará según los roles que posea.
- **Asignación de tareas:** se podrá asignar una tarea a un usuario en concreto. Se podrá listar las tareas asignadas a un determinado usuario y filtrar por estado.
- **Avisos por e-mail:** cuando a un usuario se le asigne una tarea se le avisará por correo electrónico.

Diseño

Capas



En este esquema podemos ver las capas de las que va a constar el sistema.

La capa de renderizado se refiere al navegador (Internet Explorer, Firefox, Opera, etc.).

La capa de presentación es la que responde a las peticiones de la capa de renderizado después de consultar a la capa de negocio. Su respuesta es en forma de HTML.

La capa de negocio responde a la capa de presentación después de consultar a la capa de integración. En esta capa es donde se debe implementar las “reglas de negocio”.

La capa de integración permite a la capa de negocio acceder a la persistencia.

La capa de persistencia es la que permite persistir y recuperar la información procedente o demandada por las otras capas.

Implementación de las capas

La capa de renderizado no la vamos a implementar ya que existen múltiples navegadores que siguen los estándares y que serán validos para integrarse en el sistema.

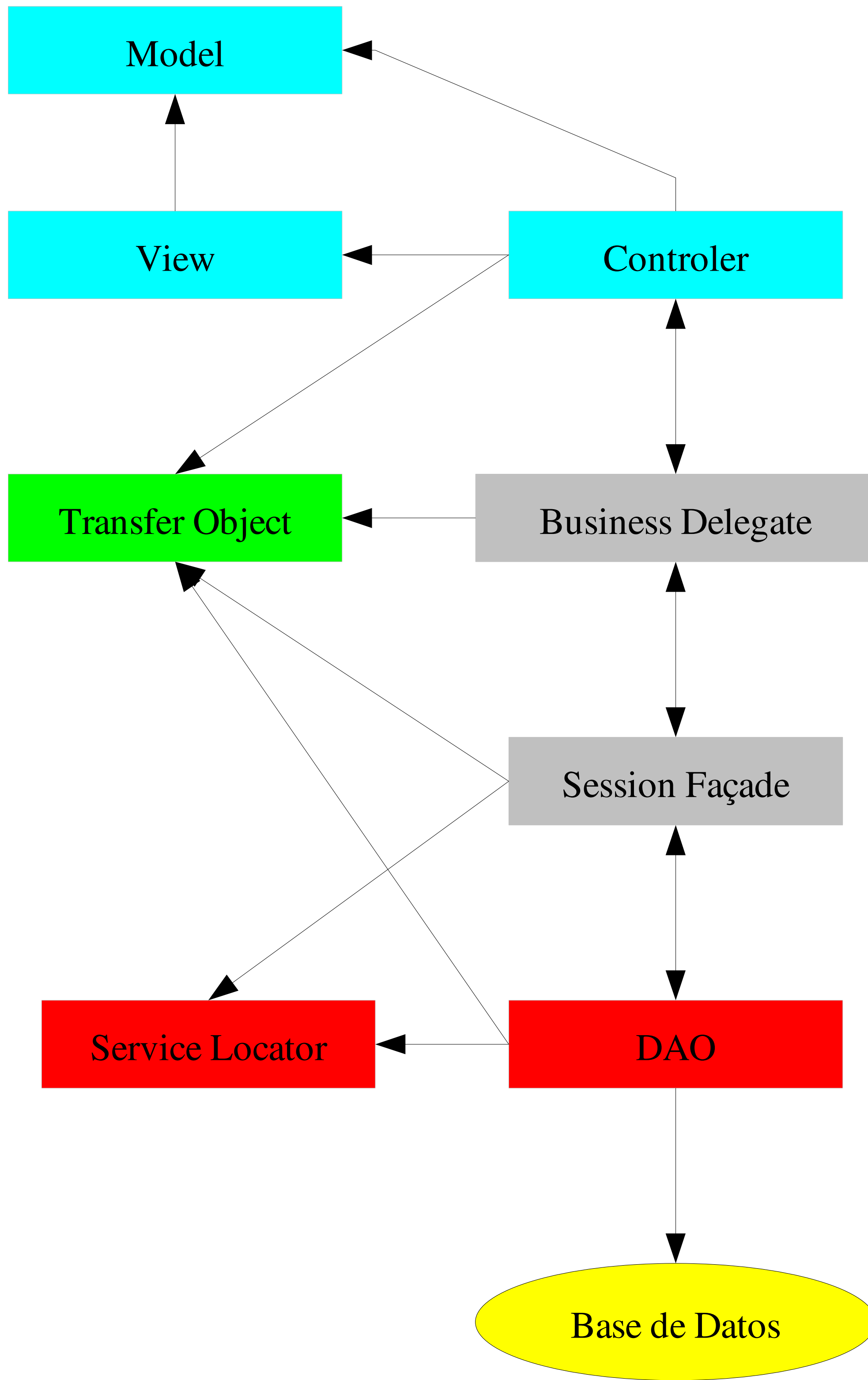
La capa de presentación la implementaremos usando el popular *framework* Struts.

