6 APLICACIÓN EJEMPLO CON J2EE

A continuación se realizará el diseño y desarrollo de una aplicación empresarial con movilidad. El objetivo del desarrollo de esta aplicación será mostrar la potencia de la tecnología J2EE, tecnología propuesta para el desarrollo de este tipo de aplicaciones.

Con este ejemplo se intentará mostrar los aspectos básicos de toda aplicación empresarial y sus características de movilidad. Una de los aspectos que exigiremos a esta aplicación ejemplo será la escalabilidad, ya que en capítulos posteriores ampliaremos los servicios ofrecidos para ser accesibles a través de SMS.

6.1 ANÁLISIS DE OBJETIVOS Y NECESIDADES DE LA APLICACIÓN

La aplicación deberá poseer y mostrar los requisitos básicos de toda aplicación empresarial: acceso a datos, manejo de transacciones, estabilidad, disponibilidad, seguridad e integración. Además se intentarán abaratar los costes de desarrollo de mantenimiento.

Se pretende diseñar y desarrollar una aplicación que ofrezca un servicio en el marco de la empresa. Se procederá en dos fases. Primero se diseñará y desarrollará una aplicación con características básicas de movilidad, y posteriormente se realizará una ampliación para proporcionar servicios vía SMS.

Es fundamental que la aplicación muestre la potencia de la tecnología escogida. La escalabilidad será un factor imprescindible, teniendo en cuenta como segunda fase del diseño implementaremos una ampliación. Además tendrá que ser capaz de hacer frente a otras posibles ampliaciones, actualizaciones, cambios o posibles migraciones de algunas de sus partes, sin que sea necesario sustituir la aplicación completa.

Por último debemos diseñar y desarrollar una aplicación real, útil en el marco empresarial en el que pretendamos movernos.

6.2 ÁMBITO Y ALCANCE

Para la realización de la aplicación debemos concretar el alcance de ésta. Vamos a centrarnos en uno de los departamentos de la organización y en un servicio en concreto que pueda ofrecer.

El objetivo general de la aplicación será ofrecer un servicio Web que posteriormente se ampliará para ofrecerlo también vía SMS. Como caso particular nos centraremos en un servicio de Solicitud de Vacaciones. Este servicio lo ofrecerá el área de Recursos Humanos de la Organización.

Este servicio se centrará en el alta de usuarios a este servicio y en la Solicitud de Vacaciones por parte de los usuarios dados de alta. No se implementarán los criterios de la concesión o no de estas vacaciones, ya que estos pueden ser muy diversos y podrían existir otras herramientas que lo implementaran. Si fuese este el caso, estas herramientas podrían integrarse con nuestra aplicación. La aplicación se centrará en la petición de vacaciones y no implementará el servicio de solicitud de permisos. Este servicio de petición de servicios podría realizarse posteriormente como ampliación del servicio inicial.

Como primera fase se diseñará e implementará un servicio de Solicitud de Vacaciones vía Web. A continuación se diseñará e implementará una ampliación de este servicio para que la solicitud de vacaciones pueda realizarse vía SMS.

También se incluirá en el diseño, un servicio de perfil de usuario con datos y permisos asociados a ese usuario.

6.2.1 Utilidad del servicio

Actualmente, en la mayoría de las empresas exigen a los empleados la entrega de un formulario para la solicitud de vacaciones y permisos. Con estos formularios de solicitud de vacaciones y permisos se pretende llevar un mejor control sobre la disponibilidad de los empleados, permitiendo por una parte a la empresa disponer de los recursos suficientes (empleados) en cada momento, y por otra parte, distribuir los períodos de vacaciones entre los empleados de modo que satisfaga en lo posible las expectativas de éstos.

La utilización de estos formularios, exige a los implicados, la presencia en las oficinas correspondientes para realizar la entrega. Teniendo en cuenta, que cada vez es mayor el número de empleados que trabajan desplazados de sus oficinas, la entrega de estos formularios puede resultar incómoda, e incluso, imposible entregarlos en un plazo adecuado para su tramitación.

La implementación de este servicio, para ser accesible vía Web, supondría, no sólo una automatización de los trámites y facilidad para el personal administrativo, si no también, la oportunidad de realizar la petición de formularios en cualquier momento, y desde cualquier lugar en el que se puede disponer de una conexión a Internet y de un navegador.

Aún así, el servicio vía Web puede representar restricciones a aquellos usuarios que debido a sus circunstancias no puedan disponer fácilmente de los requisitos necesarios. Por esto, se plantea además, implementar este servicio de modo que sea accesible vía SMS. Aunque no pueden disponer de ordenador e Internet, la mayoría de los usuarios dispondrán de un

dispositivo móvil que permita el envío y recepción de SMS. Esto permitirá que todos los empleados, desplazados o no, se encuentren ante las mismas condiciones a la hora de solicitar sus días de vacaciones o permisos.

6.3 IDENTIFICACIÓN DE REQUISITOS

El servicio principal que se pretende dar con esta aplicación es el Servicio de Solicitud de Vacaciones. Sin embargo, junto a este servicio, incluimos otros servicios complementarios que aportarán valor añadido a nuestra aplicación.

6.3.1 Agentes

Se considerarán dos agentes principales: usuario autentificado y administrador. Todo usuario que accede al sistema debe autentificarse. Al autentificarse se comprobará si el usuario es administrador o no, lo que le permitirá realizar unas u otras operaciones, o acceder a unos u otros servicios.

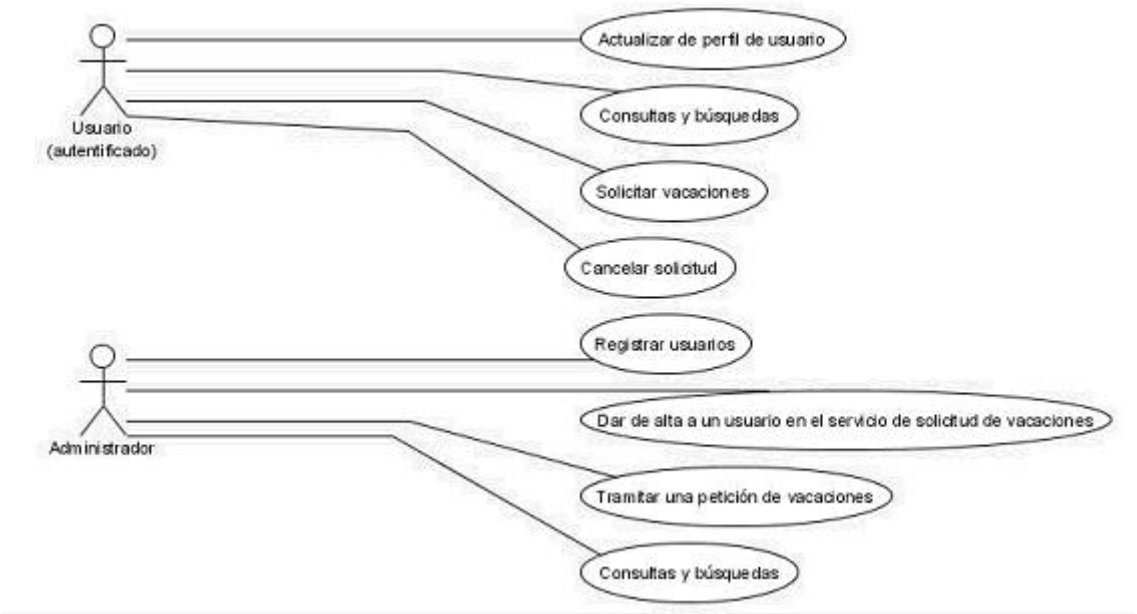
El administrador, aunque puede realizar operaciones extra, y acceder a más servicios, también puede realizar las mismas operaciones de cualquier otro usuario o acceder a sus servicios. Por esto, el término usuario se utilizará indistintivamente para cualquier usuario antes de entrar en el sistema, para un usuario autentificado sin permiso de administrador, o para un administrador que realiza las funciones de cualquier usuario.

6.3.2 Actividades

Un usuario podrá acceder a los servicios de Actualización de Perfil de Usuario, Solicitud de Vacaciones, Consultas y Búsquedas.

Las principales funciones que sólo podrá realizar un administrador serán Registro de Usuarios, Alta de Usuarios al Servicio de Solicitud de Vacaciones y Tramitación de Solicitud de Vacaciones. El Servicio de Consultas y Búsquedas al que tiene acceso el administrador será más amplio que el servicio al que acceden el resto de usuarios.
Además junto a las operaciones propias del administrador, este podrá realizar actualizaciones en su propio perfil o en el de otro usuario y realizar solicitudes para cualquier usuario.

A continuación se incluye un modelo con los casos de uso y agentes principales.

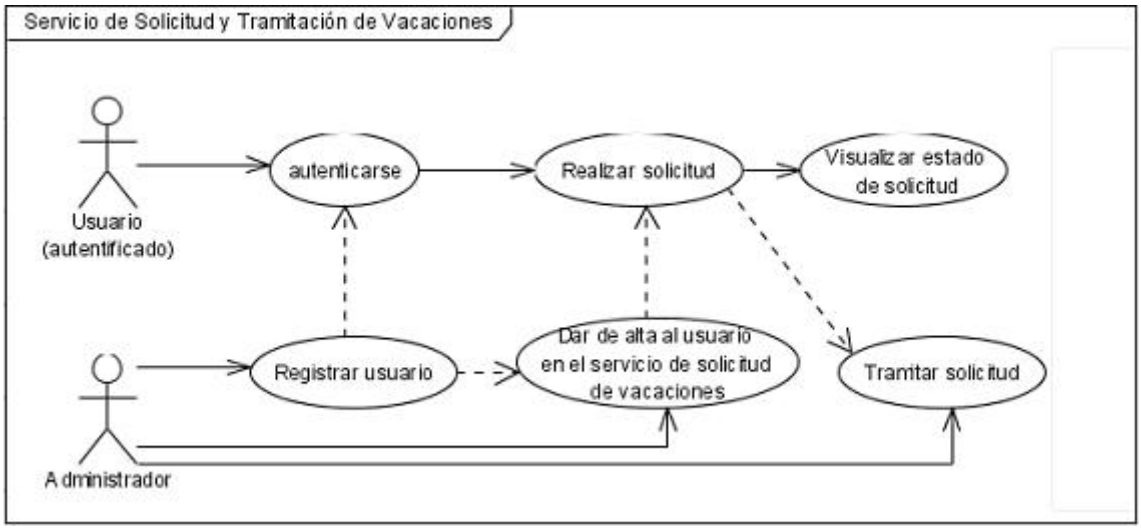


F. 6-1: Aplicación ejemplo. Modelo básico. Actividades y Agentes.

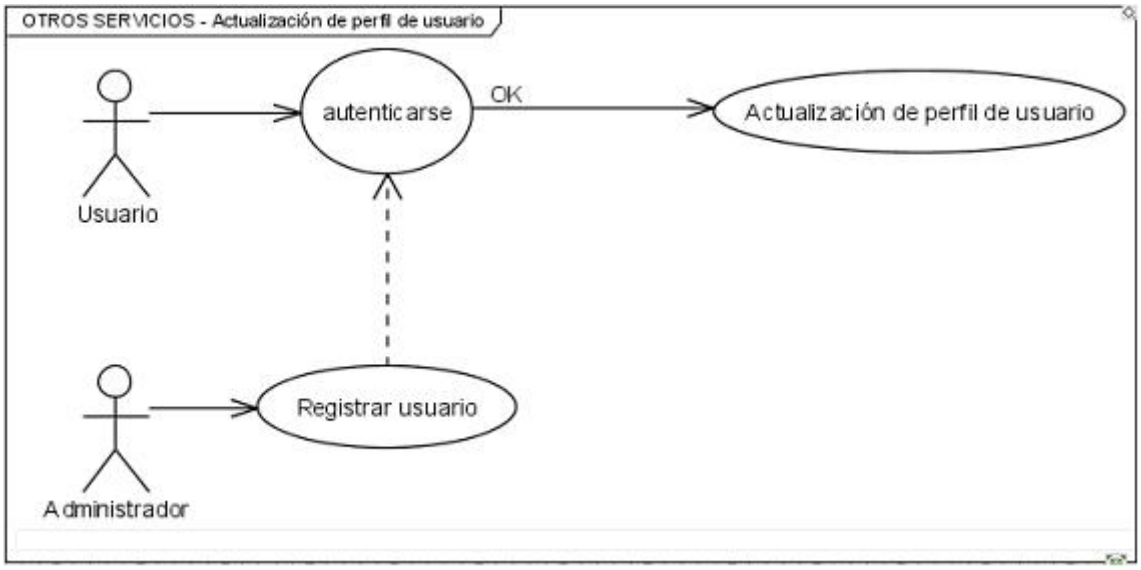
Un usuario podrá acceder al sistema si previamente ha sido registrado en el sistema por un administrador. Igualmente, para que un usuario pueda ser dado de alta en el Servicio de Solicitud de Vacaciones, el usuario debe estar registrado en el sistema.

Para que un usuario pueda realizar una solicitud de vacaciones debe estar dado de alta en el servicio. El Servicio de Solicitud de Vacaciones finalizará cuando se realice la tramitación por parte del administrador y el usuario sea informado del resultado. No se implementará el criterio de concesión o no de cierto período de vacaciones, esta decisión puede estar condicionada por agentes externos.

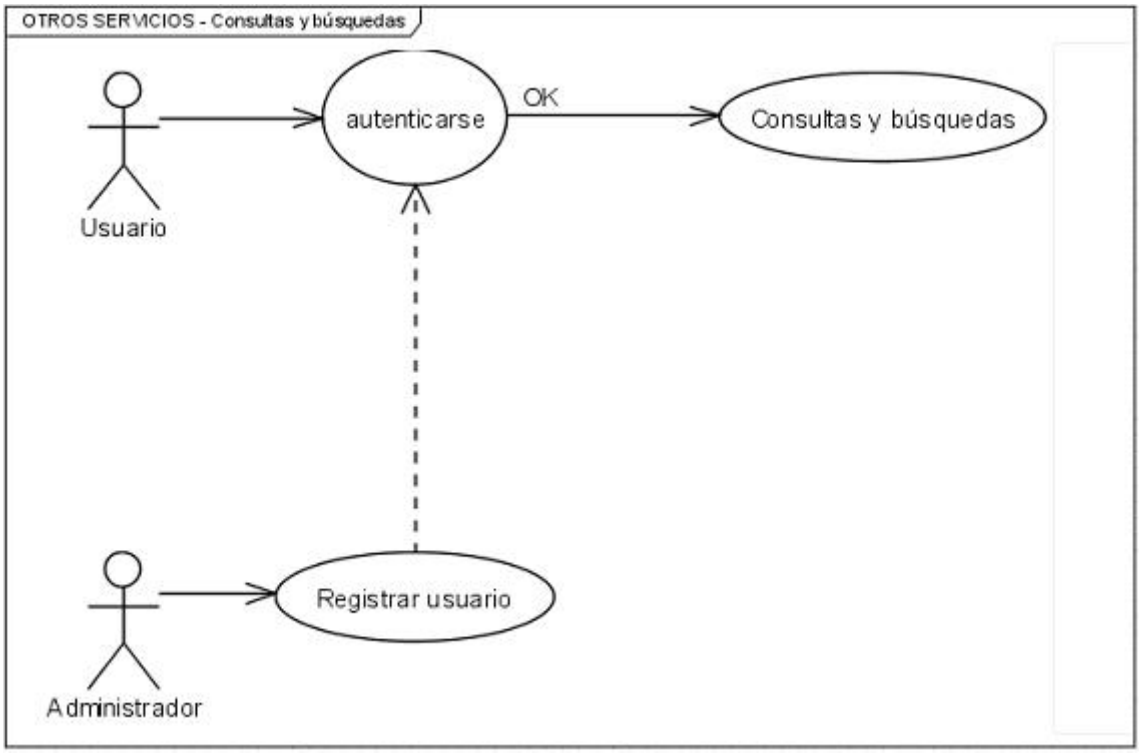
Se muestra a continuación un diagrama general del Servicio de Solicitud de Vacaciones y otros de los servicios proporcionados.



F.6-2: Aplicación ejemplo. Esquema general. Servicio de Solicitud y Tramitación de Vacaciones.



F.6-3: Aplicación ejemplo. Esquema general. Actualización de Perfil de Usuario.

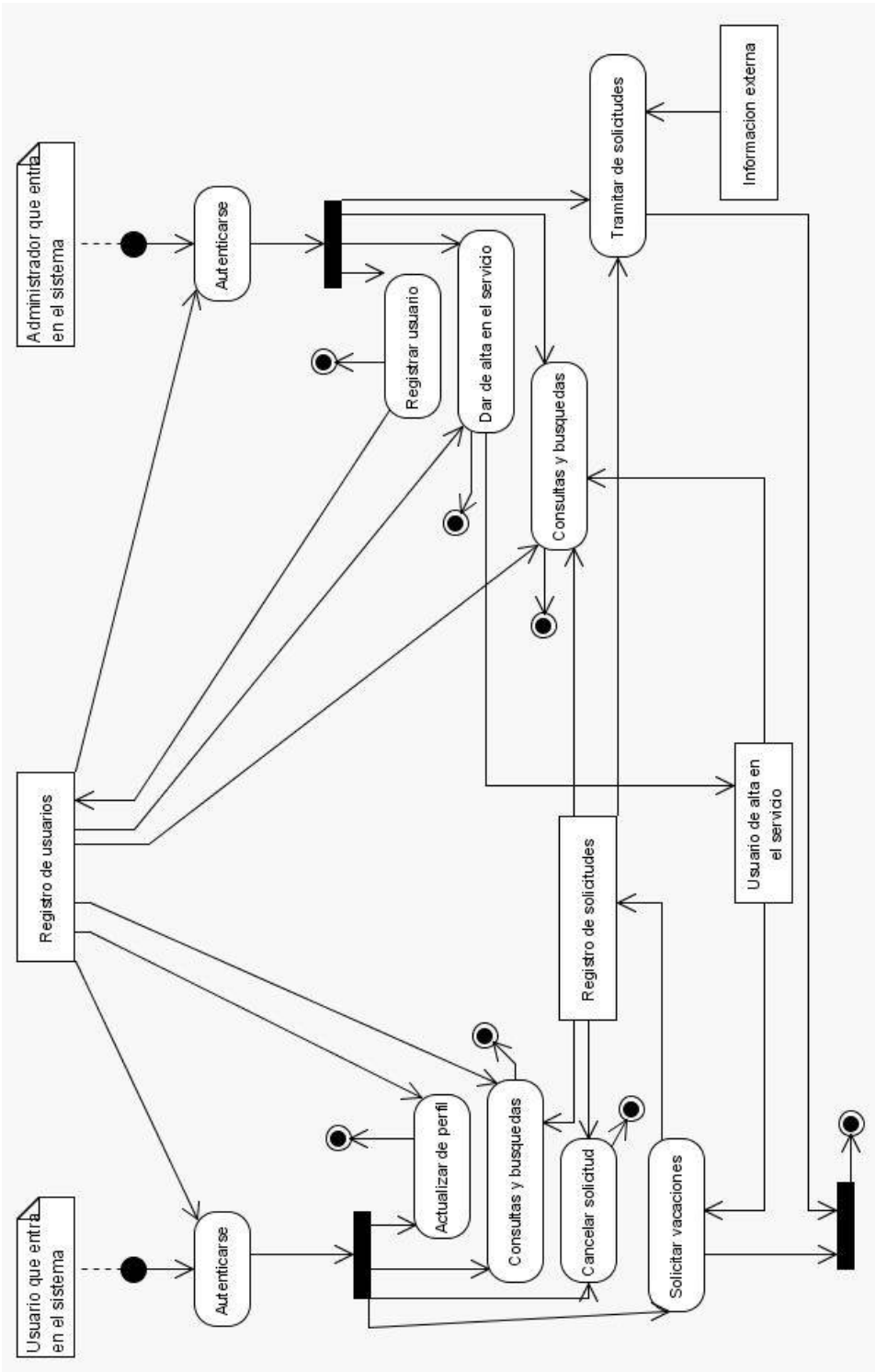


F.6-4: Aplicación ejemplo. Esquema general. Consultas y Búsquedas.

Un usuario podrá consultar algunos datos del perfil de otros usuarios, y los usuarios registrados en el sistema. No tendrá acceso al registro de usuarios dados de alta en el servicio de solicitud de vacaciones no podrá realizar consultas sobre los perfiles de vacaciones de otros usuarios.