La capa de negocio la implementaremos usando *Stateless Session Beans*.

La capa de integración la implementaremos usando el patrón DAO (ver más adelante la explicación).

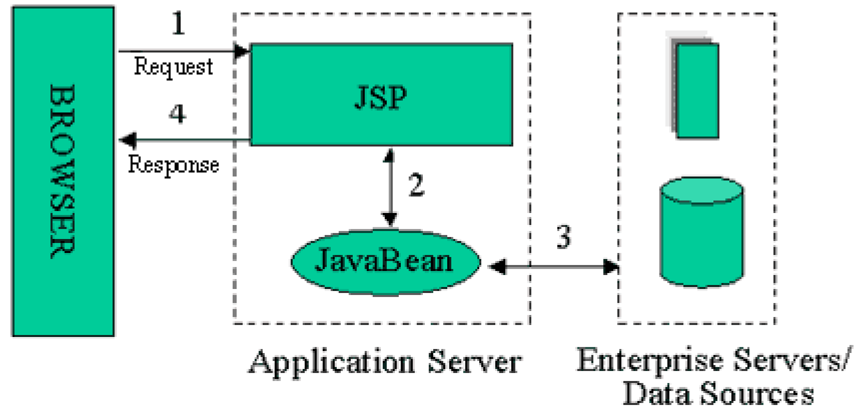
La capa de persistencia, al igual que la de renderizado, no la vamos a implementar ya que existen múltiples soluciones. En este caso confiaremos la persistencia al SGBD PostgreSQL.

Arquitectura de patrones



En color azul están los componentes del patrón MVC2., en gris los patrones de lógica de negocio, en rojo los de integración, en azul el patrón *Transfer Object* y en amarillo el SGBD que no es un patrón pero lo he incluido en el esquema para mayor claridad.

MVC2: el patrón MVC2 (Modelo Vista Controlador 2)es el que emplearé en la capa de presentación. *Struts* es un framework muy extendido que se basa en MVC2. El funcionamiento del patrón MVC2 se basa en que el controlador (*servlet*) recibe las peticiones que procesa, tras lo cual instancia el modelo (JavaBean o FormBean) e invoca a la vista (JSP) tras lo cual se contesta a la petición. En http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html hay un articulo interesante sobre la aplicación de MVC2 con JSP. De este articulo he extraído el siguiente esquema:



Session Façade: este patrón permite en una plataforma multicapa J2EE mostrar una "fachada" de la capa de lógica de negocio a la capa de presentación. Encapsula el acceso a la lógica de negocio presentando un nivel de abstracción elevado. La implementación del *Session Façade* se realiza con un *session bean.* Este patrón esta basado en el patrón *Façade* de GoF, y se describe en el libro *Core J2EE Patterns.*

Bussines Delegate: la función de este patrón es encapsular la complejidad de la comunicación remota con los componentes de negocio. Aunque se use en la parte del cliente, se considera un patrón de negocio ya que es el acceso de la lógica de negocio para el cliente y lo suele programar el programador de lógica de negocio. El *Bussines Delegate* suele usar el patrón *Service Locator* para hacer el "lookup" de los servicios. Se comunica directamente con el *Session Façade*.

DAO: el patrón DAO (Data Access Object) permite encapsular los accesos a la persistencia. Es un patrón de la capa de integración.

Service Locator: oculta la implementación del mecanismo de *lookup* del API JNDI.

Transfer Object: se usa para mover información entre capas. En Core J2EE Patterns se presenta como un patrón de lógica de negocio. Pero otros autores lo consideran un patrón multi-capa. También es conocido como *Value Object*.