Un usuario tendrá acceso a su Perfil de Usuario y su Perfil de Vacaciones. El administrador podrá acceder a todos los perfiles de todos los usuarios y a todos los registros de usuarios, de alta y de solicitudes.

Se muestra a continuación un diagrama de actividades básico.

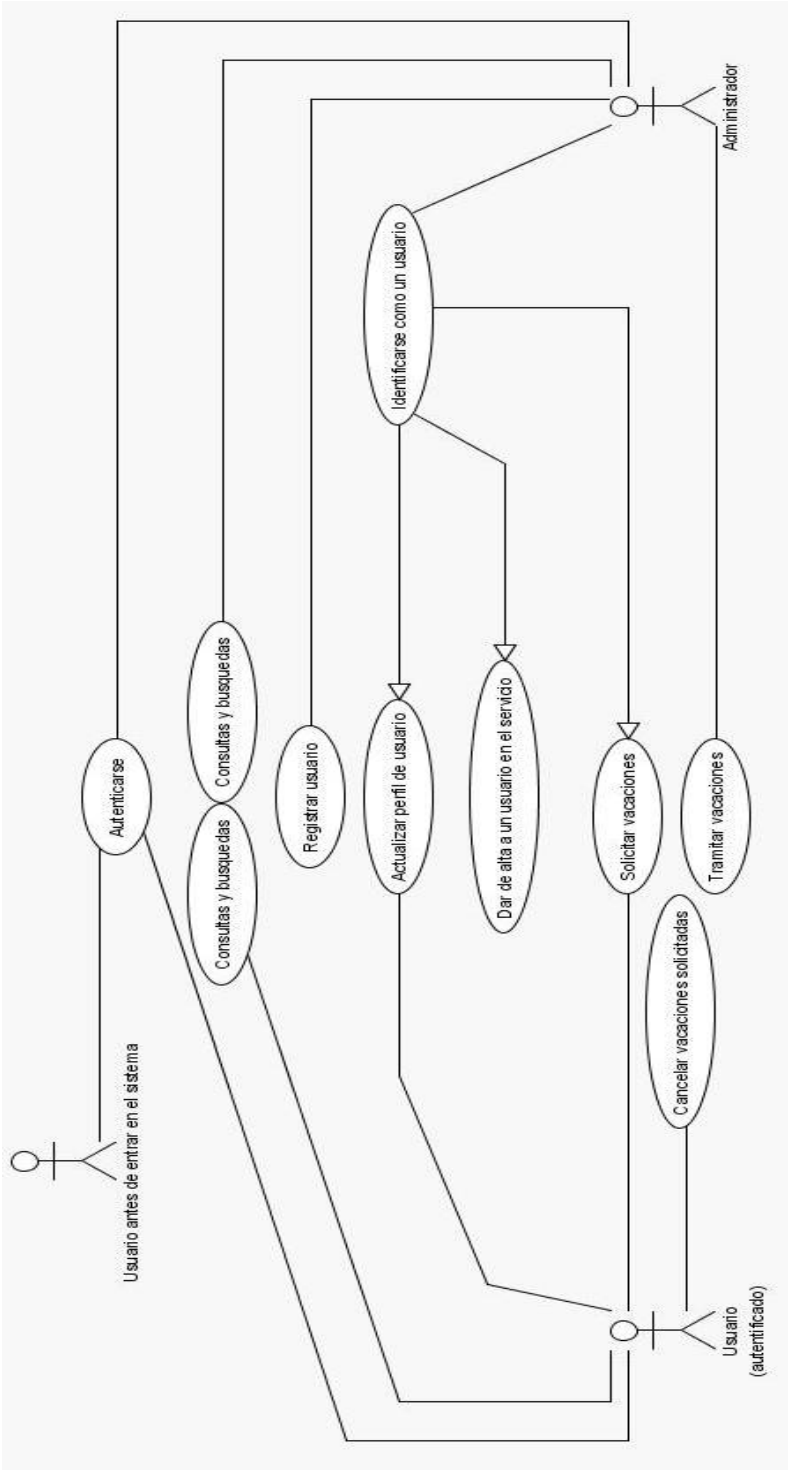


F.6-5: Aplicación ejemplo. Modelo básico. Diagrama de actividades.

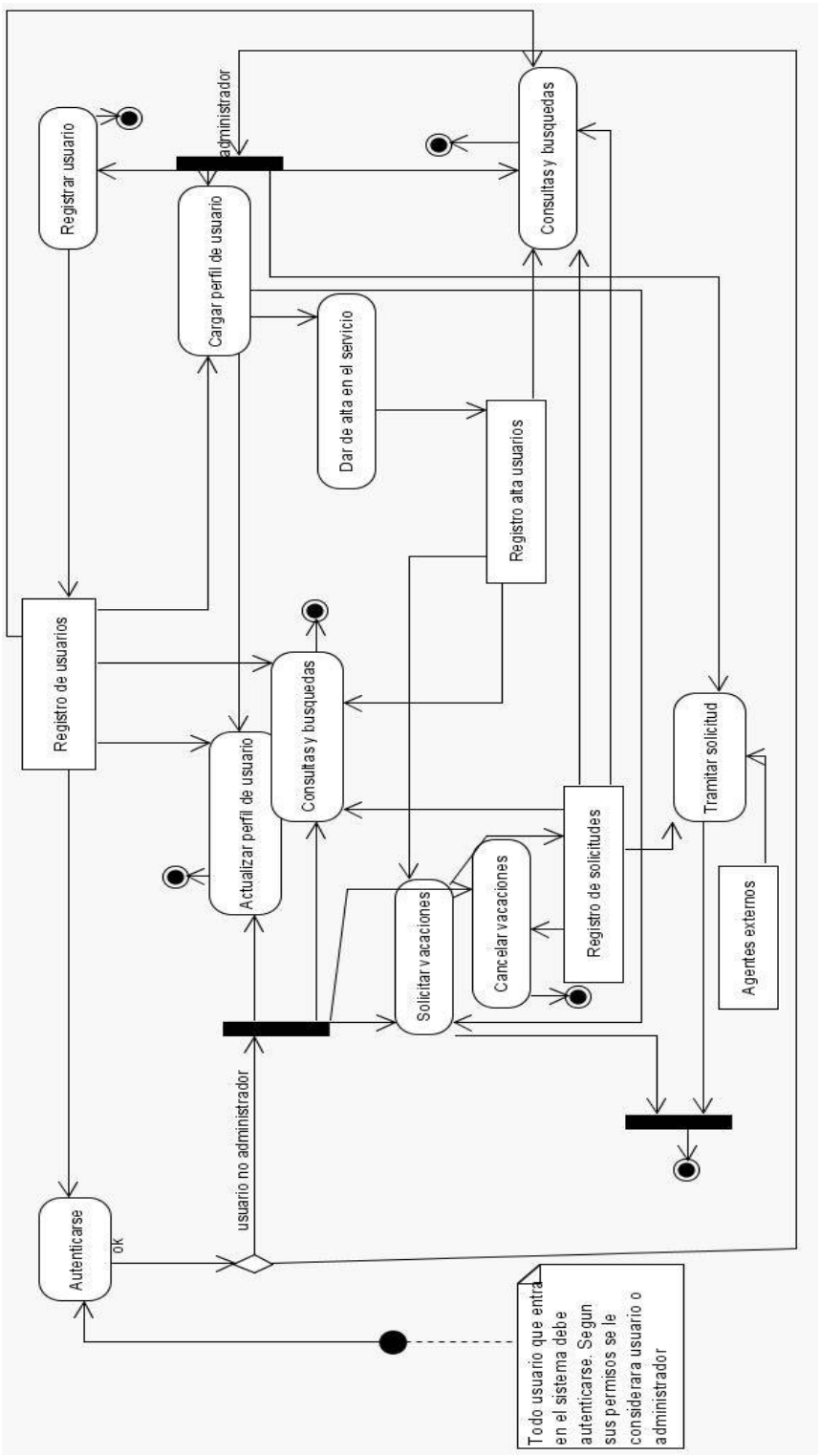
Se ha hablado de las operaciones propias de un usuario, y las operaciones propias de un administrador. Para completar el modelo es necesario contemplar las operaciones que un administrador puede realizar en los perfiles de otros usuarios.

Para que un administrador pueda realizar operaciones sobre los perfiles de cualquier otro usuario tendrá que volver a autentificarse como el usuario sobre el que se quiere trabajar para que sus perfiles se carguen en el sistema. Entre estas operaciones se incluyen actualizaciones de perfil de usuario, dar de alta al usuario en algún servicio, solicitar vacaciones y realizar algunas consultas relacionadas con perfiles del usuario.

A continuación se incluyen los diagramas completos de casos de uso y actividades. Se incluyen también las especificaciones de cada una de las actividades involucradas.



F.6-6: Aplicación ejemplo. Modelo completo. Diagrama de Casos de Uso.



F.6-7: Aplicación ejemplo. Modelo completo. Diagrama de actividades.

ACTIVIDAD: AUTENTICARSE

AGENTE: Usuario y administrador

PRECONDICIONES:

Para que la autenticación de un usuario o administrador se complete con éxito, el usuario o administrador su perfil debe estar incluido en el registro de usuarios.

POSTCONDICIONES:

Una vez autentificado en el sistema los perfiles del usuario estarán disponibles según los permisos del usuario autentificado.

ACTIVIDAD: CARGAR PERFIL DE USUARIO (IDENTIFICARSE COMO OTRO USUARIO)

AGENTE: Administrador

PRECONDICIONES:

El usuario cuyos perfiles se quieren cargar en el sistema debe estar incluido en el registro de usuarios.

POSTCONDICIONES:

Una vez cargado los perfiles de usuario en el sistema el administrador podrá operaciones sobre ellos.

ACTIVIDAD: REGISTRAR USUARIO

AGENTE: Administrador

PRECONDICIONES:

El usuario que realiza la acción debe estar autentificado y ser administrador.

POSTCONDICIONES:

Se actualiza el registro de usuarios.

ACTIVIDAD: ACTUALIZAR PERFIL

AGENTE: Usuario o administrador

PRECONDICIONES:

El usuario debe estar registrado. Los permisos de accesos sólo pueden ser modificados por un administrador.

IMPLEMENTACIÓN: Esta acción se divide en varias acciones:

- Cambiar contraseña.
- Otorgar nueva contraseña (sólo administrador).
- Actualizar detalles contacto.
- Actualizar detalles de acceso al sistema (sólo administrador).
- Actualizar otros detalles de perfil de usuario.

ACTIVIDAD: DAR DE ALTA EN EL SERVICIO

AGENTE: Administrador

PRECONDICIONES:

El usuario al que se pretende dar de alta, debe aparecer en el registro de usuarios. El administrador debe haber realizado previamente la acción de cargar perfil del usuario al que se pretende dar de alta en el servicio.

POSTCONDICIONES:

El usuario se añade en el registro alta usuarios para un período.

ACTIVIDAD: SOLICITAR VACACIONES

AGENTE: Usuario o administrador

PRECONDICIONES:

El usuario debe estar dado de alta en el servicio para el período solicitado.

POSTCONDICIONES:

La solicitud se añade en el registro de solicitudes.

ACTIVIDAD: TRAMITAR VACACIONES

AGENTE: Administrador POSTCONDICIONES: Cambia el estado de la solicitud a tramitada.

ACTIVIDAD: CANCELAR SOLICITUD DE VACACIONES

AGENTE: Usuario POSTCONDICIONES: Cambia el estado de la solicitud a cancelada.

ACTIVIDAD: CONSULTAS Y BÚSQUEDAS

AGENTE: Usuario o administrador

PRECONDICIONES:

El usuario debe estar registrado para acceder a sus distintos perfiles. El administrador debe cargar los perfiles del usuario para realizar algunas consultas sobre él.

IMPLEMENTACIÓN: Usuario y administrador podrán realizar distintos tipos de búsquedas sobre los distintos registros:

- Búsqueda de usuarios: Acceso a la lista de usuarios del sistemas y detalles de contacto. El usuario administrador podrá solicitar ver el perfil completo de un usuario.

- Búsqueda de servicio de vacaciones: Un usuario sólo tendrá acceso a consulta de su perfil de servicio para los distintos períodos anuales. Un usuario administrador podrá solicitar ver el perfil del usuario cargado en sesión o ver los servicios de todos los usuarios para un determinado período.

- Búsqueda de solicitud de vacaciones: Un usuario sólo tendrá acceso de consulta de sus solicitudes de vacaciones. Un usuario administrador podrá realizar consultas y búsquedas sobre todas las solicitudes presentadas por cualquier usuario.

OTRAS ACTIVIDADES

AGENTE: Usuario o administrador

Junto a las actividades anteriores, los usuarios podrán imprimir los resultados de las búsquedas realizadas, así como los perfiles a los que tengan acceso. También se facilitarán impresiones de formularios y manuales utilizados.

6.3.3 Objetos de datos.

De los diagramas de actividades anteriores pueden extraerse los objetos de información implicados en el sistema.

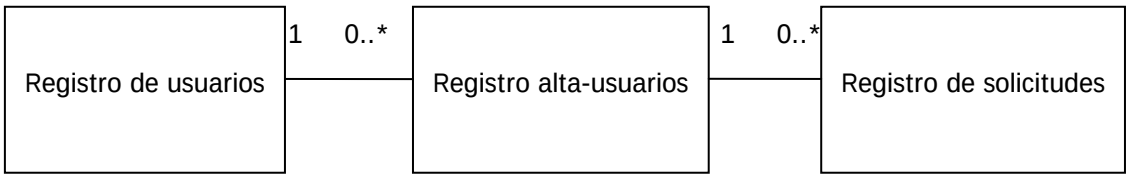
Los objetos de información serán: Registro de Usuarios, Registro de Alta de Usuarios (Registro de usuarios dados de alta en el Servicio de Solicitud de Vacaciones) y Registro de Solicitudes de Vacaciones. Podrían existir objetos de información externa que podrían estar involucrados en la concesión o no de un período de vacaciones a un usuario.

El Registro de Usuario está formado por todos los perfiles de usuarios registrados en el sistema. Al registrar un usuario el administrador añade un Perfil de Usuario al Registro de Usuarios.

Al dar de alta a un usuario en el Servicio de Solicitud de Vacaciones, el administrador añade un usuario y período de alta en el Registro de Alta de Usuarios. El usuario sólo estará dado de alta por el período o períodos incluidos en el registro.

El Registro de Solicitudes de Vacaciones estará formado por todas las solicitudes de vacaciones realizadas en cualquier estado de tramitación.

El modelo básico de datos es el que se muestra a continuación.

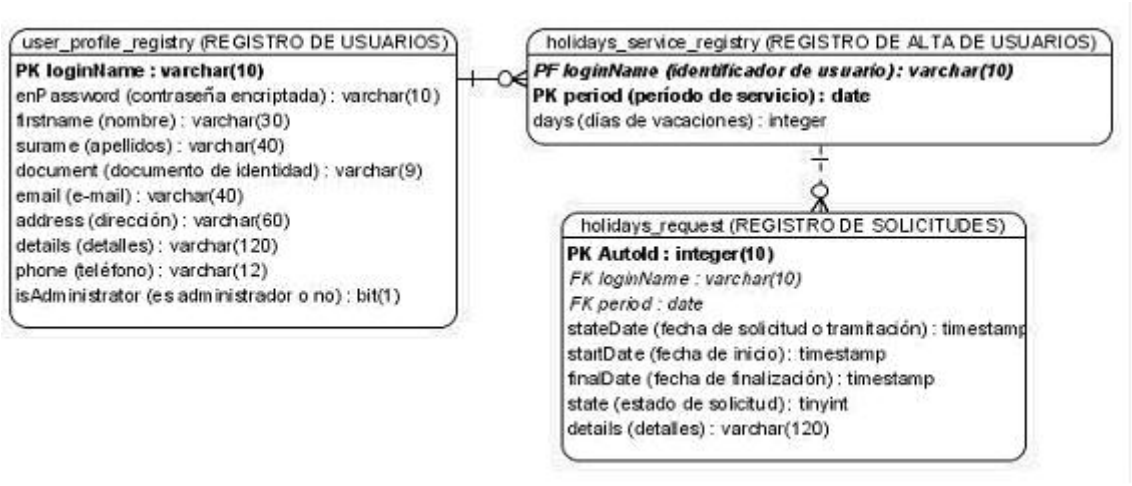


F.6-8: Aplicación ejemplo. Modelo básico. Sistema de datos.

Si especificamos cada uno de los campos necesarios para el perfil de usuario, alta de usuario y solicitud de vacaciones tendremos el siguiente modelo.

Los atributos marcados por un + precedente, pertenecerán a la clave primaria de la tabla de datos. Estos atributos identificarán única e inequívocamente cada fila de la tabla.

Los atributos marcados por un # precedente, formarán parte de la clave foránea. La clave foránea se incluye por motivos de integridad referencial en la base de datos, no tiene efectos en la aplicación.



F.6-9: Aplicación ejemplo. Modelo completo. Sistema de datos.

A continuación incluimos información más detallada de cada uno de los objetos de información.

OBJETO DE INFORMACIÓN: REGISTRO DE USUARIOS

Atributos:

- identificador de usuario
- contraseña encriptada
- nombre
- apellidos
- documento: número del documento de identidad
- email
- dirección
- notas sobre el usuario
- telefono
- esAdministrator: indica si el usuario tiene o no permiso de administrador

Restricciones:

- El identificador de usuario identificará única e inequívocamente a cada usuario.
- El identificador de usuario, la contraseña encriptada y es campo esAdministrador no pueden ser nulos.
- Por defecto todo usuario que es registrado lo hace sin permisos de administrador, a no ser que se indique lo contrario.
- CLAVE PRIMARIA: + (identificador de usuario).

OBJETO DE INFORMACIÓN: REGISTRO DE ALTA DE USUARIOS

Atributos:

- identificador de usuario
- período: período en el que un usuario es dado de alta, se corresponde con un periodo anual, es decir, un año
- días: días de vacaciones que corresponden a un usuario en el período en el que se ha dado de alta.

Restricciones:

- El conjunto identificador de usuario y período identificarán única e inequívocamente un usuario dado de alta en un período.
- Un usuario debe existir en el Registro de Usuarios para poder darse de alta en algún período en el servicio de solicitud de vacaciones.
- CLAVE PRIMARIA: + (identificador de usuario, período).
- CLAVE FORÁNEA: # (identificador de usuario). Con la clave foránea dotamos de integridad referencial a nuestro modelo de datos, de esta forma, un usuario no se podrá borrar del registro de usuarios si aparece en el registro de alta usuarios.

OBJETO DE INFRMACIÓN: REGISTRO DE SOLICITUDES

Atributos:

- autoId: identificador único de cada solicitud
- identificador de usuario
- período al que pertenece la solicitud
- fecha en la que se realiza o tramita una solicitud
- fecha de inicio de las vacaciones en la solicitud (incluido)
- fecha de finalización de vacaciones en la solicitud (incluido)
- estado de proceso de una solicitud
- detalles: descripción u observaciones de la solicitud

Restricciones:

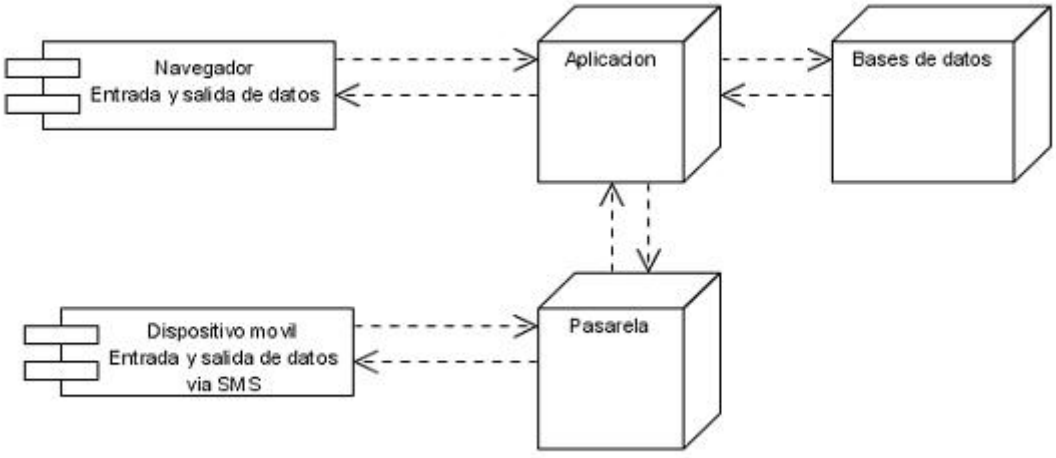
- El campo autoId define única e inequívocamente una solicitud.
- La fecha de inicio debe ser anterior o igual a la fecha de finalización.
- Estado de una solicitud tendrá valores 0, 1 ó 2 según los estados de “sin tramitar”, “solicitud aceptada” o “solicitud rechazada o cancelada”.
- Un usuario debe estar dado de alta en el servicio para el período solicitado para poder realizar una solicitud en este período.
- CLAVE PRIMARIA: + (identificador de usuario, fecha de inicio, fecha de finalización).
- CLAVE FORÁNEA: # (identificador de usuario, período). Con la clave foránea dotamos de integridad referencial a nuestro modelo de datos, de esta forma, un usuario no se podrá borrar del registro de alta usuarios para un período si existen solicitudes de este usuario para el período en concreto. El campo período en el Registro de Solicitudes tiene esta única finalidad, ya que la información que proporciona está implícita en los campos de fecha de inicio y fecha de finalización.