Diagrama de despliegue

En el diagrama de despliegue se ve como desde el cliente, por medio del navegador web, se accede al servidor de aplicaciones (Jboss) y desde este se accede al SGBD (PostgreSQL) que están en otra máquina (aunque podría estar en la misma).

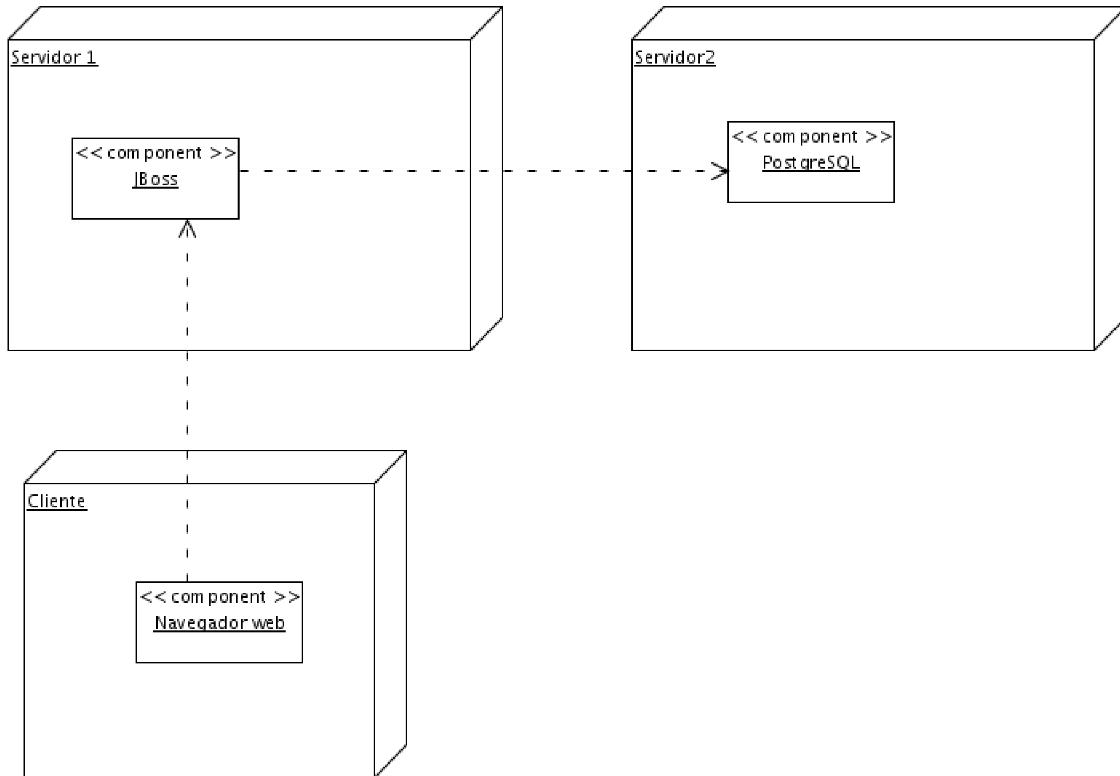


Diagrama de componentes

La aplicación consta de dos componentes. SoftwareTasks.jar es el componente que contiene la lógica de negocio, mientras que SoftwareTasks.war es el componente que contiene la lógica de presentación. El componente de lógica de negocio presenta unos interfaces a los que la lógica de presentación accede.

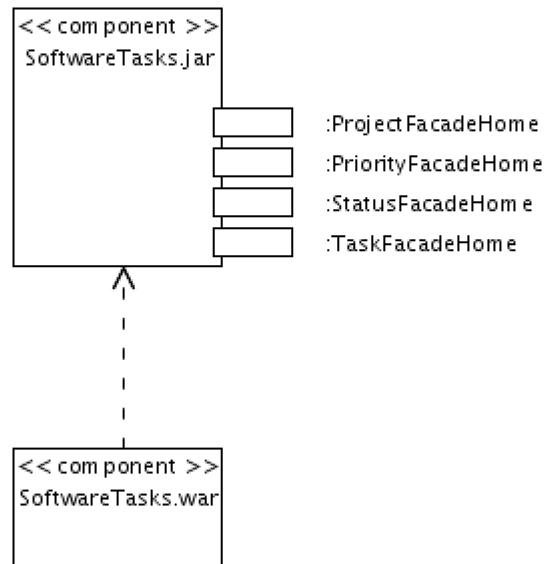
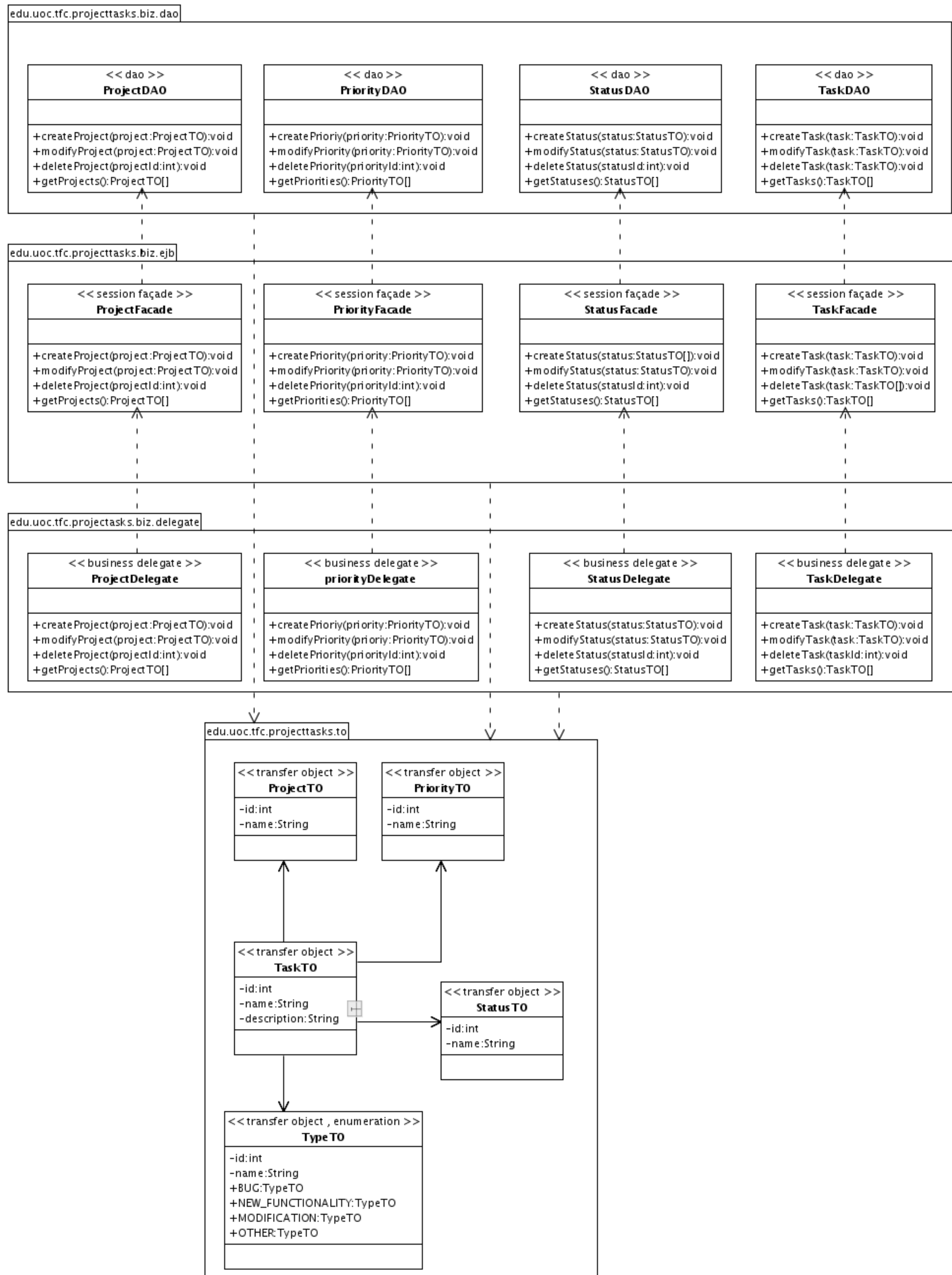
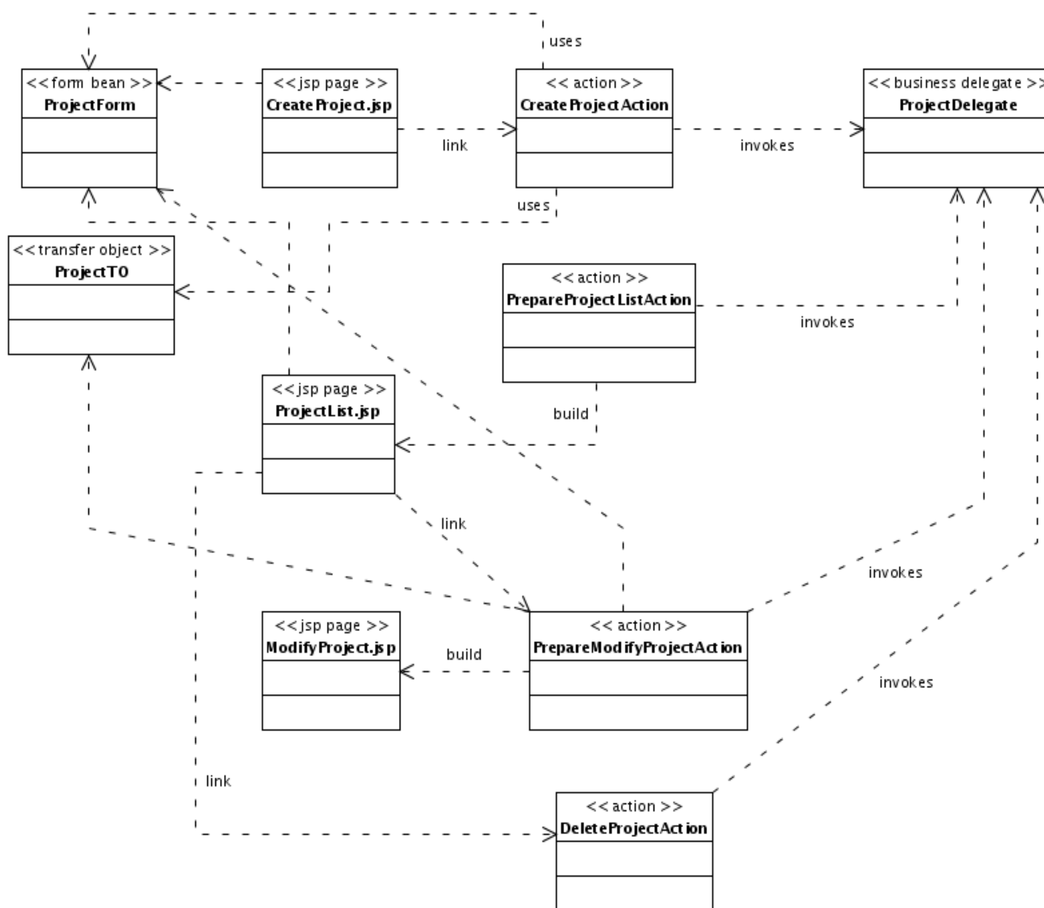