Además, con la combinación de estos registros, se podrá consultar el Perfil de Vacaciones de un Usuario en un Período. Este perfil tomará datos del Registro de Alta de Usuarios y del Registro de Solicitudes de Vacaciones, mostrando los días aún pendientes de vacaciones para ese período y usuario.

6.4 ARQUITECTURA TECNOLÓGICA

6.4.1 Acceso a datos

Los datos del sistema se almacenarán en una base de datos. El acceso a estos datos y la introducción de éstos se realizará a través de la aplicación. La entrada de datos y salida se realizará principalmente a través del navegador, por peticiones HTTP. También se podrán introducir o consultar datos a través del móvil por SMS, para ello será necesario una pasarela que transforme los SMS en peticiones HTTP (y viceversa) que serán enviadas a nuestra aplicación.



F. 6-10: Aplicación ejemplo. Arquitectura tecnológica completa.

6.4.2 Identificación de las necesidades de infraestructura

Los usuarios accederán al sistema desde cualquier navegador a través de Internet o de la Intranet. También podrán acceder vía móvil por SMS.

La comunicación de los navegadores con la aplicación se realizará con peticiones HTTP. En un PC se instalará un servidor que recibirá las peticiones y las redireccionará a la aplicación. La aplicación se encontrará en el servidor.

Para poder acceder desde SMS será necesaria una pasarela de SMS que reciba los mensajes cortos y los transforme en peticiones HTTP. Estas peticiones se tratarán como las recibidas desde cualquier navegador.

La aplicación accederá a una base de datos.

Se trabajará sobré un PC con sistema operativo Windows XP Professional Edition SP2 donde se situarán el servidor de aplicaciones, la pasarela de SMS y la base de datos. Para esta aplicación ejemplo no vamos a utilizar inicialmente un sistema distribuido, sin embargo debe considerarse la posible futura migración a sistemas de este tipo.

6.4.3 Selección de la arquitectura

El conjunto de la plataforma escogida y la infraestructura es la arquitectura tecnológica. Utilizaremos la plataforma J2EE en base al estudio realizado en capítulos anteriores.

Dentro del estándar J2EE existen varias posibilidades de diseño. Se trabajará con una arquitectura multicapa de servidor intermedio, arquitectura de tres capas. Esta es la arquitectura más utilizada en el diseño de aplicaciones Web. Los clientes acceden al servidor y éste es el que accede a los datos.

Los datos se almacenarán en una base de datos relacional. Para el acceso a datos podemos plantearnos varias alternativas:

a) JDBC. Esta opción es adecuada si no hay mayoría de casos de uso transaccionales y los clientes son sólo clientes Web, no stand-alone. Para tener independencia de la base de datos debemos implementar programáticamente una capa de acceso a datos, sin embargo, ya que los casos de uso de nuestra aplicación son sencillos, el diseñar una capa DAO (de acceso a datos) no supondrá gran esfuerzo, y permitirá en un futuro una fácil migración a otro tipo de almacenamiento y acceso a datos. Esta opción es adecuada para nuestro diseño.

b) J2EE Connector Arquitecture. Una alternativa a la conexión con JDBC es la arquitectura de conector J2EE. Esta alternativa actualmente se utiliza principalmente en conexiones a EIS, sistemas de información empresariales. Para nuestra aplicación, en la que los sistemas de información consisten en una base de datos, la opción de JDBC representa una alternativa más madura y fácil.

c) EJB. Esta opción sería la más adecuada si existiesen clientes stand-alone, carga alta o un importante número de casos de usos transaccionales. Nuestra aplicación, en principio no cumple ninguno de estos tres requisitos, así que nos podemos intentar buscar una configuración más simple y sencilla. Sólo si utilizáramos EJB sería necesario la utilización de un contenedor EJB para el manejo de componentes Ejes. Sin embargo, es importante realizar un diseño que nos permite migrar a EJB fácilmente si en un futuro se diera alguna de las tres condiciones comentadas. Esta alternativa podría utilizarse tanto con JDBC o con J2EE Connector Arquitecture.

d) RMI. El uso de RMI con JDBC sería la opción más adecuada si con clientes stand-alone y carga pequeña en un entorno distribuido. Nuestra aplicación, en su primer desarrollo, no trabajará en entorno distribuido, así que no utilizaremos RMI.

Utilizaremos pues, la API JDBC. Este API es una interfaz abstracta que nos proporcionará la conectividad a la base de datos. Para utilizar esta API para el acceso a datos, tendremos que utilizar el drive adecuado para la base de datos utilizada. Idealmente, si cambiásemos de base de datos, bastaría con sustituir el drive y nuestro diseño seguiría siendo aplicable.

La información de configuración debe estar accesible vía JNDI, que crea un contexto válido mientras dure la sesión de usuario.

Se podrá utilizar un framework para facilitar el diseño de nuestra aplicación. Un framework proporciona funcionalidades comunes de la aplicación como manejo de peticiones, invocación de métodos del modelo, y selección y composición de las vistas. Para el desarrollo de la aplicación se podrán extender, usar o implementar las clases e interfaces del framework, lo que hace más fácil el uso de tecnologías Web.

Para el manejo de peticiones y la generación de la vista utilizaremos las API de Servlets y JSP. Para ello necesitaremos utilizar un contenedor Web. El contenedor podrá trabajar con un servidor de aplicaciones o stand-alone. Puesto que para nuestra aplicación sólo manejaremos páginas JSP o HTML no será necesario el servidor, el contenedor Web realizará las funciones de servidor pero además permitirá el uso de Servlets y JSP.

También se podrán utilizar tanto etiquetas estandarizadas como personalizadas para evitar el uso de scripting en las páginas JSP.

J2EE es independiente del sistema operativo, por tanto puede utilizarse sobre Windows XP Professional Edition, que es el sistema operativo disponible inicialmente. Para el desarrollo de la aplicación se utilizará el mismo PC.

La pasarela de SMS, aunque puede considerarse cliente de nuestra aplicación, se situará en el mismo PC que el servidor (o contenedor) Web y la base de datos.

6.4.4 Implementación

Necesitaremos:

- Entorno de desarrollo.
- Servidor J2EE o Implementación J2EE de un contenedor de servlets y JSP.
- Implementaciones de un parser XML (los ficheros de configuración de contenedores utilizan este lenguaje) y de un servicio de nombres y directorios.
- Un framework.
- Implementación J2EE de etiquetas estándares.
- Una base de datos relacional y el drive adecuado.
- Pasarela de SMS.

Como entorno de desarrollo se utilizará NetBeans, un entorno de desarrollo para Java bajo licencia de Sun (SPL, Sun Public License). Esta licencia no es compatible con GPL, pero es una licencia de Software Libre.

El contenedor de servlets y JSP que se utilizará será el Tomcat, con licencia ASL (Apache Software License). No esta muy claro si esta licencia es totalmente compatible con GPL, pero si se trata de una licencia de Software Libre. Aunque se podrá usar de forma independiente, en las etapas de desarrollo y pruebas utilizaremos la versión de Tomcat incluida en el entorno de desarrollo NetBeans.

Tanto para NetBeans, como para Tomcat, es necesaria una implementación de la plataforma Java. Utilizaremos J2SE de Sun MicroSystems que incluye los API JAXP, JDBC y JNDI, parser XML, controlador de acceso a datos y servicio de nombres y directorios respectivamente. Sun Microsystems distribuye de forma gratuita el J2SE, sin embargo su licencia no es totalmente

compatible con el Software Libre ya que aunque el código fuente está disponible su modificación y redistribución están sometidas a ciertos requisitos. Se trata de la licencia Sun Microsystems, Inc. Binary Code License Agreement y Puesto que nosotros sólo lo utilizamos, sin modificación, ni redistribución, las restricciones no nos afectan.

Para el diseño Web utilizaremos el framework de Struts y las etiquetas Standard Taglibs, implementación de JSTL. Tanto Struts como Standard Taglibs se distribuyen bajo la licencia de Apache ASL.

La base de datos que se utilizará será MySQL. MySQL está disponible bajo un modelo de licencia dual, se puede usar bajo la licencia de Software Libre GPL o bajo una licencia comercial. Utilizaremos el drive MySQL Connector/J, drive de MySQL para aplicaciones Java, también disponible bajo licencia GPL.

Para la pasarela de SMS utilizaremos Kannel, bajo licencia Kannel Software License compatible con GPL. Kannel está escrito en C, para poder utilizarlo en Windows será necesario instalar Cygwin, entorno Linux en Windows. Cygwin se distribuye bajo licencia de Software Libre GNU GPL.

Para más información sobre el tipo de licencias consultar el apartado 2.5 del capítulo de INTRODUCCIÓN y 10.1 del capítulo de ANEXOS.

Ahora es necesario especificar las versiones que utilizaremos de cada una de estas tecnologías y herramientas. Es importante utilizar versiones compatibles entre ellas, por lo que la selección de versiones no es algo trivial.

- NetBeans IDE 4.0.
- Tomcat 5.0.28 (incluido en NetBeans).
- J2SE 5.0. Incluye JAXP 1.3, JDBC 3.0 e implementaciones de JNDI.
- Struts 1.2.8.
- JSTL 1.1.2.
- MySQL Database Server 4.1.9.

- Connector/J 3.1.6, drive de MySQL para aplicaciones Java.
- Kannel 1.4.0, pasarela de SMS.

Es importante destacar que nuestra aplicación la desarrollaremos en un entorno J2EE 1.4, que incluye las especificaciones de Servlets 2.4 y de JSP 2.0. Esta decisión supone el uso de una versión de Tomcat 5.x. Además, al usar JSP 2.0 es recomendable utilizar una implementación de etiquetas estándar JSTL 1.1 y no anterior. La única restricción que estas decisiones suponen a la hora de elegir una versión de J2SE, es que esta sea igual o superior a J2SE 1.4, por lo que es válido el uso de J2SE 1.5 (hace referencia al mismo producto que J2SE 5.0, tan sólo es un cambio de nomenclatura).

Se incluye más información sobre la elección y compatibilidad de versiones en el capítulo de ANEXOS, apartado 10.3 VERSIONES.

6.4.5 Patrones de diseño.

Además de una arquitectura física multicapa, en la implementación de nuestro software utilizaremos un patrón arquitectónico layer (de capas). Nuestro software estará divido en capas, de forma que cambios en una capa, idealmente, no afectará a capas superiores o inferiores.

El primer patrón que utilizaremos lo proporciona la especificación de Servlets desde su versión 2.3. La especificación implementa el patrón Intercepting Filter incluyendo filtros o cadena de filtros de un modo configurable en el fichero `web.xml`. Estos filtros realizarán una discriminación previa de las peticiones que le llegará al controlador.

Al utilizar Struts, implícitamente se utiliza el patrón Front Controller (controlador), junto a una serie de Helpers (ayudantes).

Los principales Helpers que Struts ofrece son Request Processor Helper (procesado de peticiones), Perform Action Helper (ejecución de acción) y View Helper (vista). El Request Precessor Helper.

El Front Controller, el Request Processor Helper y el Perform Action Helper lo implementan el Front Controller, el Request Processor y las Actions de Struts. Se pueden utilizar las clases que Struts proporciona o bien extender estas.

El patrón View Helper en Struts lo implementan los Action Forms (formularios de Struts) y las etiquetas que Struts proporciona. Junto a Struts, también se utilizarán otras etiquetas como las JSTL y Custom Tags (etiquetas personalizadas) diseñadas por nosotros. JSP proporciona también etiquetas que implementarían el View Helper.

Para la composición de la vista se propone el patrón Composite View (compositor de vista) que Struts implementa con los métodos de plantilla Template o Tiles. Nosotros utilizaremos Tiles. También utilizaremos etiquetas JSP para la implementación de este patrón.

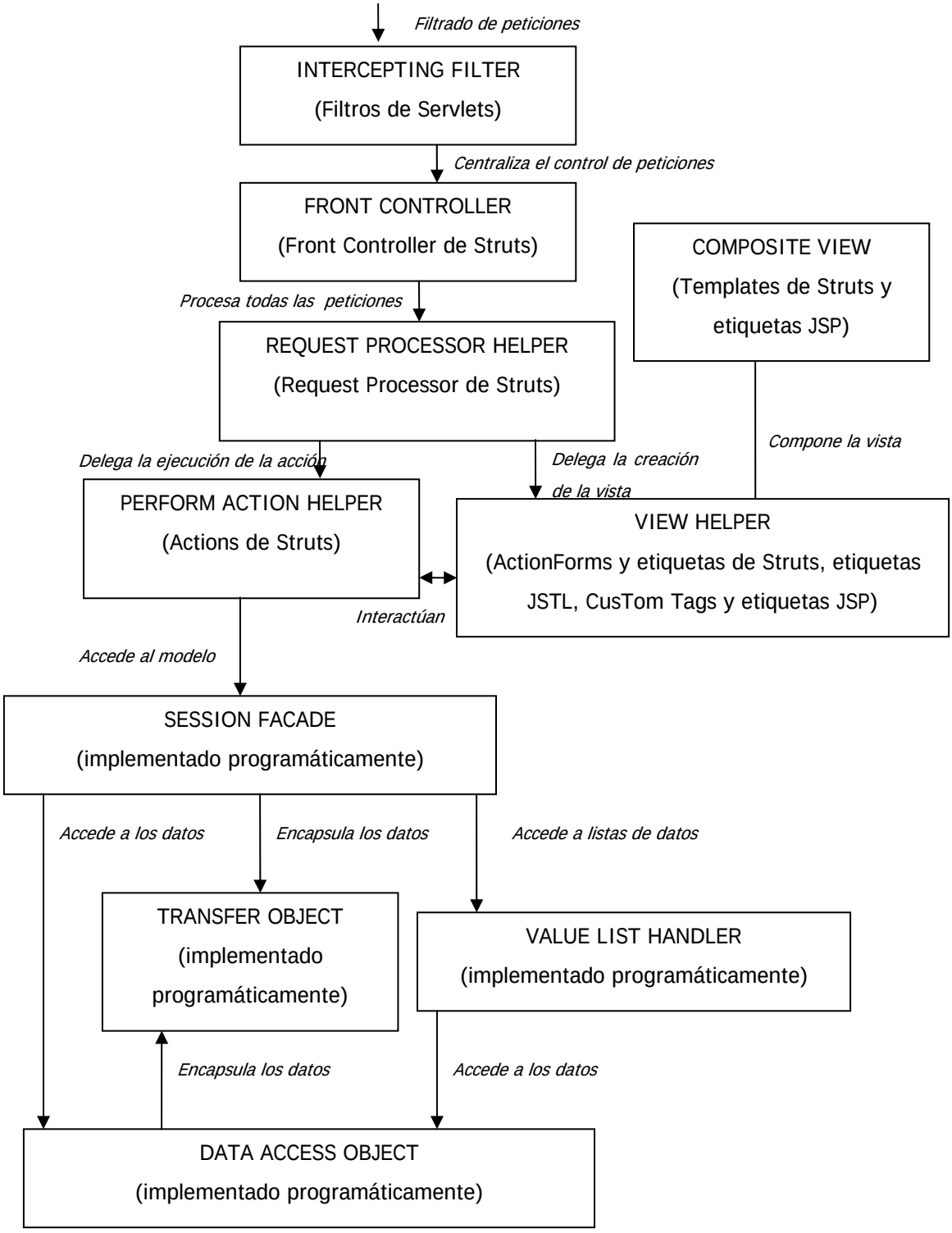
Programáticamente implementaremos algunos patrones utilizados en el modelo. El patrón Session Facade (fachada de sesión) nos servirá como interfaz de acceso al modelo.

Para el acceso a datos utilizaremos, también programáticamente, los patrones Transfer Object (objeto de transferencia), Data Access Object (objeto de acceso a datos) y Value List Handler (manejador de lista de valores).

Con los Data Access Object accederemos a la base de datos, los Tranfer Objects o Value Objects guardarán los valares de los datos capturados. Cuando obtengamos de la base de datos una lista de objetos utilizaremos el patrón Value List Handler.

Un esquema general de los patrones de Diseño J2EE utilizados se muestra en la página siguiente. Junto a estos patrones se pueden utilizar patrones generales de diseño, como por ejemplo Factories (factorías) para cada uno de estos patrones. Para mayor información sobre

patrones de diseño consultar el capítulo 5 DISEÑO DE APLICACIONES CON J2EE, apartado 5.6 PATRONES DE DISEÑO.



F.6-11: Aplicación ejemplo. Patrones de Diseño J2EE utilizados.

6.5 DISEÑO E IMPLEMENTACIÓN

Con los requisitos de la aplicación bien definidos, el diseño se hace relativamente fácil.

Comentar que el lenguaje que se utilizará en el diseño es el inglés por lo que se intentará dejar claro las equivalencias entre el modelo extraído de los requisitos, en el que se ha utilizado el español, con el diseño que se expone a continuación.

6.5.1 Estructura general de la aplicación

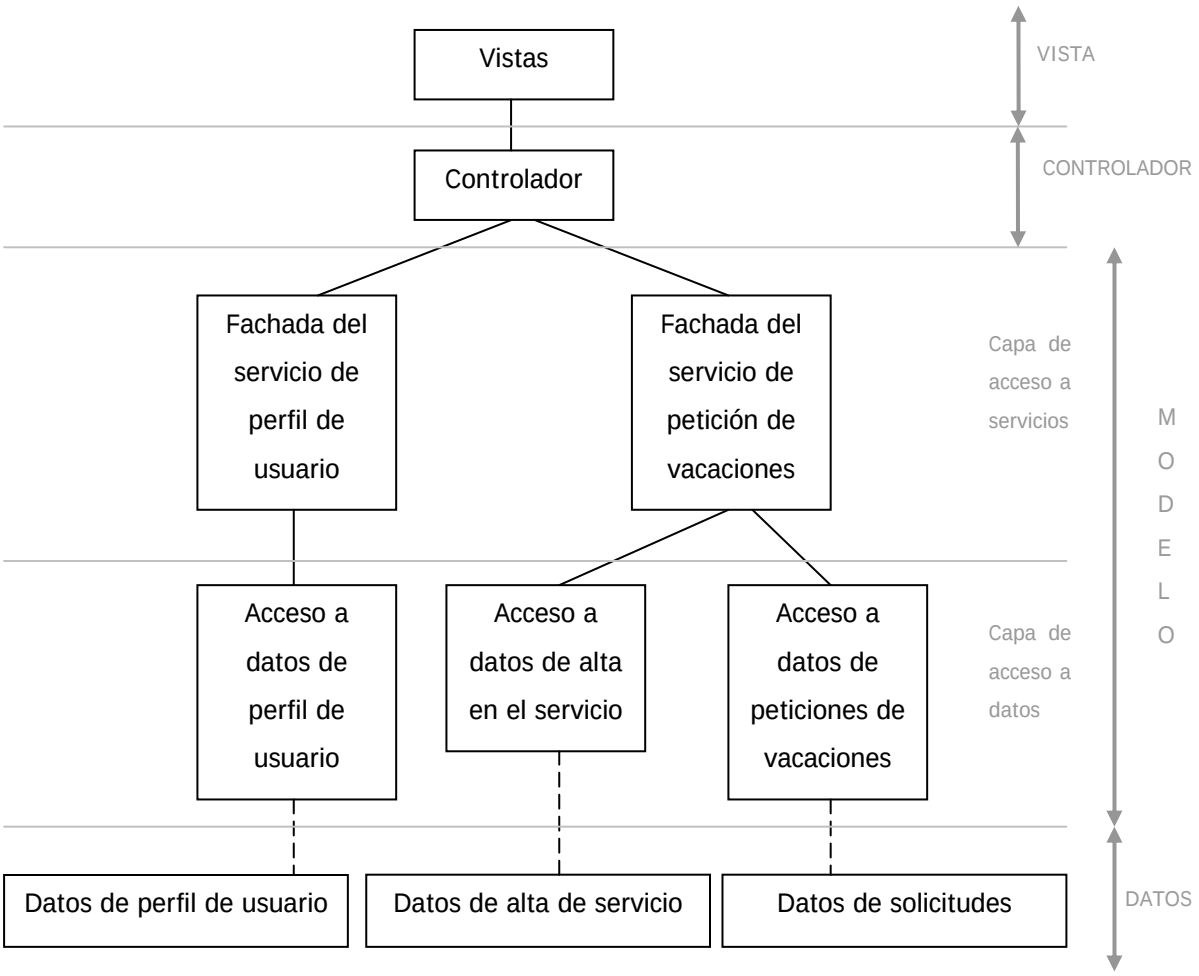
El diseño de la aplicación se ha realizado teniendo en cuenta los dos servicios principales que pretendemos ofrecer, el servicio de perfil de usuario y el servicio de solicitud de vacaciones.