Diagrama de clases

El siguiente diagrama se corresponde con la capa de negocio e integración.



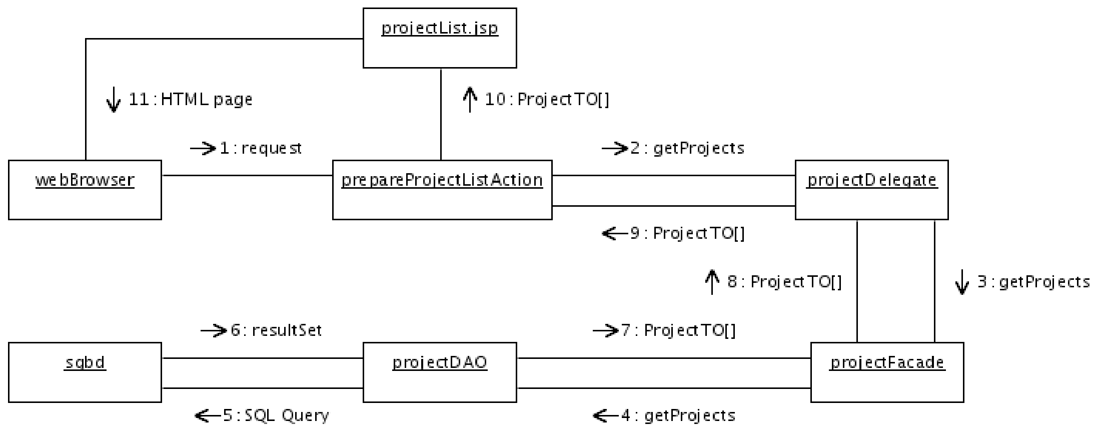
El siguiente diagrama se corresponde con la capa de presentación. Se ha incluido la clase *ProjectDelegate* que no pertenece estrictamente a la lógica de presentación.



Únicamente se hace referencia a las clase relacionadas con la entidad *proyecto* ya que los mecanismo relacionados con las demás entidades son muy similares.

Diagrama de colaboración

El siguiente diagrama de colaboración se refiere al caso de uso de mostrar todos los proyectos. Es representativo de los demás casos de uso.



Persistencia

La labor de persistencia se confía al sistema gestor de bases de datos PostgreSQL. A continuación se definen las tablas junto con las sentencias de creación .

Tabla: Project		
Campo	Tipo	Referencia a
ProjectId	serial	
Name	text	

```
CREATE TABLE Project (
      ProjectId serial PRIMARY KEY,
      Name text);
```

Tabla: Priority		
Campo	Tipo	Referencia a
PriorityId	serial	
Name	text	

```
CREATE TABLE Priority (
      PriorityId serial PRIMARY KEY,
      Name text);
```

Tabla: Status		
Campo	Tipo	Referencia a
StatusId	serial	
Name	text	

```
CREATE TABLE Status (
      StatusId serial PRIMARY KEY,
      Name text);
```

Tabla: Task		
Campo	Tipo	Referencia a
TaskId	serial	
Name	text	
Description	text	

Tabla: Task		
Campo	Tipo	Referencia a
ProjectId	int	Project
StatusId	int	Status
PriorityId	int	Priority
TypeId	int	

```
CREATE TABLE Task (
	TskId serial PRIMARY KEY,
	Name text,
	Description text,
	ProjectId int REFERENCES Project,
	StatusId int REFERENCES Status,
	PriorityId int REFERENCES Priority,
	TypeId int);
```

Detalles de implementación

ServiceLocator

Para implementar el patrón *ServiceLocator* he estudiado dos ejemplos. El primero ha sido el de la aplicación Java Adventure Builder que esta en los *blueprints de* Sun sobre Java. El código fuente está en:

http://java.sun.com/blueprints/code/adventure/1.0/src/com/sun/j2ee/blueprints/servicelocator/ejb/ServiceLocator.java.html

Por otro lado he mirado el ejemplo de Oracle:

http://www.oracle.com/technology/sample_code/tech/java/j2ee/vsm12/src/oracle/otnsamples/util/ServiceLocator.java.html

Me voy a quedar con el ejemplo de Oracle por su simplicidad. Aunque lo modificaré para implementar el patrón *Singleton* y no tener que instanciar la clase cada vez que se quiera usar.

Typesafe Enumeration

Los tipos de tareas no son configurables. Para representarlos usaré el patrón *Typesafe Enumeration* que es descrito en *Design Patterns: Elements of Reusable Object-Oriented Software.*

Este patrón permite representar un conjunto fijo de valores de manera segura. Cada valor de la enumeración posee una instancia publica. Existe un único objeto para cada valor.

Interfaz de usuario

Aquí se muestra una pantalla significativa de lo que puede ser el listado de tareas.

El menú está posicionado a la izquierda mientras que a la derecha aparecen los diálogos.

Implementación

En este apartado voy a comentar los aspectos, a mi juicio, más significaticos de la implementación.

Service Locator

La funcionalidad del *Service Locator* ya ha sido explicada en el apartado de diseño. Para implementar mi *Service Locator* me he basado en uno desarrolado por Oracle: (http://www.oracle.com/technology/sample_code/tech/java/j2ee/vsm12/src/oracle/otnsamples/util/ServiceLocator.java.html)

Lo he modificado para que implementara el patrón *Singlenton* y para que no haya que hacer casting de los diferentes objetos devueltos.