En las capas inferiores del modelo, las de acceso a datos, el servicio de solicitud de vacaciones se ha divido en las operaciones de alta en el servicio, y por otra parte en las solicitudes realizadas. Se pueden considerar tres ramas en las capas de acceso a datos: perfil de usuario, alta de servicio de vacaciones, y petición de vacaciones.

Estos servicios proporcionan una fachada (o facade) para la interacción del usuario con los servicios. En esta capa de fachada, encontramos tan sólo dos puntos de acceso, un acceso a los servicios que proporciona el perfil de usuario, y otro acceso para el servicio de vacaciones completo, tanto para operaciones de alta de servicio, peticiones y operaciones comunes.

La capa de controlador proporciona una capa común de acceso a ambos servicios. En esta capa se manejarán, además, los objetos de sesión que fueran necesario.

Un esquema muy general de la aplicación se presenta a continuación. En apartados posteriores se detallarán cada una de las capas.

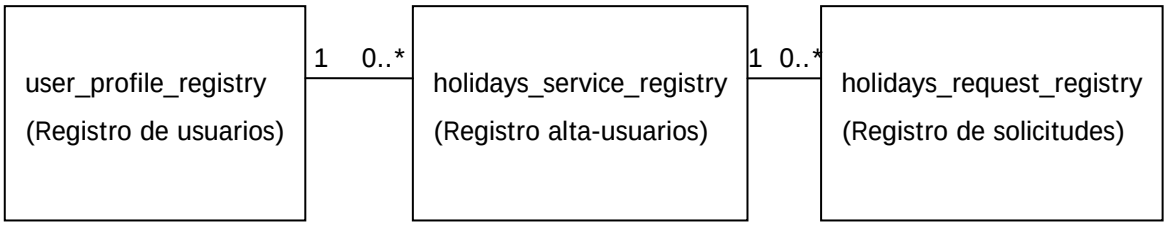


F. 6-12: Modelo de capas de la aplicación.

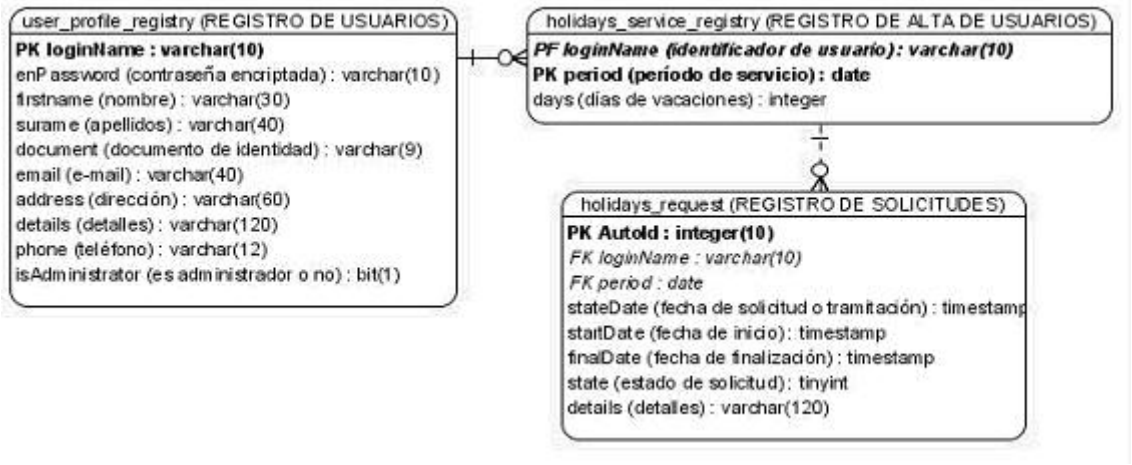
6.5.2 Estructura de datos

La estructura de datos es una equivalencia directa al modelo de datos definido en los requisitos. Los registros definidos como Registro de Usuarios, Registro de Alta-Usuario y Registro de

Solicitudes se traducen en tablas cuyas columnas se identifican con los atributos definidos para cada uno de estos registros. Tan sólo queda especificar el tipo de dato utilizado para cada atributo.



F.6-13: Estructura de tablas de datos de la Aplicación Ejemplo.



F.6-14: Estructura de tablas de datos de la Aplicación Ejemplo. Atributos.

Estas tablas cumplen los requisitos impuestos previamente. Los atributos con PK forman parte de la clave primaria, los atributos con FK forman parte de una clave foránea, y por último, los atributos con PF pertenecen a la clave primaria y a la foránea.

En la tabla `user_profile_registry` los campos `loginName`, `enPassword` e `isAdministrator` no pueden ser nulos. `isAdministrator` por defecto tiene valor 0 (no es administrador) al insertar una nueva fila en la tabla correspondiente a un nuevo perfil de usuario. En esta tabla casi todos los campas son de tipo VARCHAR, con longitudes distintas adecuadas a las características del campo. `isAdministrator` será de tipo BOOLEAN, ya que sólo aceptará valores 0 o 1 según carezca o posea permisos de administrador.

La tabla `holidays_service_registry` contiene los perfiles de alta de un usuario para un período de servicio. El campo `loginName` debe corresponderse con el campo `loginName` de `user_profile_registry` y por tanto es del mismo tipo de dato. El atributo `period`, representa el periodo por el que el usuario es dado de alta en el servicio y debe corresponderse con un año. El tipo de dato será por tanto tipo DATE, en concreto tipo YEAR. El campo `days` podrá tener un valor máximo de 366 días por lo que un tipo INTEGER es adecuado. Ninguno de estos campos podrán ser nulos, y `days` por defecto tomará valor 0.

La tabla `holidays_request_registry` no podrá contener solicitudes de usuarios no dados de alta en el período para el que se realiza la solicitud. Los campos `stateDate`, `startDate` y `finalDate` son de tipo TIMESTAMP para operar más fácilmente con ellos. El atributo `state` por defecto tendrá valor 0, es decir, toda nueva solicitud tiene estado de no procesada. El campo `stateDate`, representará la fecha en la que se realizó la solicitud o bien, si esta ya ha sido tramitada, la fecha de tramitación.

6.5.2.1 Estructura de datos en MySQL

Para crear nuestras tablas, previamente nos creamos una base de datos que llamaremos `apps` donde guardaremos todos nuestros datos.

Elegimos tablas tipo InnoDB que permite trabajar con transacciones en MySQL.

Al definir las claves foráneas imponemos que al intentar borrar o actualizar campos que atenten contra la integridad referencial de nuestra base de datos, no realice la acción y muestre un aviso.

En resumen, nuestra base de datos nos permitirá trabajar con transacciones y no permitirá operaciones que atenten contra la integridad de nuestros datos.

En los ANEXOS, en el apartado 10.6 CÓDIGO, se incluyen los scripts de creación de nuestra base de datos y las tablas necesarias. En el apartado 10.4 INSTALACIÓN Y CONFIGURACIÓN BÁSICA DE HERRAMIENTAS y 10.5 INSTALACIÓN Y CONFIGURACIÓN DE LA APLICACIÓN, se explica detalladamente la instalación de nuestra versión de MySQL y el procedimiento más adecuado para la creación de nuestra base de datos.

6.5.2.2 Configuración de la fuente de datos para acceder desde nuestra aplicación.

Existen varias formas para la configuración de la fuente de datos, o datasource. La más común es configurarla en el fichero del contenedor Web, `server.xml` en el caso de Tomcat. Tomcat pone a disposición de la aplicación, vía JNDI, las datasources configuradas.

En este fichero de Tomcat se configuran las aplicaciones, que sirve el contenedor. Entre los datos configurables nombraremos el path con el que se accederaá a la aplicación, ficheros de log, y los datasources.

Estos parámetros se configuran dentro la etiqueta `<context>` y `</context>`. Tomcat, sin embargo, permite configurar estos parámetros, en un fichero de nuestra aplicación ya que en muchos casos los desarrolladores no tendrán acceso al fichero `server.xml`.

Nosotros utilizaremos un fichero `context.xml`, dentro de la carpeta `META-INF` de nuestra aplicación. Se recomienda estudiar las posibilidades que cada contenedor proporciona para esta labor.

En `context.xml` configuramos las datasources. Esto implica dar valor a los parámetros necesarios para acceder a la base de datos. Es conveniente ver la documentación de la base de datos utilizada para conocer los parámetros necesarios y sus posibles valores.

Con la etiqueta `<ResourceParams>`, y su atributo `name` damos un nombre lógico a la fuente de datos, el cual se utilizará para acceder a ella desde nuestra aplicación. Puesto que accederemos vía JNDI y utilizaremos JDBC normas de nombrado recomiendan comenzar el nombre por `jdbc/`. Será necesario dar valores a la url de conexión con la base de datos, el nombre de la clase del drive, el usuario con el que se accederá a la base de datos desde la aplicación y la contraseña.

Tomcat nos permite utilizar un pool de conexiones, esto es, puede realizar un determinado número de conexiones que maneja y controla si están activas o no, lo que permite reutilizar conexiones abandonadas permitiendo una mayor eficiencia en las conexiones para no sobrecargar la base de datos. Además, podrán configurarse otros elementos referentes a las conexiones como conexiones activas, cuando se borrará una conexión abandonada, tiempo máximo de espera para una conexión antes de darse por fallida, etc. El pool de conexiones se configura con el parámetro `factory`, y Tomcat ofrece la librería `org.apache.commons.dbcp.BasicDataSourceFactory`.

El ejemplo siguiente muestra la configuración de la fuente de datos para nuestra aplicación con MySQL y Tomcat de los valores más comunes. Este código se encontrará entre etiquetas `<context></context>`, ya sea en el fichero de configuración del contenedor (`server.xml`) o directamente en `context.xml`.

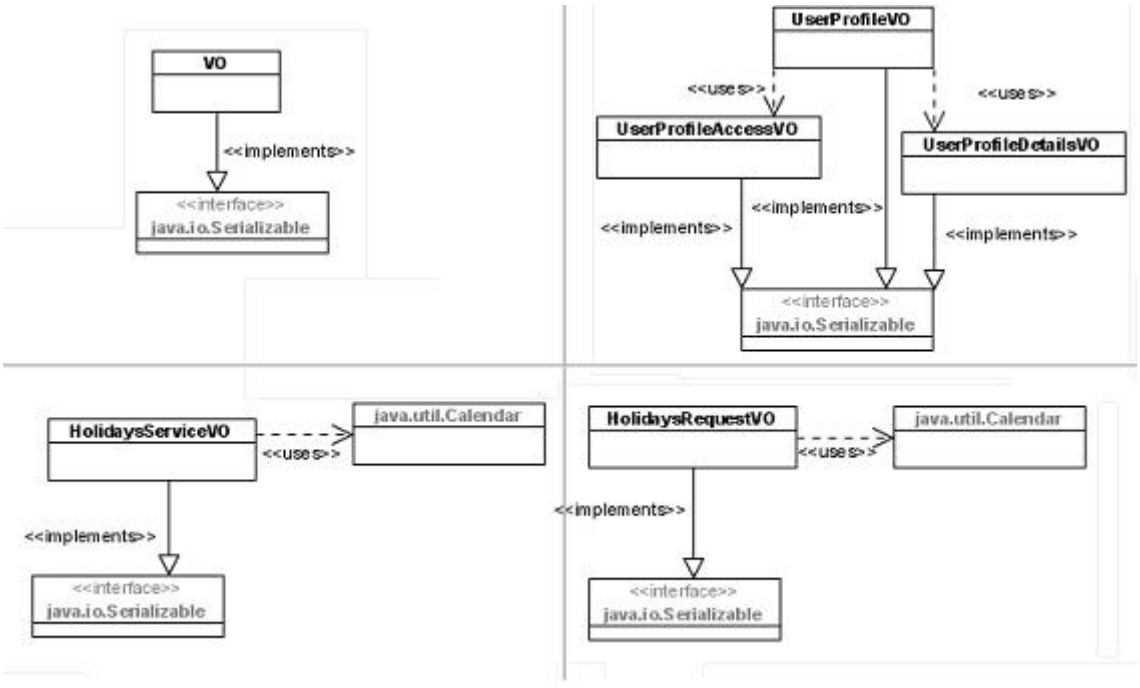
```
<!-- Configuración de las fuentes de datos para acceso vía JNDI. -->
<!-- Nombre JNDI para acceder desde la aplicación. -->
<ResourceParams name="jdbc/ApplicationDataSource">
<!-- Url para la conexión con MySQL, por defecto, al utilizar drive
jdbc, “jdbc:mysql://localhost:3306/nombre_base_datos_creada”. -->
<parameter>
<name>url</name>
<value>jdbc:mysql://localhost:3306/apps</value>
</parameter>
<!-- Nombre de la clase del drive, será necesario ver la documentación
del drive utilizado en nuestro caso utilizamos Connector/J 3.1.6,
drive de MySQL para aplicaciones Java. -->
<parameter>
<name>driverClassName</name>
<value>com.mysql.jdbc.Driver</value>
</parameter>
<!-- Usuario y password para acceder a la base de datos desde la
aplicación. En MySQL se configurará un usuario con este username y
password con permisos para insertar, actualizar y consultar datos. -->
<parameter>
<name>username</name>
<value>userapps</value>
</parameter>
<parameter>
<name>password</name>
<value>userapps</value>
</parameter>
<!-- Configuración del pool de conexiones que ofrece Tomcat. -->
<parameter>
<name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>
<!-- A continuación se configurarían el resto de parámetros del pool
de conexiones como máximo número de conexiones activas con
“maxActive”, máximo número de conexiones inactivas con “maxIdle”,
máximo tiempo de espera para una conexión antes de considerarla
fallida con “maxWait”, etc. -->
</ResourceParams>
```

Bloque de código 6-1 Configuración de datasources.

6.5.3 Objetos de datos

Implementaremos el patrón *Transfer Objec*t (también llamado *Value Object*), VO, por cada uno de los objetos de datos que utilizaremos en la aplicación.

En nuestro caso crearemos las clases `UserProfileVO`, `HolidaysServiceVO` y `HolidaysRequestVO`. Estas clases son JavaBeans, es decir, básicamente están formadas por una serie de atributos privados, y métodos públicos `get/set` para acceder o modificar estos atributos e implementa a la interfaz `serializable`.



F. 6-15: Objetos de datos. Patrón Value Object (VO).

Los JavaBeans deben cumplir unas reglas de nomenclatura. Los métodos `get` y `set` se llamarán `getAtributo`, y `setAtributo`, donde `atributo` es el nombre de cada atributo de la clase.

Los atributos corresponderán con los valores de cada fila de cada una de las tablas.

En el caso de los perfiles de usuarios, dividiremos `UserProfileVO` en los atributos `loginName`, `enPassword` y `phone`, y los subconjuntos `UserProfileAccessVO` y `UserProfileDetailsVO`. De este modo clasificamos las propiedades del perfil de usuario en varias categorías. Las principales, los datos de acceso y detalles. En datos de acceso, `UserProfileAccessVO` incluimos el atributo `isAdministrator`, en los detalles, `UserProfileDetailsVO`, el resto de atributos. Tanto `UserProfileAccessVO` como `UserProfileDetailsVO` son clases JavaBeans.

Esta clasificación en el perfil de usuario nos facilicitará el acceso en operaciones de actualización de datos. Las operaciones de actualización se realizarán sobre la contraseña, sobre el teléfono, sobre los detalles del perfil, o en los datos de acceso al sistema.

Para las fechas en general utilizaremos el tipo de datos `Calendar`. Este tipo es `Serializable` y por tanto se podría transportar en redes distribuidas. En el caso de `HolidaysServiceVO`, el atributo `period`, será de este tipo. Sin embargo, se incluirá un método que permitirá calcular el año de esta fecha tipo `Calendar`, ya que en la base de datos sólo se guarda este valor y este método podrá ser útil en las operaciones relacionadas con actualizaciones y búsquedas.

Igualmente, en `HolidaysRequestVO`, se añaden métodos para obtener el día, el mes y el año, de las fechas involucradas.

Los anteriores Value Objects son del tipo Domain Value Object, ya que sus atributos coinciden con los atributos de una de las tablas del sistema (o proceden indirectamente de ellas). Más adelante se crearán los Custom Value Objects que serán útiles para la interacción del usuario con el sistema.

```
//EJEMPLO ESTRUCTURA TRANSFER OBJECT
//paquete
//librerías y APIs a importar
import java.io.Serializable;
/**
 * Esta clase puede utilizarse para interactuar con los datos. Los
 * atributos se corresponden con variables privadas. Se utilizan los
 * métodos públicos get y set para acceder a estos.
 */
public class TransferObjectVO implements Serializable {

    /** Atributos. */
    private String attribute;

    /** Constructor. Inicializa el objeto. */
    public TransferObjectVO(String attribute) {
        this.attribute = attribute;
    }
    /** Obtiene valor actual del atributo. */
    public String getAttribute()  {
        return attribute;
    }

    /** Da un nuevo valor al atributo. */
    public void setAttribute(String attribute)  {
        this.attribute = attribute;
    }

    /** Proporciona una cadena representación del objeto VO. */
    public String toString(){
        return new String("attribute = " + attribute);
    }
}
```

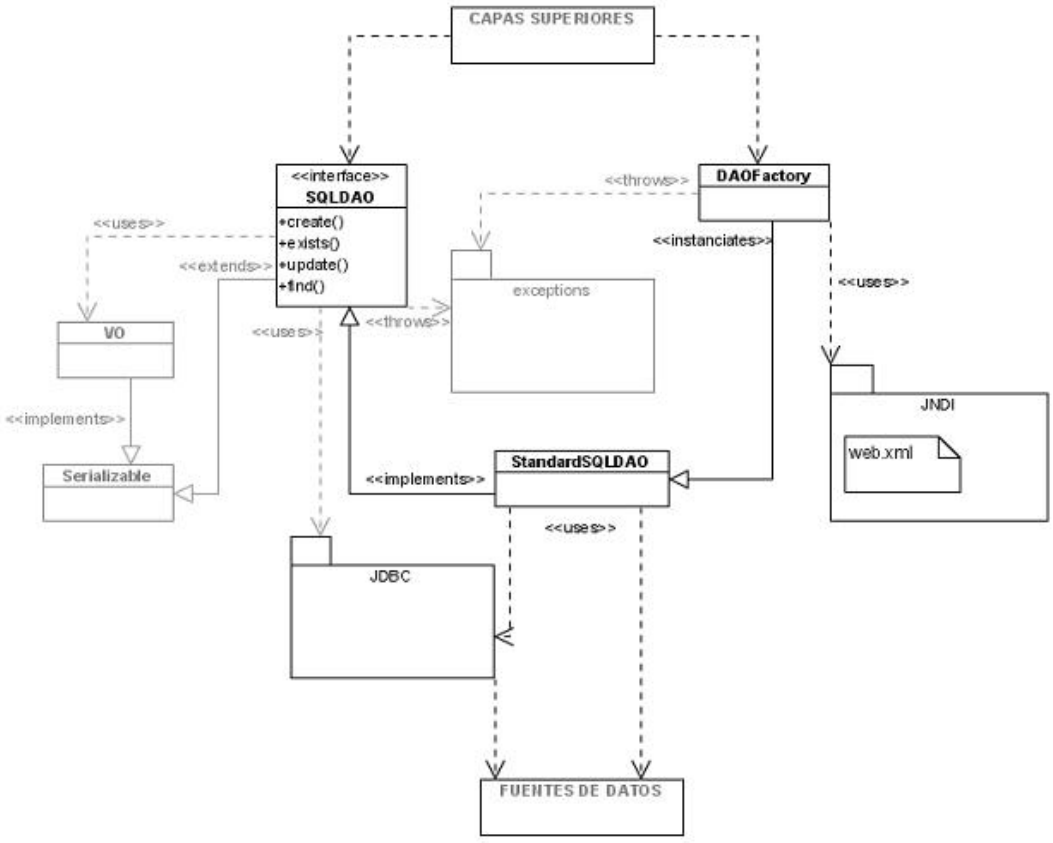
Bloque de código 6-2: Ejemplo estructura Transfer Object (VO).