El resultado es el siguiente:

```
package edu.uoc.projecttasks.util;

import java.util.Hashtable;

import javax.ejb.EJBHome;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import javax.sql.DataSource;

/*
 * Basado en el ServiceLocator de Oracle
 *
 */

public final class ServiceLocator {

  // Cache of objects in JNDI tree
  private static Hashtable homeCache = new Hashtable();

  // Initial context
  private static InitialContext defaultContext = getContext();

  private ServiceLocator() {
  }

  private static InitialContext getContext() {
      try {
        return new InitialContext();
    } catch (NamingException e) {
        throw new ServiceLocatorException(e.getMessage());
    }
  }

  /**
   * Method to return an object in the default JNDI context, with  the supplied
   * JNDI name.
   * @param <b>jndiName</b> The JNDI name
   * @return <b>Object</b> The object in the JNDI tree for this name.
   * @throws <b>UtilityException</b> Exception this method can throw
   */
```

```
  public static Object getService(String jndiName) {
    try {
      if (!homeCache.containsKey(jndiName)) {

        // If the service is not in the cache, get the object for the
        // supplied jndi name and put it in the cache
        homeCache.put(jndiName, defaultContext.lookup(jndiName));
      }
    } catch (NamingException ex) {
      throw new ServiceLocatorException("Exception thrown from getService "
              + "method of ServiceLocator class : "
              + ex.getMessage());
    } catch (SecurityException ex) {

      throw new ServiceLocatorException("Exception thrown from getService "
              + "method of ServiceLocator class : "
              + ex.getMessage());
    }

    // Return object from cache
    return homeCache.get(jndiName);
  }

  public static Integer getInteger(String jndiName) {
      return (Integer) getService(jndiName);
  }

  public static String getString(String jndiName) {
      return (String) getService(jndiName);
  }

  public static DataSource getDataSource(String jndiName) {
      return (DataSource) getService(jndiName);
  }

  public static EJBHome getRemoteHome(String jndiName, Class className) {
      return (EJBHome) PortableRemoteObject.narrow(
              getService(jndiName),
              className);

  }

}
```

XDoclet

XDoclet (http://xdoclet.sf.net/) es una herramienta de generación automática de código. La aplicación que he desarrollado hace uso de los *Stateless Session Beans*, que son bastante engorrosos de programar debido a que hay que realizar varias clases para uno solo. Gracias a XDoclet basta con programar un única clase y las demás se generan automáticamente. Por ejemplo, para programar *PriorityFacade* se han insertado en el código fuente los siguientes atributos XDoclet:

```
/**
 * @ejb.bean name="PriorityFacade"
 *           display-name="Name for PriorityFacade"
 *           description="Description for PriorityFacade"
 *           jndi-name="ejb/PriorityFacade"
 *           type="Stateless"
 *           view-type="both"
 */
```

Tags

En la capa de presentación las tareas son muy similares. Para no tener que realizar código JSP muy similar en cada JSP he utilizado tags de JSP. El más útil me ha sido el que he empleado para los listados:

```
<%@ tag language="java" %>

<%@ variable name-given="elementList" %>

<%@ attribute name="modifyAction" required="true" %>
<%@ attribute name="deleteAction" required="true" %>
<%@ attribute name="groupName" required="true" %>

<%@ taglib uri="/WEB-INF/tlds/c.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html"%>

<% int count=0;%>

<c:forEach var="element" items="${elementList}">
	<tr>
		<td class="<%= (count % 2==0 ?"bodytablerow1":"bodytablerow2") %>">
			<form name=<c:out value="${groupName}"/><%= count %> method="post" 
action="/ProjectTasksWeb//<c:out value="${modifyAction}"/>">
				<input type="hidden" name="id" value="<c:out 
value="${element.id}"/>">
				<input type="hidden" name="name" value="<c:out 
value="${element.name}"/>">
				<a href="javascript:document.<c:out value="${groupName}"/><%= 
count %>.submit()">
					<c:out value="${element.name}"/>
				</a>
			</form>
		</td>
		<%-- boton eliminar --%>
		<td class="bodytablerow1">
			<form method="post" action="/ProjectTasksWeb//<c:out 
value="${deleteAction}"/>" onsubmit="return confirmDelete('<c:out 
value="${element.name}"/>')">
				<input type="hidden" name="id" value="<c:out 
value="${element.id}"/>">
				<input type="hidden" name="name" value="<c:out 
value="${element.name}"/>">
				<html:submit styleClass="buttoncolornormal" onmouseover = 
"this.className='buttoncolorover'" onmouseout = "this.className='buttoncolornormal'">
					Eliminar
				</html:submit>
			</form>
		</td>
	</tr>
	<% count++;%>
</c:forEach>
```

Type Safe Enumeration

Para implementar el patrón *Type Safe Enumeration* he hecho uso del *Enum* de Java 5. Lo cual ha simplificado bastante las cosas.