6.5.4 Lógica de negocio

6.5.4.1 Capa de acceso a datos

Para la capa de acceso a datos implementamos el patrón de diseño Data Access Object, DAO. Este patrón proporcionará a nuestra aplicación mayor independencia de las fuentes de datos.

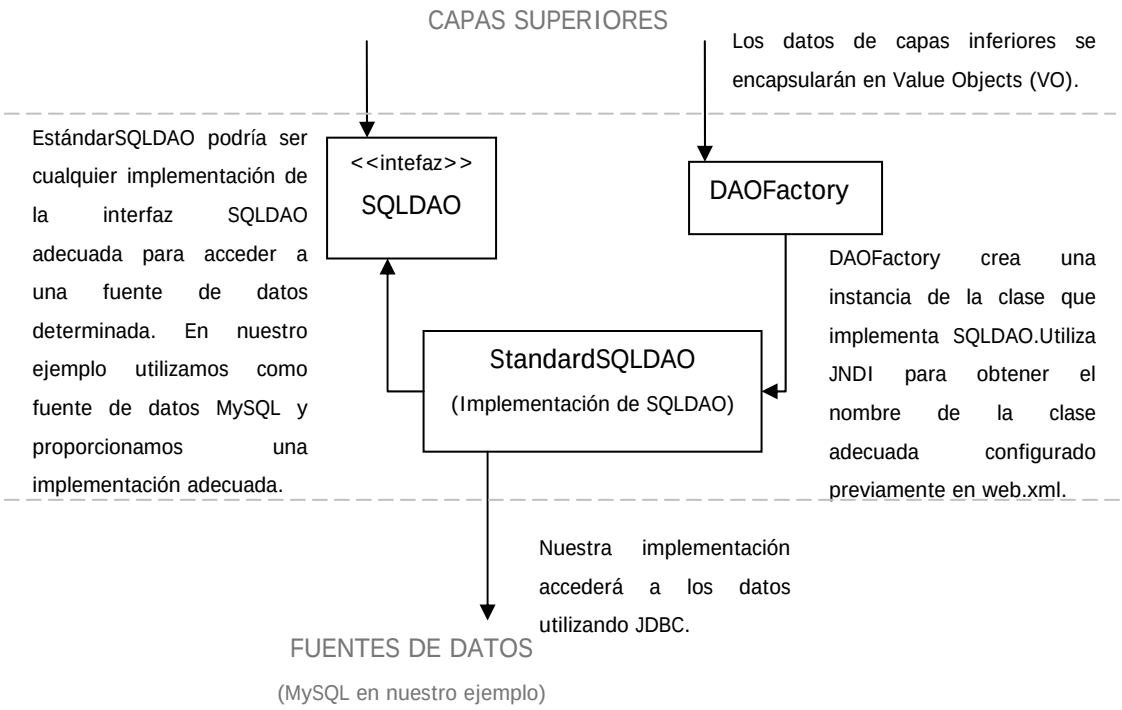
Comenzamos definiendo una interfaz que definirá las operaciones que se podrán realizar sobre los datos. Con el patrón factory, visto en el apartado 5.6.11.1 Factorías, podremos configurar en el fichero `web.xml` la implementación adecuada de esta interfaz según la fuente de datos.



F. 6-16: Implementación del patrón DAO con factorías.

La figura F. 6-16: Implementación del patrón DAO con factorías., muestra el diagrama generalizado de las capas de acceso a datos de la aplicación. Para nuestra aplicación se utilizarán diagramas similares para los accesos a los datos referentes al perfil de datos, a usuarios dados de alta en el servicio de vacaciones y a las peticiones de vacaciones utilizadas.

Este patrón nos proporcionará independencia de las fuentes de datos. En la figura F. 6-17: Relación del patrón DAO con capas superiores e inferiores., nos muestra como las capas superiores no dependen de una implementación concreta para una determinada base de datos.



F. 6-17: Relación del patrón DAO con capas superiores e inferiores.

Se creará una interfaz para cada uno de los accesos a los distintos objetos de datos. Estas interfaces definirán las operaciones básicas de interactuación con la base de datos: crear nueva fila de datos, comprobar si existe un dato y actualizar datos. Nuestra aplicación no contemplará la eliminación de datos. Junto a estas operaciones básicas se definirán un conjunto de posibles operaciones de búsquedas para cada una de las interfaces. Crearemos las interfaces: `SQLUserProfileDAO`, `SQLHolidaysServiceDAO` y `SQLHolidaysRequestDAO`.

A continuación se crearán las implementaciones necesarias de estas interfaces. Para nuestro ejemplo, crearemos una implementación válida para el acceso a datos en MySQL con JDBC, para cada uno de los objetos de datos. Como se ha comentado en capítulos anteriores, debido a que no todas las bases de datos utilizan el mismo "dialecto" SQL, es posible que una implementación para una base de datos relacional, no sea adecuada para otra. Crearemos las implementaciones: `StandardSQLUserProfileDAO`, `StandardSQLHolidaysServiceDAO` y `StandardSQLHolidaysRequestDAO`.

Para una mayor reutilización del código se crearán algunas clases útiles en diversas situciones incluidas en `application.util.*`.

Para el manejo y control de errores creamos un paquete que contendrá las excepciones que pueden lanzar las clases de nuestra aplicación. Este paquete será `application.util.exceptions` y en esta capa de acceso a datos podremos lanzar las excepciones `DuplicateInstanceException`, `InstanceNotFoundException` e `InternalErrorException`. La primera comprende los errores de intentar registrar un usuario, servicio o solicitud ya existente en la fuente de datos. La segunda, es lanzada cuando en la búsqueda de un elemento éste no existe. Por último, en esta capa de acceso a datos, `InternalErrorException` se utilizará para encapsular errores internos de la fuente de datos como insertar más de una vez un nuevo elemento, no actualizar datos, error en las sentencias SQL, etc.

También se han creado utilidades para acceder al contexto JNDI en `application.util.jndi`. La clase `ConfigurationParametersManager` es utiliza en esta capa, por las clases Factory para crear una instancia de la implementación adecuada de la

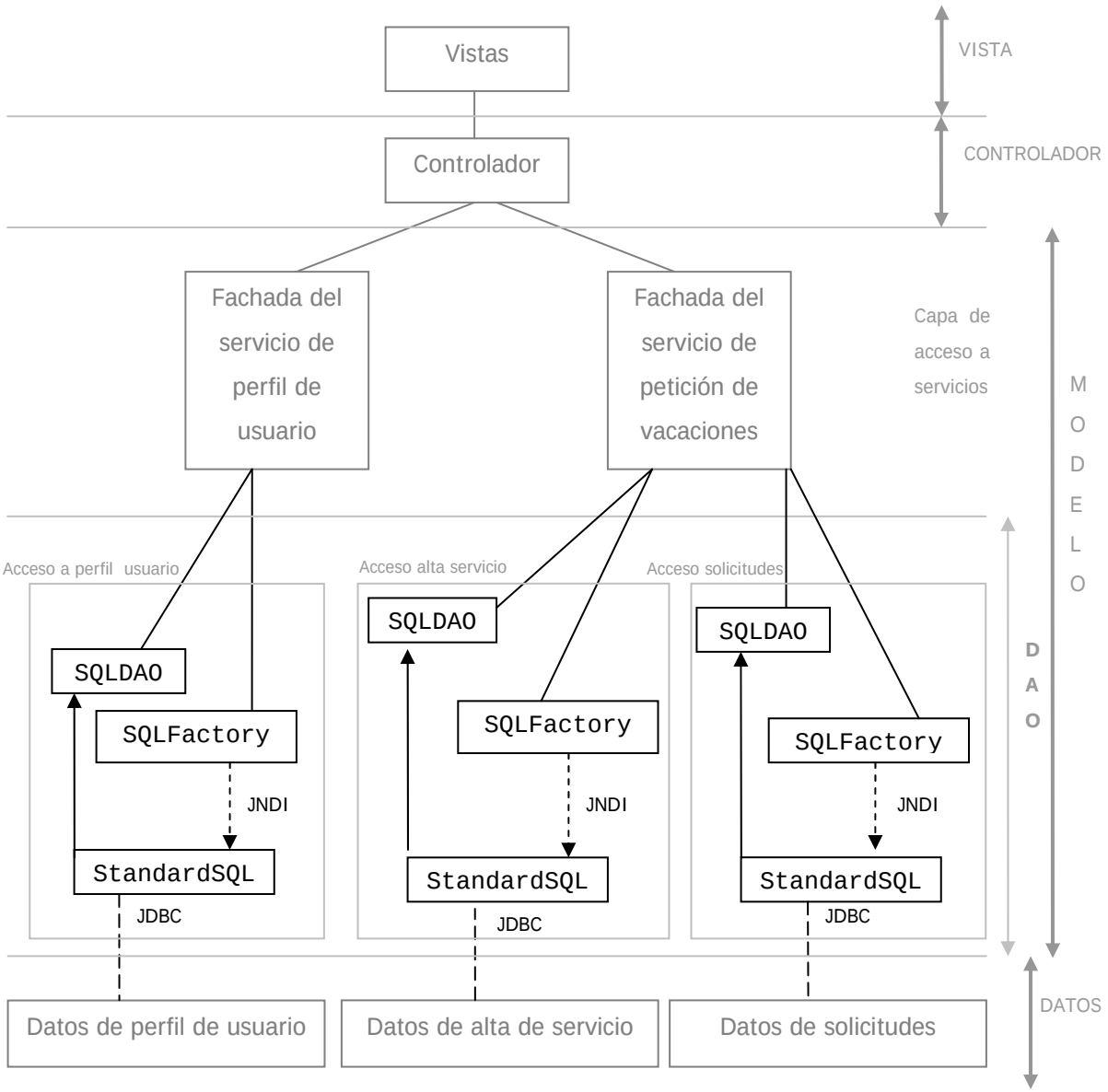
interfaz DAO. Si un parámetro no es localizado en el contexto JNDI, lanza una excepción `MissingConfigurationParameterException`.

Como se ha comentado anteriormente, al utilizar las clases Factory JNDI, es necesario especificar en el fichero `web.xml`, las implementaciones correspondientes a las interfaces de acceso a datos. En Bloque de código 6-5: Configuración de la capa de fachadas a servicios (façade) utilizando factorías., se incluye parte del fichero `web.xml` de la aplicación donde se la configuración de las implementaciones DAO con etiquetas `<env-entry>`.

```
<!-- Configuración de la capa DAO en web.xml para hecer la
implementación adecuada accesible vía JNDI. -->


<web-apps>
<!-- Configuración de servlets, filtros, taglibs,... -->
<!-- Configuración DAO perfil de usuario. -->
<env-entry>
<env-entry-name>SQLUserProfileDAOFactory/daoClassName
      </env-entry-name>
<env-entry-value>
application.model.userprofileservice.dao.StandardSQLUserProfileDAO
      </env-entry-value>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<!—Configuración DAO alta de usuarios en el servicio de solicitud de
vacaciones -->
<env-entry>
<env-entry-name>SQLHolidaysServiceDAOFactory/daoClassName
      </env-entry-name>
<env-entry-value>
application.model.holidaysrequestservice.dao.holidaysservice.
StandardSQLHolidaysServiceDAO
      </env-entry-value>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<!—Configuración DAO solicitudes de vacaciones. -->
<env-entry>
<env-entry-name>SQLHolidaysRequestDAOFactory/daoClassName
      </env-entry-name>
<env-entry-value>
application.model.holidaysrequestservice.dao.holidaysrequest.
StandardSQLHolidaysRequestDAO
      </env-entry-value>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>
<!-- Resto de parámetros de configuración. -->

</web-app>
```

Bloque de código 6-3: Configuración de la capa de acceso a datos (DAO) utilizando factorías.

Para finalizar con la capa de acceso a datos, incluimos el esquema de capas de nuestra aplicación en el que incluimos más la capa de acceso a datos. Para mayor información sobre la implementación de la capa de acceso, consultar el anexo que incluye el código de la aplicación y los diagramas de relaciones de clases.

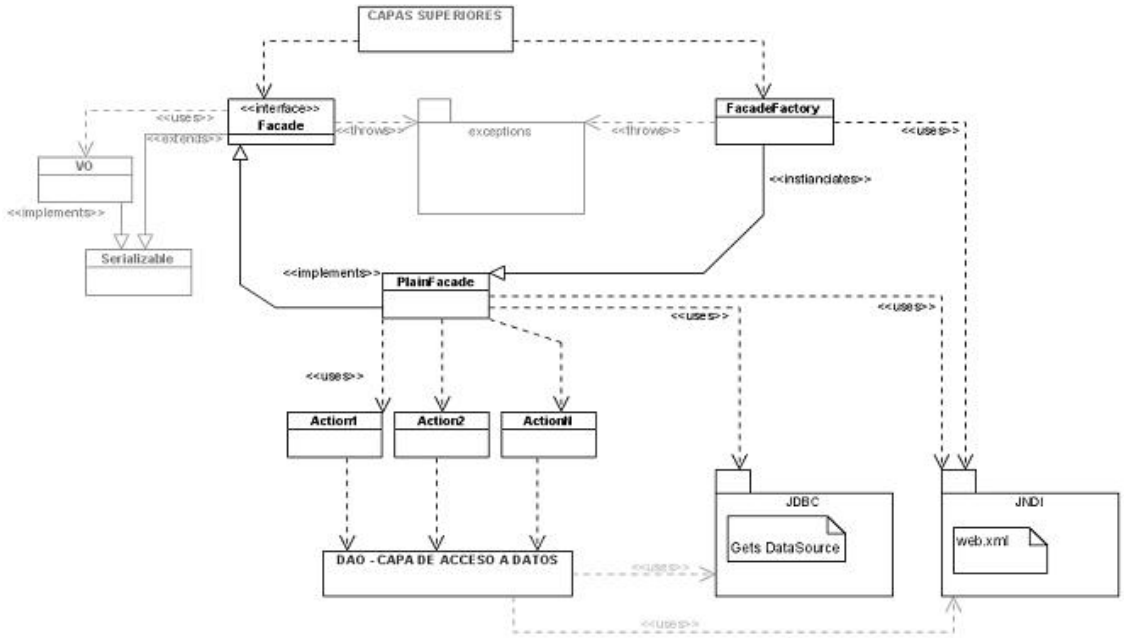


F. 6-18: Capa de acceso a datos.

6.5.4.2 Fachadas de acceso a servicios

Una vez implementada la capa de acceso, pasamos al diseño de las fachadas de los servicios. Las fachadas encapsularán la lógica de negocio de la capa de acceso a datos y proporcionará un único punto de acceso al servicio desde las capas superiores.

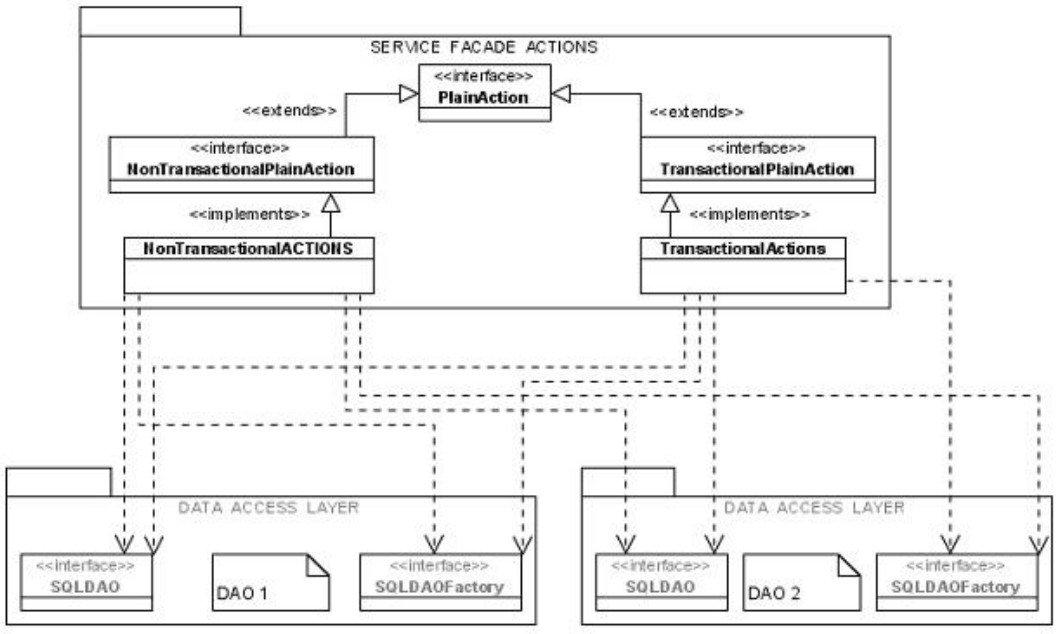
Al igual que en la capa inferior, utilizaremos factorías. La implementación de cada fachada será configurable en `web.xml` y una clase del tipo `Factory` podrá instanciar la clase adecuada una vez que vía JNDI lee el valor del parámetro correspondiente a cada fachada.



F. 6-19: Implementación de las fachadas de los servicios de la aplicación (Façade).

En la capa de acceso a datos se implementaron las operaciones básicas sobre los datos, inserción, actualización y búsquedas. Es en esta capa donde implementamos acciones más complejas y donde controlamos el carácter transaccional o no de cada acción. Recordamos que EJB maneja transacciones de un modo transparente, sin embargo, puesto que para nuestra aplicación se decidió trabajar directamente con JDBC y no EJB, este manejo de transacciones deberá implementarse programáticamente.

En el paquete `application.util.sql` se proporcionan interfaces que nos ayudarán en esta tarea. En la implementación de las capas fachadas que aportamos, todas las clases descenderán indirectamente de la interfaz `PlainAction`. Se han creado otras dos interfaces, que extienden a esta interfaz básica: las interfaces `NonTransactionalAction` y `TransaccionalAction`. Todas las acciones de esta capa serán implementación de una de estas dos interfaces.



F. 6-20: Acciones en la fachada a servicios. Modelado del caracter transaccional de una acción.

Hasta ahora no se ha modelado el carácter transaccional, tan sólo se han divido las acciones en dos grupos. Es la clase `PlainActionProcessor`, también en el paquete `application.util.sql`, la que aporta las diferencias entre estos dos grupos de acciones.

`PlainActionProcessor` aporta el método process para cada uno de estos grupos de acciones. La sobrecarga de este método nos permite diferenciar entre las clases transaccionales y las no transaccionales. Esta clase recibe como argumento una `DataSource` y uno de los dos tipos de `PlainAction`. La interfaz `DataSource` nos permite trabajar con la fuente de datos, `PlainActionProcessor` busca la fuente de datos (configurada en `context.xml`) y abre una conexión. Recordamos que el manejo del pool de conexiones lo maneja directamente Tomcat, al configurarlo en `context.xml` con el parámetro `factory`.

Una vez encontrada la fuente de datos y abierta una conexión, la clase `PlainActionProcessor` es la encargada de activar o desactivar el modo AUTO COMMIT, realizar rollbacks, y todas las operaciones relacionadas con un transacción. Esto lo realiza a través de los métodos `process` que proporciona, diferenciandose si la llamada incluye acciones transaccionales o no.

```
    /**
     * Ejecuta una acción no transaccional sobre una conexión.
     */
    public final static Object process(DataSource dataSource,
            NonTransactionalPlainAction action) throws ModelException,
            InternalErrorException {

        Connection connection = null;

        try {

            /* Por defecto la conexión se encuentra en modo AUTO
             * COMMIT, es decir, en modo no transaccional. Cada
             * operación que se realice se realiza individualmente
             * independientemente de las otras operaciones. */
            connection = dataSource.getConnection();
            return action.execute(connection);

        } catch (SQLException e) {
            throw new InternalErrorException(e);
        } finally {
```

```
            GeneralOperations.closeConnection(connection);
        }
    }


    /**
     * Ejecuta una acción transaccional sobre una conexión con el
     * nivel de isolación en "TRANSACCIÓN SERIALIZABLE". Si la acción
     * lanza alguna excepción, la transacción no se ejecuta (rolled
     * back).
     */
    public final static Object process(DataSource dataSource,
            TransactionalPlainAction action) throws ModelException,
            InternalErrorException {

        Connection connection = null;
        boolean commited = false;

        try {

            /* El modo AUTO COMMIT se desactiva (false) y el nivel de
             * isolación se pasa a TRANSACTION SERIALIZABLE, todas las
             * aciones que se realicen se tratarán como una unidad
             * atómica, si una de ellas no puede llevarse a cabo, se
             * deshacen todas las acciones realizadas (roll back) y se
             * vuelve al estado inicial. */
            connection = dataSource.getConnection();
            connection.setAutoCommit(false);
            connection.setTransactionIsolation(
                    Connection.TRANSACTION_SERIALIZABLE);

            /* Ejecuta la acción. */
            Object result = action.execute(connection);

            /* Ejecuta la transacción. */
            connection.commit();
            commited = true;

            /* Retorna el resultado */
            return result;

        } catch(SQLException e) {
            throw new InternalErrorException(e);
        } finally {
            try {
                if (connection != null) {
                    if (!commited) {
                        connection.rollback();
                    }
                    connection.close();
                }
```

```
            } catch (SQLException e) {
                throw new InternalErrorException(e);
            }
        }
    }
}
```

Bloque de código 6-4: Acciones transaccionales y no transaccionales con PlainActionProcessor.

Una vez implementadas las acciones, extendiendo las interfaces `NonTransactionalAction` o `TransactionalAction`, según el carácter transaccional de cada una de ellas, es la clase `PlainxxxFacade` la que proporciona una implementación de la fachada del servicio. Esta clase implementa la interfaz `xxxFacade` y será instanciada por `xxxFacadeFactory` una vez configurado en el fichero `web.xml`, con etiquetas `<env-entry>`, el parámetro `xxxFacadeFactory/facadeClassName`.

Como puede observarse en las figuras, hemos creados un único punto de acceso a la capa de fachada del servicio desde capas superiores, y sólo las clases `Actions`, accederán a la capa de acceso a dato por sus puntos de acceso, `SQLxxxDAO` y `SQLxxxDAOFactory`.

La fachada de servicio de perfil de usuario ofrece la posibilidad de realizar las operaciones de identificación de usuario, registro de nuevos usuarios, actualización del perfil, cambio de contraseña, cambio de teléfono y búsquedas. Un usuario será el usuario activo tras una operación de identificación de usuario o cuando el administrador se identifique como este otro usuario.

El paquete `application.model.userprofileservice.facade.delegate` proporciona la clase `PlainUserProfileServiceFacade` que implementa a la interfaz `UserProfileServiceFacade`. Las acciones transaccionales son las que incluyen operación de escritura en la base de datos, es decir, registro y actualizaciones. Las operaciones no transaccionales son las de identificación de usuario y búsquedas. Todas las acciones de perfil de usuario se encuentran en el paquete `application.model.userprofileservice.facade.actions`.

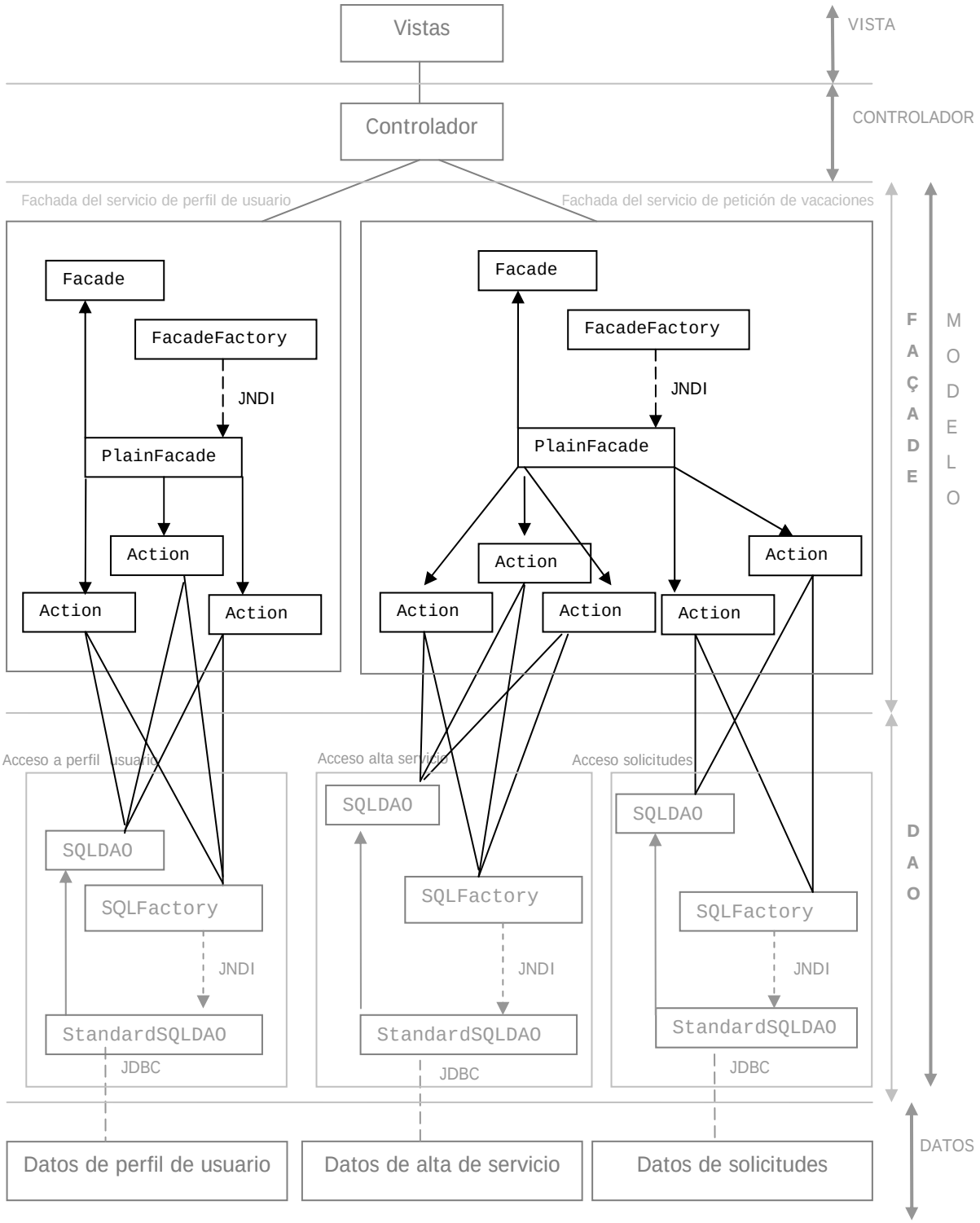
Además se crean dos nuevas excepciones: `IncorrectPasswordException` y `AdministratorRequiredException`. La primera es lanzanda en las operaciones de login y cambio de password, cambio de datos de teléfonos y cuando el administrador se identifica como otro usuario cuando no se introduce la contraseña correcta. La segunda es lanzada cuando un usuario no administrador intenta realizar una acción sólo permitida al administrador.

Por último, crearemos un objeto de datos llamado `UserFacadeVO`. Este objeto puede ser util en capas superiores tras la operación de identificador de usuario, ya que nos proporciona un objeto con el identificador de usuario y sus permisos del sistema, en concreto si es administrador o no.

La fachada de servicio de solicitud de vacaciones ofrece la posibilidad de realizar las operaciones de registro de un nuevo servicio de solicitud, registro de nuevas peticiones, actualización del servicio de vacaciones, actualización del estado de una solicititud y gran variedad de consultas y búsquedas. Esta fachada utilizará la fachada de servicio de perfil de usuario para inicializarse con el mismo usuario.

El paquete `application.model.holidaysrequestservice.facade.delegate` proporciona la clase `PlainHolidaysRequestServiceFacade` que implementa a la interfaz `HolidaysRequestServiceFacade`. Al igual que en el servicio de perfil de usuario, las acciones transaccionales son las que incluyen operación de escritura en la base de datos, es decir, registro y actualizaciones. Las operaciones no transaccionales comprenderán las búsquedas. A diferencia del servicio de solicitud de vacaciones, este servicio muestra dos grupos de acciones, las que se realizan contra la base de datos de alta en el servicio, y las que se realizan contra la base de datos de solicitudes de vacaiones. En la estructura de paquetes se encuentran respectivamente en `application.model.holidaysrequestservice.facade.actions.holidaysservice` y `application.model.holidaysrequestservice.facade.holidaysrequest`.

Por último, crearemos los objetos de datos `HolidaysProfileVO` y `HolidaysVO`. Estos objetos serán útiles en capas superiores.



F. 6-21: Capa de fachadas de servicios

Para finalizar la capa de fachadas, será necesario la configuración en el fichero `web.xml` de los parámetros `UserProfileServiceFacadeFactory/facadeClassName` y `HolidaysRequestServiceFacadeFactory/facadeClassName` para que las clases factorías `UserProfileServiceFacadeFactory` y `HolidaysRequestServiceFacadeFactory` puedan crear una instancia de la clase adecuada.

```
<!-- Configuración de la capa DAO en web.xml para
hacer la implementación adecuada accesible vía JNDI. -->


<web-apps>
<!-- Configuración de servlets, filtros, taglibs,... -->
<!-- Configuración FAÇADE servicio perfil de usuario. -->
         <env-entry>
         <env-entry-name>
         UserProfileServiceFacadeFactory/facadeClassName
             </env-entry-name>
         <env-entry-value>application.model.userprofileservice.
             facade.delegate.PlainUserProfileServiceFacade
         </env-entry-value>
         <env-entry-type>java.lang.String</env-entry-type>
         </env-entry>

<!-- Configuración FAÇADE servicio solicitud de vacaciones. -->
         <env-entry>
         <env-entry-name>
         HolidaysRequestServiceFacadeFactory/facadeClassName
             </env-entry-name>
         <env-entry-value>application.model.holidaysrequestservice.
             facade.delegate.
         PlainHolidaysRequestServiceFacade
         </env-entry-value>
         <env-entry-type>java.lang.String</env-entry-type>
         </env-entry>

</web-apps>
```

Bloque de código 6-5: Configuración de la capa de fachadas a servicios (façade) utilizando factorías.

6.5.5 Controlador

La mayor parte del controlador de nuestra aplicación estará implementada por el framework de Struts. Aunque aún tendremos que crear algunas clases y configurar el controlador adecuadamente, con los ficheros de configuración de struts y en el descriptor de despliegue.

6.5.5.1 Session Façade

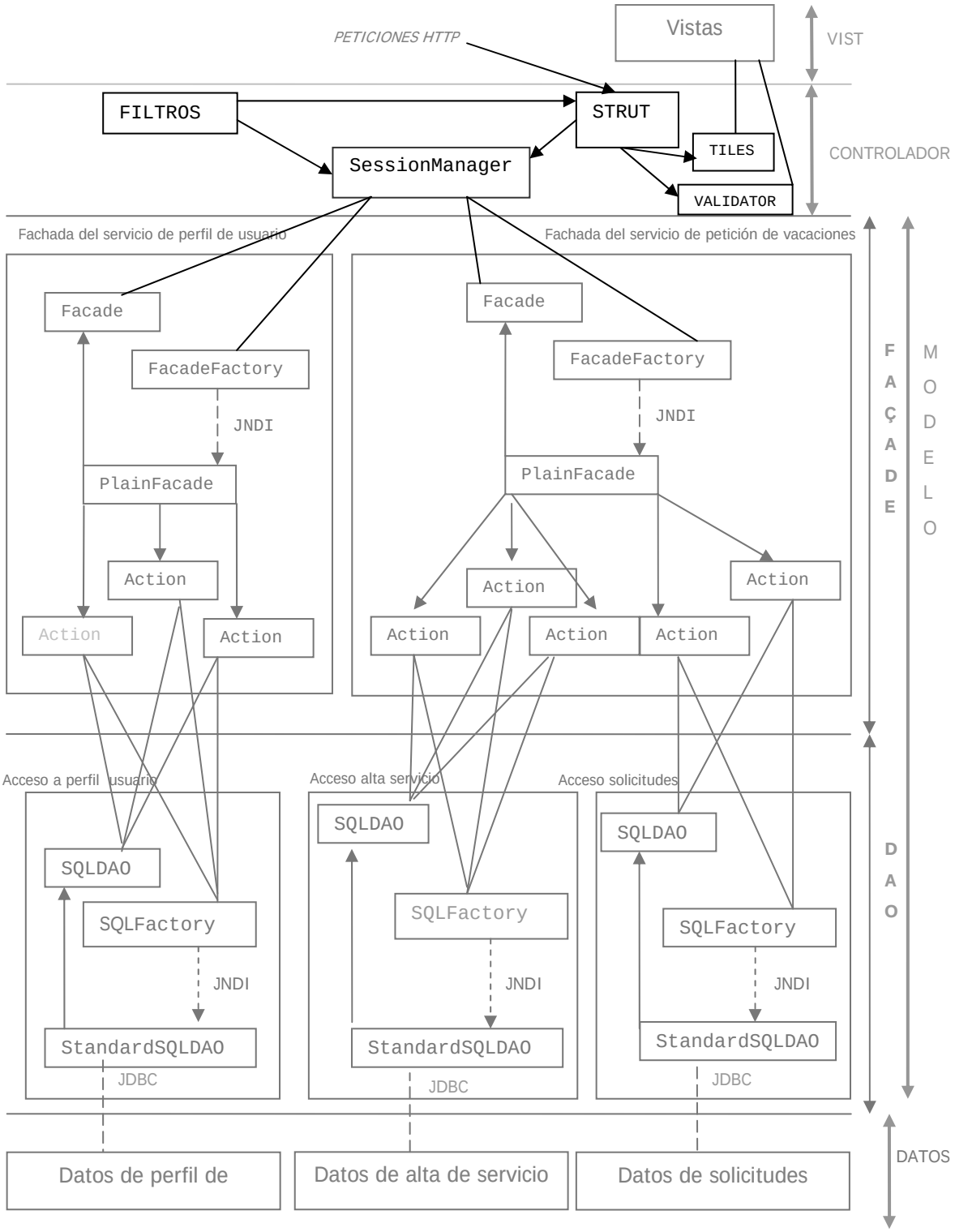
Implementaremos este patrón programáticamente. Las acciones que implementarán la interfaz que proporciona Struts, en algunos casos tendrán que acceder al modelo, ya sea para ejecutar alguna acción de inserción o actualización de datos, o para obtener resultados tras una consulta. También, al realizar operaciones de filtrado podría ser necesario el acceso al modelo.

Como vimos en la capa modelo de nuestra aplicación creamos dos fachadas, cada una implementando uno de los servicios que nuestra aplicación proporciona. Este patrón proporciona un acceso único al modelo manejando además datos de sesión para el acceso a nuestros servicios desde un navegador Web.

Esta será la primera capa de acceso al modelo desde la aplicación en una aplicación puramente Web. Si trabajáramos con sistemas distribuidos el patrón Bussines Delegate encapsularía a nuestra Session Façcade proporcionando acceso a métodos remotos.

Implementamos el patrón en la clase `SessionManager` del paquete `application.http.controller.session`. Esta clase se centrará en el manejo de objectos de sesión como identificador de usuario, si el usuario es administrador, usuario cargado en la sessión, fachadas del usuario a los servicios, etc.

Junto al manejo de los objetos de sesión, ofrece métodos para realizar el filtrado según un usuario esté identificado o no, sea o no administrador, y otros métodos para realizar las distintas operaciones de perfil de usuario y de solicitud de vacaciones.



F. 6-22: Controlador. SessionManager.

6.5.5.2 Struts – Front Controller

Como se comentó en el capítulo 5 DISEÑO DE APLICACIONES CON J2EE, lo adecuado es utilizar la estrategia de servlet como controlador. Para utilizar Struts es necesario configurar la aplicación para que todas las peticiones sean recibidas por el controlador de Struts. Este controlador es el servlet `ActionServlet` o bien una extensión de éste. Esta configuración se realiza en el descriptor de despliegue web.xml.

```
<!-- Configuración del controlador. -->
<servlet>
<servlet-name>action</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet
   </servlet-class>
<init-param>
<param-name>config</param-name>
<param-value>/WEB-INF/Struts/struts-config.xml</param-value>
</init-param>
</servlet>

<!-- Mapeo de URLs -->
<servlet-mapping>
<servlet-name>action</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Bloque de código 6-6: Configuración del controlador de Struts en web.xml.

Como se muestra en el bloque de código anterior, la configuración del controlador se realiza en dos partes. La primera es definir el servlet controlador y darle un nombre lógico. Además `ActionServlet` lee el parámetro `config` para buscar el fichero de configuración de Struts. La segunda parte en la configuración es el mapeo de las URL que se le pasarán al controlador, este mapeo se realiza con el nombre lógico especificado anteriormente y un patrón de las URL.

Lo correcto sería que todas las URL generadas, los hipervínculos y las acciones asociadas a los formularios sigan el patrón especificado, es decir, sigan el patrón *.do, de este modo todas las

peticiones serán recibidas por el controlador. Intentaremos no crear directamente direcciones JSP, en vez de ellos se realizaría una petición a un tipo de acción (por supuesto que cumpla con el patrón *.do) y está sería la que redireccionaría a la página JSP.

Si en vez de utilizar el servlet controlador por defecto de Struts, se utiliza una extensión, puede incluirse en el controlador algún tipo de procesado extra. En el controlador suelen incluirse operaciones relacionadas con los parámetros enviados, cabeceras de la petición, etc.

Una vez que el controlador recibe la petición, la delega a la clase `RequestProcessor`. Esta clase implementa el patrón Request Processor Helper, un manejador de peticiones que se dedica al procesado de éstas. Al igual que con el controlador, se puede utilizar la clase que Struts proporciona por defecto, `RequestProcessor`, o bien extender esta clase. A veces interesa extender esta clase para incluir algún tipo de procesado o filtros.

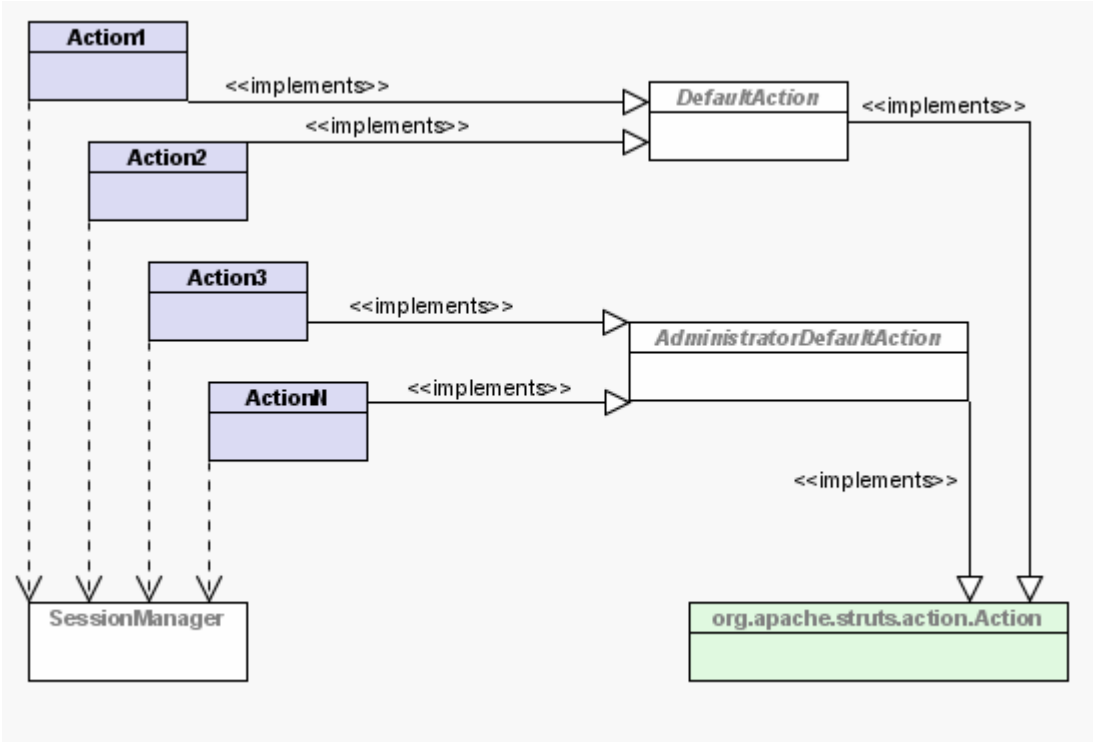
```
<struts-config>
<!--  ... -->
<!—Configuración del controlador. -->
<controller processorClass=
      "org.apache.struts.action.RequestProcessor"/>
<!-- ... -->
</struts-config>
```

Bloque de código 6-7: Configuración del RequestProcessor en struts-config.xml.

Struts utiliza el fichero de configuración `struts-config.xml` para la definición de acciones y formularios. Cuando el `RequestProcessor` recibe una petición comprueba que la URI se corresponde con algunas de las acciones definidas en el fichero. Comprueba si la acción en concreto tiene asociado un formulario y si requiere validación. Luego crea una instancia, si no existe previamente, de la clase que implementa la acción y la ejecuta. Una vez completada la acción ésta redirecciona a otra acción o a una vista para que sea enviada al usuario.

Existen dos tipos de acciones en Struts. Las que tan sólo son un redireccionamiento de la petición a otra acción o a una vista, `org.apache.struts.actions.ForwardAction`, y las que extienden a la clase `Action` de Struts. Los formularios definidos serán una extensión de la clase `ActionForm`.

Para nuestra aplicación utilizaremos dos tipos de acciones base, `DefaultAction` y `AdministratorDefaultAction`. Estas acciones base extienden a la interfaz `Action` que Struts proporciona. El resto de acciones de nuestra aplicación implementarán el método `doExecute` de una de estas acciones. Aquellas acciones que requieran administrador implementarán `AdministratorDefaultAction`, que lanza una excepción de `AdministratorRequiredException` en caso de error, y el resto de acciones implementarán `DefaultAction`.



F. 6-23:Implementación de Actions de Struts.

Es posible que la clase que necesitemos crear utilice algún tipo de formulario. Existen varios tipos de formularios en Struts, todos ellos deben implementar a la clase `org.apache.struts.action.ActionForm`. Esta clase proporciona métodos `getter` y `setter` para el manejo de datos del formulario, método reset, y validación.

En la mayoría de los casos no es necesario programar una clase tipo `ActionForm`. Struts ofrece una extensión de esta clase en los formularios `org.apache.strtus.action.DynaActionForm` que nos permite crear los formularios directamente en el fichero de configuración de Struts.

Una vez creada la clase `Action` y el formulario correspondiente se procede a la configuración de la clase en el fichero de configuración de Struts, `struts-config.xml`. Veamos a continuación algunos ejemplos de código.