Ha quedado así:

```
package edu.uoc.tfc.projecttasks.to;
```

```java
import java.io.Serializable;

public enum TypeTO implements Serializable {

     BUG(1, "bug"), NEW_FUNCTIONALITY(2, "Nueva funcionalidad"),
     MODIFICATION(3, "Modificación"), OTHER(4, "Otros");

     private int id;
     private String name;

     private TypeTO(int id, String name) {
            this.id = id;
            this.name = name;
     }

     public int getId() {
            return id;
     }

     public void setId(int id) {
            this.id = id;
     }

     public String getName() {
            return name;
     }

     public void setName(String name) {
            this.name = name;
     }
}
```

Conclusiones

La realización de un proyecto J2EE no es trivial. Los patrones de diseño especificos para J2EE son una orientación básica, por lo cual es necesario tener conocimientos de diagramación UML. Debido a lo tedioso de la programción de las interfaces, locales y remotas, es muy recomendable recurrir a Xdoclet para crear código de forma automática.

En la parte de presentación la base es el conocimiento del HTML. Conociendo HTML no es muy complicado entender la filosofia del JSP. Dentro de JSP hay una variedad de librerias que se pueden escoger. En el caso de este proyecto he optado por las que parecen ser más difundidas Struts y JSTL. Struts además de proporcionar librerias de tags proporciona un *Front Controller* que implemeta MVC2. Si no se dispusiera del framework Struts , o de cualquier otro, la programación de la parte de presentación sería mucho más compleja.

La conclusión final es que programar una aplicación J2EE requiere de una curva de aprendizaje elevada. A cambio J2EE proporciona una serie de ventajas: separación de lógica de negocio y presentación, seguridad controlada por el servidor de aplicaciones y robustez de las aplicaciones, entre otras ventajas.

Guía de instalación

Lo explicado aquí se refiere a la instalación de la aplicación en un servidor de aplicaciones JBoss. Se ha probado en un JBoss versión 4.0.3.. Es de esperar que funcione en otras versiones, aunque no ha sido verificado.

Paso1

Crear en el directorio de despliegue un fichero, correspondiente al *data source,* llamado *tasks-ds.xml* con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ===================================================================== -->
<!--                                                                       -->
<!--  JBoss Server Configuration                                           -->
<!--                                                                       -->
<!-- ===================================================================== -->

<!-- $Id: postgres-ds.xml,v 1.3 2004/09/15 14:37:40 loubyansky Exp $ -->
<!-- ==================================================================== -->
<!--  Datasource config for Postgres                                      -->
<!-- ==================================================================== -->


<datasources>
  <local-tx-datasource>
    <jndi-name>TasksDS</jndi-name>
    <connection-url>jdbc:postgresql://localhost:5432/tasks</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>javier</user-name>
    <password></password>
        <!-- sql to call when connection is created
        <new-connection-sql>some arbitrary sql</new-connection-sql>
        -->

        <!-- sql to call on an existing pooled connection when it is obtained from pool
        <check-valid-connection-sql>some arbitrary sql</check-valid-connection-sql>
        -->

      <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional) -->
      <metadata>
         <type-mapping>PostgreSQL 8.0</type-mapping>
      </metadata>
  </local-tx-datasource>

</datasources>
```

Es posible, dependiendo de la configuración de la base de datos, que haya que modificar el usuario y contraseña del fichero.

Paso 2

Crear la estructura de la base de datos con las siguientes sentencias:

```
CREATE TABLE Project (
      ProjectId serial PRIMARY KEY,
      Name text);
CREATE TABLE Priority (
      PriorityId serial PRIMARY KEY,
      Name text);
CREATE TABLE Status (
      StatusId serial PRIMARY KEY,
      Name text);
CREATE TABLE Task (
      TskId serial PRIMARY KEY,
      Name text,
      Description text,
      ProjectId int REFERENCES Project,
      StatusId int REFERENCES Status,
      PriorityId int REFERENCES Priority,
      TypeId int);
```

Paso 3

Copiar en el directorio de librerias del servidor las libreris correspondientes a Struts y a JSTL.

Las librerias de Struts están disponibles en http://apache.rediris.es/struts/library/struts-1.2.8-lib.zip.

Las librerias de JSTL están disponibles en http://www.apache.org/dist/jakarta/taglibs/standard/jakarta-taglibs-standard-current.zip

Paso 4

Copiar en el directorio de despliegue el fichero ProjectTasksApplication.ear.

Paso 5

Arrancar JBoss y acceder a http://localhost:8080/ProjectTasksWeb/