```
//package
//import

import java.io.IOException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import org.apache.struts.action.DynaActionForm;

public class MyAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm
            form, HttpServletRequest request, HttpServletResponse
```

```
            response) throws IOException, ServletException {

        DynaActionForm myForm =
                (DynaActionForm) form;

        String field1 = (String) myForm.get("field1");

        //Implementación de la lógica de negocio de la acción
        //Returns ActionForward

    }

}
```

Bloque de código 6-8: Action de Struts.

```
<struts-config>

<!-- Definicion de formularios. -->

<form-beans>

<!-- Formulario utilizando una clase que extiende directamente a
ActionForm y que debe implementar los métodos getter y setter de sus
campos, reset y validate. -->
<form-bean name="myForm" type="application.myclasses.myActionForm"/>

<!-- Formulario utilizando la clase DynaActionForm. Definimos los
campos y el formulario es creado automáticamente, no es necesaria la
creación de una clase por el programador. -->
    <form-bean name="myDynaForm"
    type="org.apache.struts.action.DynaActionForm">
        <form-property name="field1" type="java.lang.String"/>
        <form-property name="field2" type="java.lang.String"/>
    </form-bean>

<!-- Resto de formularios de la aplicación. -->
</form-beans>

<!-- Resto de parámetros de configuración. -->
</struts-config>
```

Bloque de código 6-9: Definición de ActionForm.

```
<struts-config>
<!—Definición de formularios y otros. -->
<!-- Mapeo de acciones. -->
<action-mappings>

  <!-- Accion creada por el programador que extiende a ActionForm. -->
    <action path="/MyAction"
    type="application.myclasses.myAction"
    name="myDynaForm"
    scope="request">
        <forward name="myAction" path="/myPage.jsp" redirect="true"/>
    </action>

   <!-- Accion Forward de Struts. Redireccionamiento de la acción. -->
    <action path="/MyRedirectAction"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/myPage.jsp"/>

<!-- Resto de mapeo de acciones. -->
</action-mappings>

<!-- Resto de parámetros de configuración. -->

</struts-config>
```

Bloque de código 6-10: Mapeo de acciones.

Struts tiene también librerías para el manejo de errores, `org.apache.struts.action.ActionMessages`. El mensaje de error es un `org.apache.struts.action.ActionMessage` que dispone de un constructor que permite especificar el identificador del mensaje de error en el fichero `Messages.properties` que veremos con más detalles en el apartado 6.5.6.6 Internacionalización.

En los ejemplos se muestran configuraciones básicas, para más información se recomienda consultar la documentación que Struts proporciona.

6.5.5.2.a Validator – View Helper

El proyecto Jakarta, subproyecto de Apache, nos ofrece también el Commons Validator Framework creado especialmente para la validación de la integridad de los datos de entrada. Aunque puede utilizarse independientemente, desde la versión 1.1 de Struts lo encontramos integrado en su distribución.

Este framework nos permite validar los datos de entrada de los formularios antes de realizar la acción correspondiente. La verificación de la integridad de los datos es una potente arma ante un posible comportamiento indeseado de nuestra aplicación al recibir datos inesperados, tanto si estos datos son introducidos voluntaria o involuntariamente. Por ejemplo, es bastante conocida la técnica de SQL Injection para el ataques a páginas Web. Una validación adecuada puede prevenir exitos en este tipo de ataques.

La configuración de este framework se realiza principalmente en los ficheros de configuración correspondientes. Normalmente las validaciones que proporciona Validator son suficientes, si no fuera así, en casos de validaciones complejas, puede extenderse el framework creando nuevas clases de validación. Otra solución en estos casos, para un formulario en concreto, es crear el formulario como una extensión directa de ActionForm de Struts e implementar la validación en el método validate que esta clase proporciona. Aún así, insistimos en que estos casos sólo se dan cuando son necesarias validaciones muy complejas.

Para su configuración, Validator suele utilizar dos ficheros: `validation.xml` y `validator-rules.xml`. En el primero se definen las reglas de validación de cada campo de cada formulario definido en el fichero de configuración de Struts. El segundo, normalmente no se modifica y se utiliza el que proporciona la distribución por defecto. Este fichero define los tipos de validaciones, que por defecto incluye entre otros: campo obligatorio, número entero, máscaras, validación de fechas, reglas para definir validez según el valor de otros campos del formulario, etc…

Para poder aprovechar estas herramientas de validación, los formularios deben extender `org.apache.struts.validator.action.ValidatorForm` en vez de

`org.apache.struts.action.ActionForm`. De este modo, Validator nos ofrece los formularios del tipo `org.apache.struts.validator.DynaValidatorActionForm` o `org.apache.struts.validator.DynaValidatorForm` que se definen como el resto de formularios `DynaActionForm` vistos en Struts. Existen otros tipos de formularios como `org.apache.struts.validator.LazyValidatorForm`, pero al igual que en el apartado anterior, para más información se recomienda mirar la documentación que Commons Validator proporciona.

Por último, para la correcta integración de los frameworks de Struts y Validator, es necesaria su configuración en el fichero struts-config.xml.

```
<struts-config>

<!-- Definicion de formularios. -->
<form-beans>

<!—Ejemplo formulario DynaValidatorForm. -->
    <form-bean name="myDynaForm"
    type="org.apache.struts.validator.DynaValidatorForm">
        <form-property name="field1" type="java.lang.String"/>
        <form-property name="field2" type="java.lang.String"/>
    </form-bean>

<!-- Resto de formularios de la aplicación. -->
</form-beans>

<!-- Resto de parámetros de configuración. -->

<!-- Plug-in -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
    value="/WEB-INF/Struts/validator-rules.xml,
    /WEB-INF/Struts/validation.xml"/>
</plug-in>


</struts-config>
```

Bloque de código 6-11: Coniguración de Validator con Struts.

```
<form-validation>

<formset>

<!—Validación ejemplo de formulario tipo DynaValidatorForm -->
<form name="loginForm">
    <field property="loginName" depends="required, mask">
    <var><var-name>mask</var-name>
        <var-value>^\w{3,9}$</var-value></var>
    <msg name="mask" key="errorsMessage.mask.login"/>
    <arg0 name="mask" key="userProfile.loginName"/>
    </field>

    <field property="password" depends="required, mask">
    <var><var-name>mask</var-name>
        <var-value>^\w{3,9}$</var-value></var>
    <msg name="mask" key="errorsMessage.mask.login"/>
    <arg0 name="mask" key="userProfile.password"/>
    </field>
</form>

<!-- Validación ejemplo de formulario tipo DynaValidatorActionForm -->
<form name="/Login">
    <field property="loginName" depends="required, mask">
    <var><var-name>mask</var-name>
        <var-value>^\w{3,9}$</var-value></var>
    <msg name="mask" key="errorsMessage.mask.login"/>
    <arg0 name="mask" key="userProfile.loginName"/>
    </field>

    <field property="password" depends="required, mask">
    <var><var-name>mask</var-name>
        <var-value>^\w{3,9}$</var-value></var>
    <msg name="mask" key="errorsMessage.mask.login"/>
    <arg0 name="mask" key="userProfile.password"/>
    </field>
</form>

<!-- Resto de validaciones -->
</formset>
</form-validation>
```

Bloque de código 6-12: Validación de formularios en validation.xml.

```
<form-validation>

   <global>

      <validator name="required"
            classname="org.apache.struts.validator.FieldChecks"
               method="validateRequired"
         methodParams="java.lang.Object,
                       org.apache.commons.validator.ValidatorAction,
                       org.apache.commons.validator.Field,
                       org.apache.struts.action.ActionMessages,
                       org.apache.commons.validator.Validator,
                       javax.servlet.http.HttpServletRequest"
                   msg="errors.required"/>

      <validator name="requiredif"
                 classname="org.apache.struts.validator.FieldChecks"
                 method="validateRequiredIf"
                 methodParams="java.lang.Object,

org.apache.commons.validator.ValidatorAction,
                               org.apache.commons.validator.Field,

org.apache.struts.action.ActionMessages,
                               org.apache.commons.validator.Validator,
                               javax.servlet.http.HttpServletRequest"
                 msg="errors.required"/>

      <validator name="validwhen"
          msg="errors.required"

classname="org.apache.struts.validator.validwhen.ValidWhen"
                 method="validateValidWhen"
                 methodParams="java.lang.Object,
                       org.apache.commons.validator.ValidatorAction,
                       org.apache.commons.validator.Field,
                       org.apache.struts.action.ActionMessages,
                       org.apache.commons.validator.Validator,
                       javax.servlet.http.HttpServletRequest"/>

<!—Otras reglas de validación -->
</global>
</form-validation>
```

Bloque de código 6-13: Reglas de validación en validator-rules.xml.

Para la utilización de Validator, necesitaremos copiar las librerías `jakarta-oro.jar` y `antlr.jar` en la carpeta `WEB-INF/lib`, junto a las librerías `commons-beanutils.jar`, `commons-digester.jar`, `commons-validator.jar` y `standard.jar` también necesarias para la utilización de etiquetas de Struts. Todas estas librerías se proporcionan en la distribución de Struts.

6.5.5.2.b Tiles – Composite View

Tiles es un framework para el uso de plantillas que proporciona Stuts. Permite crear portales con una presentación uniforme y crear componentes de la vista reusables. Anteriormente Struts ya proporcionaba la facilidad del uso de plantillas con la librería de etiquetas `template`, sin embargo, con Tiles, Struts proporciona un framework mucho más potente para la creación de la vista.

Para la utilización de Tiles es necesario hacer un cambio importante en la configuración de la aplicación. En el apartado 6.5.5.2 Struts – Front Controller, cuando se introdujo el controlador, se hizo referencia al Request Processor. El Request Processor, como su nombre indica, es el encargado de procesar las peticiones. En ese punto, hablabamos del `RequestProcessor` de Struts, ahora, al utilizar Tiles, tendremos que cambiar la configuración para utilizar `org.apache.struts.tiles.TilesRequestProcessor`, extensión que ofrece Tiles del Request Processor original de Struts.

El motivo de este cambio, es el modo en que se realizan las definiciones de la vista con Tiles. Al igual que los otros frameworks vistos, Tiles nos ofrece la posibilidad de definir la vista en un fichero de configuración, `tiles-def.xml`. La utilización del nuevo Request Porcessor, se debe a la necesidad de integrar las páginas definidas con Tiles con el resto de la aplicación. La extensión de `TilesRequestProcessor` añade esta funcionalidad.

Además, al igual que Validator, es necesario hacerle saber a Struts la ubicación de los recursos que ofrece Tiles.

```
<struts-config>
<!-- Otros parámetros de configuración. -->
<!-- Configuración del controlador. -->

<controller processorClass=
"org.apache.struts.tiles.TilesRequestProcessor"/>
<!--<controller processorClass=
"org.apache.struts.action.RequestProcessor"/>-->

<!-- Plug-in -->

<plug-in className="org.apache.struts.tiles.TilesPlugin">
    <set-property property="definitions-config"
    value="/WEB-INF/Struts/tiles-defs.xml"/>
</plug-in>

<!-- Otros plug in. -->
</struts-config>
```

Bloque de código 6-14: Configuración de Tiles con Struts.

Con Tiles podemos crear distintas plantillas que el resto de páginas extenderán. Por ejemplo, para nuestra aplicación creamos una plantilla básica para la página de acceso (`BasicTemplate.jsp`), una plantilla que será la utilizada por defecto (`DefaultTemplate`) y otra más para la vista de impresión de formularios, resultados de búsquedas y otros (`PrintTemplate`).
En estas plantillas se utilizan etiquetas, de la librería de etiquetas Tiles, para la obtención de datos específicos de cada página. Por ejemplo, en la plantilla por defecto definimos atributos como `title`, `user_working`, `page_links`, `left_links`, `resume` y `content`. En el fichero de definición daremos valores a estos atributos para cada página.

Una vez definidas todas las páginas de nuestra aplicación, debemos configurar adecuadamente las acciones.

Anteriormente hemos visto, en el ejemplo Bloque de código 6-10: Mapeo de acciones., como las acciones una vez realizadas, redireccionaban a una página `*.jsp` o a otra acción. Ahora, al utilizar Tiles, las acciones de Struts no redireccionarán directamente a una página jsp, sino a

una página definida en el fichero de configuración de Tiles, que para identificarlas fácilmente, tendrán el formato `.pagina`.

```
    <action path="/Auth/MainPage"
    type="org.apache.struts.actions.ForwardAction"
    parameter=".mainPage"
    redirect="true"/>
```

Bloque de código 6-15: Mapeo de acciones con Tiles.

```
<tiles-definitions>

<!-- Ejemplo plantilla por defecto. Definimos los valores por defecto,
cada página podrá sobre escribir estos valores. -->
<definition name=".defaultLayout"
page="/HTML/layout/DefaultTemplate.jsp">
    <put name="title" value="${title}"/>
    <put name="user_working" value="/HTML/common/User_working.jsp"/>
    <put name="main_upper_links" value=".mainUpperLinks"/>
    <put name="action_upper_links" value=".actionUpperLinks"/>
    <put name="page_links" value="" type="string"/>
    <put name="user_links" value=".vUserProfileLinks"/>
    <put name="holidays_links" value=".vHolidaysLinks"/>
    <put name="utilities_links" value=".vUtilitiesLinks"/>
    <put name="resume" value="${resume}"/>
    <put name="contentAsMessage" value="content.noContentAsMessage"/>
    <put name="content" value="" type="string"/>
</definition>

<!-- Ejemplo de enlaces superiores de la plantilla por defecto. -->
<definition name=".mainUpperLinks"
page="/HTML/layout/HLinksTemplate.jsp">
    <putList name="hLinks">
        <item classtype="application.util.view.tiles.
        PlusTargetMenuItem" link="/Auth/MainUserProfilePage.do"
        value="links.userProfile.title"/>
        <item classtype="application.util.view.tiles.
        PlusTargetMenuItem" link="/Auth/MainHolidaysServicePage.do"
        value="links.holidaysService.title"/>
        <item classtype="application.util.view.tiles.
        PlusTargetMenuItem" link="/Auth/MainUtilitiesPage.do"
```

```
        value="links.utilities.title"/>
    </putList>
</definition>

<!-- Ejemplo de definición de la página principal. -->
<definition name=".mainPage" extends=".defaultLayout">
    <put name="title" value="title.mainPage"/>
    <put name="resume" value="resume.mainPage"/>
    <put name="content" value="/HTML/MainPage.jsp"/>
</definition>

</tiles-definitions>
```

Bloque de código 6-16:Definición de la vista entiles-def.xml.

En el ejemplo anterior, puede observarse cómo Tiles ofrece la posibilidad de crear listas de elementos, incluidas listas de enlaces. Esto puede se una herramienta muy potente a la hora de crear portales con estructuras de columnas o filas.

Al igual que los otros frameworks, Tiles es altamente personalizable. Por ejemplo, para nuestra aplicación, hemos creado la clase `application.util.view.tiles.PlusTargetMenuItem`. Esta clase nos ofrece más funcionalidades que la clase `org.apache.struts.tiles.beans.SimpleMenuItem` que Tiles proporciona para la creación de links. Con nuestra nueva clase podemos definir el valor del atributo `target` que indica si el enlace se abre en una nueva ventana(útil en la vista de impresión) o en la ventana activa. También hemos creado el atributo `role`, que nos permitirá determinar si un enlace es visible para todos los usuarios, sólo para un usuario administrador, o bien, sólo para usuarios no administradores.

Como puede observarse, este framework está intimamente relacionado con la capa de la vista de la aplicación. Es Tiles quién controlar el uso de las páginas *.jsp, que se utilizarán principalmente para definir contenido.

6.5.5.3 Servlets – Intercepting Filter

Aunque en algunos casos puede plantearse la opción de extender el RequestProcessor de Struts para incluir en éste el filtrado de las peticiones HTTP, la especificación de Servlets, desde su versión 2.3, nos ofrece la posiblidad de implementar el patrón de diseño Intercepting Filter.

Este patrón consiste en el filtrado previo de las peticiones recibidas por el contenedor que soporta nuestra aplicación. La especificación nos permite definir una cadena de filtros que procesen las distintas peticiones según un patrón de URI. Los filtros deben extender al servlet `Filter` directa o indirectamente.

```
<!-- Filtros -->
<filter>
<filter-name>Filter1</filter-name>
<filter-class>classThatExtendsFilter</filter-class>
<init-param>
<param-name>initParameterName</param-name>
<param-value>initParameterValue</param-value>
</init-param>
</filter>

<!-- Mapeo de filtros -->
<filter-mapping>
<filter-name>Filter1</filter-name>
<url-pattern>/*</url-pattern>
<dispatcher>FORWARD</dispatcher>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Bloque de código 6-17: Definición de filtros en web.xml.

Al igual que los servlets, para los filtros es necesario definir un nombre lógico y la clase que implementa el filtro en cuestión. Además, pueden configurarse parámetros iniciales como por ejemplo la página de error si no cumple satisfactoriamente las restricciones que impone el filtro.

Junto a la definición del filtro es necesario el mapeo de URI que deben pasar cada filtro. Puede también especificarse entre otras opciones, si el filtro sólo se pasan a las peticiones procedentes del cliente, o también a las peticiones procentes de un redireccionamiento interno de la aplicación.

Crearemos para nuestra aplicación la clase `DefaultFilter`. Todos los filtros que creemos posteriormente implementaran a esta clase. `DefaultFilter` implementa la interfaz `javax.servlet.Filter` de servlets y sus métodos `init`, `destroy` y `doFilter`. Sin embargo, crea el método abstracto y protegido `doFilterProcess` que el resto de los filtros de nuestra aplicación tendrán que implementar para ejecutar una acción de filtrado en concreto. `DefaultFilter` permite tomar los parámetros iniciales de `forward` y `error`. El valor de `forward` indica dónde redireccionar la petición en el caso de no pasar el filtro satisfactoriamente. El valor del parámetro `error` indica hacia dónde redireccionar la petición si existe un error.

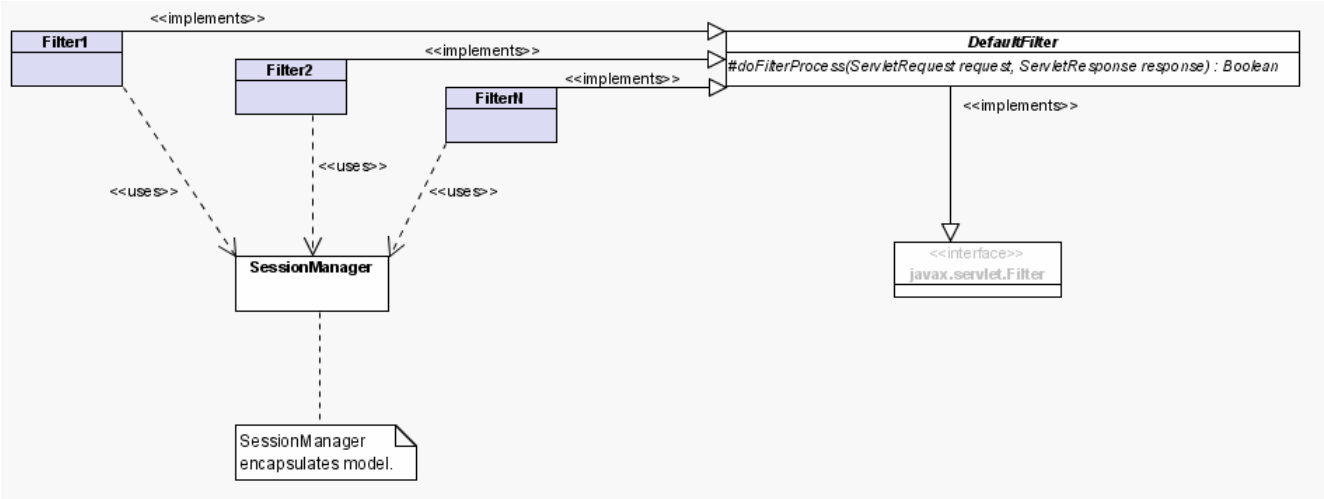
```
/**
 * Implementamos el método doFilter para crear la cadena de
 * filtros y obtener los parámetros iniciales necesarios.
 */
public void doFilter(ServletRequest request, ServletResponse
  response, FilterChain filterChain) throws IOException,
  ServletException{

    RequestDispatcher requestDispatcher = null;

    try {

    /* Ejecuta filtrado. Cada filtro tendrá que implementar su
     * propio método doFilterProcess. */
    Boolean passFilter = doFilterProcess(request, response);

    /* Comprueba la cadena de filtros. */
    if (passFilter){


        filterChain.doFilter(request, response);
```

```
        } else{
            /* Si la petición no ha pasado satisfactoriamente obtiene
           * el parámetro iniciales de forward. */
            String forward = filterConfig.getInitParameter("forward");

            requestDispatcher = request.getRequestDispatcher(forward);
            requestDispatcher.forward(request, response);
        }

        } catch (InternalErrorException e){

            /* Si existe algún error obtiene el parámetro inicial de
             * error para redireccionar la petición. */
            String errorPage = filterConfig.getInitParameter("error");

            requestDispatcher =
            request.getRequestDispatcher(errorPage);
            requestDispatcher.forward(request, response);

        }

    }

    /**
     * Cada filtro tendrá que implementar este método con sus
     * operaciones de filtrado.
     */
    protected abstract Boolean doFilterProcess(ServletRequest request,
            ServletResponse response) throws IOException,
      ServletException,
            InternalErrorException;
```

Bloque de código 6-18: Implementación DefaultFilter.

Para nuestra aplicación utilizaremos dos filtros. Uno de ellos comprobará que un usuario está autenticado en la sesión. El segundo comprobará si el usuario autenticado es administrador.

Tal como se indica en el inicio de este apartado, los filtros de servlet utilizan un mapeo de URI, es por esto que tendremos que definir las acciones con una estructura predefinida para que las peticiones pasen por los filtros definidos.



F. 6-24: Implementación de filtros.

Todas las acciones, excepto la de autenticación, deberán pasar por el filtro de autenticación, `AuthenticationPreprocessingFilter`, y para ellos definiremos un patrón de URI del siguiente modo: `/Auth/*`. Es decir, todas las peticiones que comiencen de este modo (no tenemos en cuenta la parte de la URL que especifica el contexto) serán procesadas con el filtro de autenticación.

Igualmente, definiremos un filtro para aquellas operaciones que exijan que el usuario autenticado sea administrador. Para ello utilizaremos el filtro `AdministratorPreProcessingFilter` y la URI: `/Auth/Adm/*`. Como se observa, estas peticiones pasarán por ambos filtros, primero se comprobará que el usuario está autenticado y posteriormente se comprobará que este usuario es administrador.

Por último definiremos también las URI: `Print/Auth/*` y `Print/Auth/Adm/*` para las acciones con vista de impresión para usuario registrado y usuario administrador. Esto lo hacemos por comodidad a la hora de definir esta acciones con plantillas. Se verá cuando estudiemos la vista de la aplicación.

```
 <!-- Definicion de filtros -->
<filter>
    <filter-name>AuthenticationRequired</filter-name>
    <filter-class>application.http.controller.filtering.
    AuthenticationPreProcessingFilter</filter-class>
    <init-param>
          <param-name>forward</param-name>
          <param-value>/AuthenticationRequired.do</param-value>
    </init-param>
    <init-param>
          <param-name>error</param-name>
    <param-value>/InternalError.do</param-value>
    </init-param>
</filter>

<filter>
    <filter-name>AdministratorRequired</filter-name>
    <filter-class>application.http.controller.filtering.
    AdministratorPreProcessingFilter</filter-class>
    <init-param>
          <param-name>forward</param-name>
          <param-value>/Auth/AdministratorRequired.do
                </param-value>
    </init-param>
    <init-param>
          <param-name>error</param-name>
          <param-value>/InternalError.do</param-value>
    </init-param>
</filter>

<!-- Mapeo de filtros -->
<filter-mapping>
    <filter-name>AuthenticationRequired</filter-name>
    <url-pattern>/Auth/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>

<filter-mapping>
    <filter-name>AdministratorRequired</filter-name>
    <url-pattern>/Auth/Adm/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>

<filter-mapping>
    <filter-name>AuthenticationRequired</filter-name>
    <url-pattern>/Print/Auth/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
```

```
            <dispatcher>REQUEST</dispatcher>
        </filter-mapping>

        <filter-mapping>
            <filter-name>AdministratorRequired</filter-name>
            <url-pattern>/Print/Auth/Adm/*</url-pattern>
            <dispatcher>FORWARD</dispatcher>
            <dispatcher>REQUEST</dispatcher>
        </filter-mapping>
```

Bloque de código 6-19: Definición de filtros en web.xml.

6.5.6 Vista

6.5.6.1 Páginas JSP

Las páginas JSP tienen una apariencia muy similar a las clásicas páginas con etiquetas HTML, sin embargo esta tecnología está especialmente orientada a la creación de contenido dinámico.

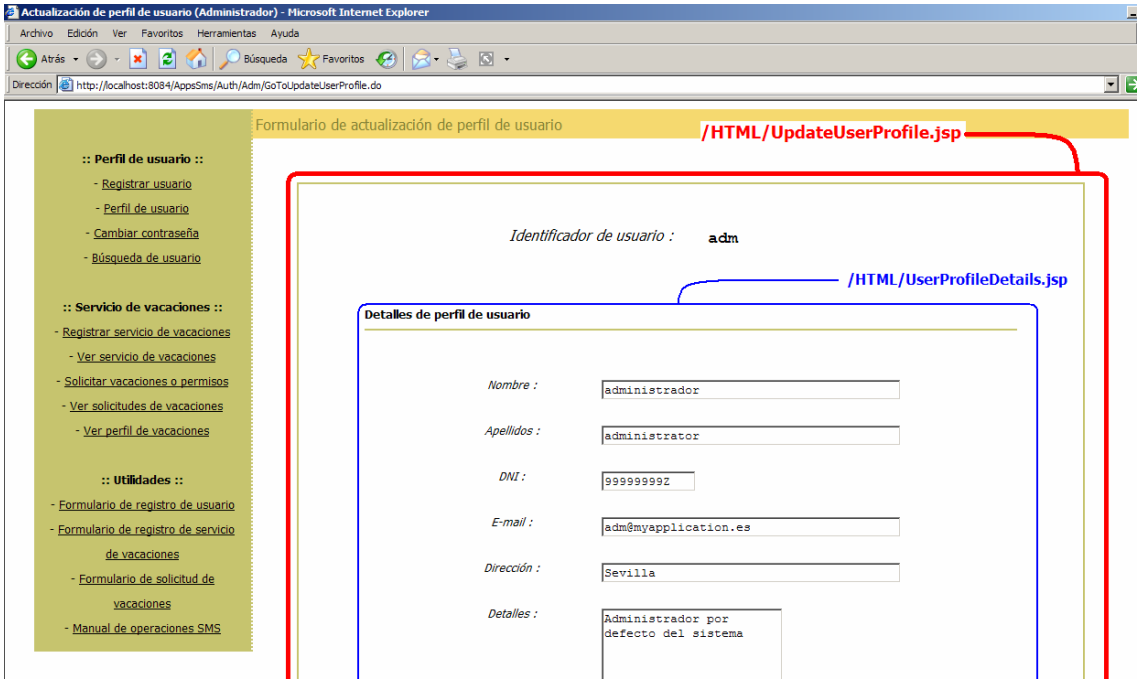
Puesto que JSP es una tecnología de servidor, cuando la página es servida lo hace en código HTML que es lo que le llega al cliente. Por esta razón, la página final servida, debe tener la estructura básica HTML. Debe contener dos partes delimitadas por las etiquetas `<head></head>` y `<body></body>`.

Sin embargo, no todas las páginas JSP tienen que guardar esta estructura. La tecnología JSP ofrece etiquetas JSP y directivas que facilitan la reutilización de código JSP. También pueden contener scriptlets y permite la utilización de etiquetas personalizadas o de otros fabricantes siempre y cuando sigan unas reglas estándares. Por esto, podemos crear páginas JSP que contengan fragmentos de código, que se insertarán en las páginas finales utilizando las directivas, etiquetas o utilidades de frameworks especializados.

La tecnología JSP, por ejemplo, dispone de la directiva `include` y la etiqueta `include`, aunque tienen una sintaxis similar, sus prósitos son distintos. La directiva incluye texto literal en la página JSP cuando la página JSP es compilada. No está pensada para usar con contenido dinámico, si no más bien para incluir texto como cabeceras o pie de página. La etiqueta inserta el contenido, tanto dinámico como estático, cuando la página es servida. En general, la directiva se utiliza para modularizar páginas Web, reutilizar contenido, y mantener tamaños de páginas Web fácil de manejar; la etiqueta se centra principalmente en la inclusión de contenido dinámico en tiempo de ejecución siendo más flexible que la directiva, pero requiriendo tiempo de procesado en tiempo de ejecución por lo que es más lenta que la directiva.

```
<%-- Ejemplo /HTML/UpdateUserProfile.jsp --%>

<%-- Definición de etiquetas utilizadas en la página y resto de
código. --%>

<div class="row">
<span class="label" style="width: 50%; text-align: right"><fmt:message
key="userProfile.loginName"/> : </span>
<span class="value" style="text-align: left; font-weight: bolder;
font-size: larger">
<c:out value="${requestScope.userProfileForm.map.loginName}"/>
</span></div>

<div class="row"><p class="title"><fmt:message
key="userProfile.details.title"/></p></div>

<jsp:include page="UserProfileDetails.jsp"/>

<div class="row"><p class="title"><fmt:message
key="userProfile.access.title"/></p></div>

<jsp:include page="UserProfileAccess.jsp"/>

<div class="row"><p class="title"><fmt:message
key="userProfile.sms.title"/></p></div>

<div class="row"  style="width: 70%; font-size:smaller">

<%-- Resto de código. --%>
```

Bloque de código 6-20: Ejemplo de utilización de etiqueta JSP `include`.



F. 6-25: Ejemplo de utilización de etiqueta JSP `include`.

Desde una página JSP se puede llamar al método `forward` del manejador de peticiones, directamente o a través de una etiqueta personalizada. Esta práctica debe evitarse porque la página JSP estaría actuando como controlador, que debe ser implementado con servlets.

En nuestra aplicación hacemos uso de la etiqueta JSP `forward` en la página de incio de la aplicación, definida en el fichero `web.xml` como `index.jsp`. En el ejemplo Bloque de código 6-21: Ejemplo de utilización de la etiqueta JSP *`forward`*. puede observarse como, aunque la página de inicio es una página JSP, gracias a esta etiqueta pasamos el control al controlador de Struts realizando una invocación a una acción del tipo `*.do`.

```
<%-- index.jsp --%>

<jsp:forward page="Index.do"/>
```

Bloque de código 6-21: Ejemplo de utilización de la etiqueta JSP `forward`.

Cuando se utilizan etiquetas que no son las que la tecnología JSP ofrece por defecto, es necesario declarar estas etiquetas al inicio de la página JSP. En nuestra aplicación utilizamos la implementación de las librerías de etiquetas estándares del proyecto Jakarta y las etiquetas del framework de Struts. La definición de estas librerías de etiquetas se realizan con la utilización de la directiva `taglib`.

La directiva `taglib` nos permite definir un prefijo para cada librería y la URI que es la ruta del descriptor de la librería. En el caso de JSTL y Struts se utilizan URI estándares. Para utilizar nuestras propias etiquetas, en URI se tendría que indicar la ruta exacta en nuestra aplicación del descriptor de la librería.

```
<%-- Librerías de etiquetas JSTL y Struts utilizadas. --%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<html:form action="/Auth/Adm/LoginAsUser.do" focus="password">

<fieldset>

<div class="row"><blockquote><p class="rule">
(<fmt:message key="rules.includePassword"/>.)</p></blockquote></div>

<div class="row">
<span class="label"><fmt:message key="userProfile.loginName"/> :

<%-- Resto de código. --%>
```

Bloque de código 6-22: Ejemplo de utilización de la directiva JSP `taglib`.

Junto a las etiquetas y código HTML, en una página JSP podemos encontrar partes de código (scriptlets) en código Java. Estos trozos de código se delimitan por los símbolos <% y %>, y aunque no se recomienda su abuso por temas de mantenibilidad, en algunas ocasiones puden ser útiles.

Para la obtención de datos dinámicos se puede utilizar utilizar el lenguaje EL definido en la especificación JSP 2.0. Este lenguaje presenta una alternativa más intuitiva para diseñadores no familiarizados con la programación Java. Para la utilización del lenguaje EL se utilizan los delimitadores ${ y }. Veamos a continuación un ejemplo de obtención de datos dinámicos con scriptlets y con lenguaje EL.

```
<%-- Ejemplo de utilización de scriptlets para la obtención de datos
dinámicos. --%>
<c:set var="errors"
value="<%=request.getAttribute("org.apache.struts.action.ERROR")%>"/>
<c:if test="${!(empty errors)}"><p class="error"><fmt:message
key="error.action.login"/></p></c:if>

<%-- Ejemplo de utilización de lenguaje EL para la obtención de datos
dinámicos. --%>
<div class="row"><span class="label" style="width: 50%; text-align:
right">
<fmt:message key="userProfile.loginName"/> : </span>
<span class="value" style="text-align: left; font-weight: bolder;
font-size: larger">
<c:out
value="${requestScope.userProfileForm.map.loginName}"/></span></div>
```

Bloque de código 6-23: Ejemplo de utilización de scriptlets y lenguaje EL para la obtención de datos dinámicos.

Los comentarios JSP se delimitan con <%-- y --%>. Estos comentarios no son enviados al cliente ya que no se traducen a código HTML.

6.5.6.2 Hojas de Estilo

Además, utilizaremos Hojas de Estilo para ofrecer distintos formatos. De este modo, al mostrar la información, según la hoja de estilo aplicada lo hará de una forma u otra. Pueden definirse distintos colores, tamaños de letras, estilos de fuentes, margenes, y muchísimas más carácterísticas que afectan al formato. También puede definirse si cierto bloque, es visible o no, funcionalidad muy útil para no mostrar botones de formularios, ni enlaces en la vista de impresión. Pueden utilizarse hojas de estilo en cascada, de forma que se aplican distintas hojas de estilo simultáneamente según unas reglas de prioridad de elementos.

Las hojas de estilo se pueden definir en un fichero externo de extensión `css` o bién en la misma página JSP (o HTML, ya que esta tecnología no es exclusiva para JSP). Con esta tecnología podemos aplicar distintos estilos de formato a cada etiqueta definiendo una serie de atributos. Tiene propiedades como herencia y se basa en una definición de formato basados en bloques. Cada bloque puede definir su tamaño, situación fija o relativa, margen, bordes, rellenos, formatos de texto, fondo del bloque (tanto color como imagen), etc.

```
<!-- Hoja de estilo para vista Web. -->
    BODY {  background-color: #FFFFFF;
            margin: 2em;
            color: black;
            line-height: 2em;
            font-family: tahoma, serif;
            font-size: small;
            font-style: normal;
            font-weight: normal;
            }

    A { color: black }

    BLOCKQUOTE {    line-height: 1.5em; text-align: justify; }

    DIV.logo {  float: left;
                position: fixed;
                top: 2em;
```

```
                line-height: 2em;
                font-size: large;
                font-weight: bold;
                }
<!-- ... -->
    DIV.content {   float: right;
                    position: fixed;
                    right: 2em;
                    top: 12em;
                    width: 80%;
                    background-color: #FFFFFF;
                    padding: 3em;
                    color: black;
                    text-align: center;
                    }
<!-- ... -->
    P.error {   color: red;}
<!-- ... -->
    #m {    font-family: monospace; }
```

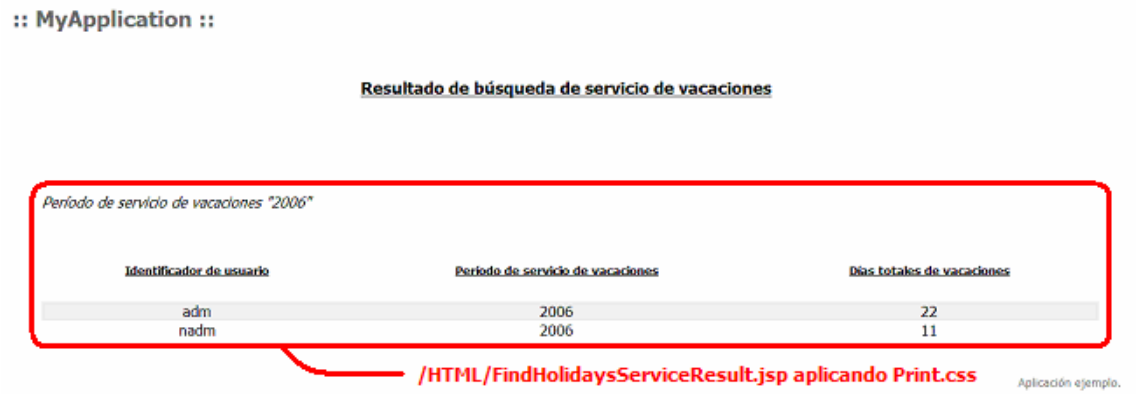
Bloque de código 6-24: Ejemplo de hoja de estilo.

```
<%-- /HTML/layout/DefaultTemplate.jsp --%>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1">
<title><fmt:message key="${title}"/></title>

<!-- Aplicación de hoja de estilo externa. -->
<LINK REL=StyleSheet
HREF="${pageContext.request.contextPath}/CSS/Default.css"
TYPE="text/css">

</head>
```

Bloque de código 6-25: Aplicación de hoja de estilo externa.

En nuestra aplicación utilizaremos dos hojas de estilos. `Default.css` se aplicará a la vista Web, `Print.css` se aplicará a la vista de impresión. Veamos a continuación un ejemplo visual de sus aplicaciones.

F. 6-26: Aplicación de hojas de estilo. Vista Web.



F. 6-27: Aplicación de hojas de estilo. Vista de impresión.

6.5.6.3 Librerías de etiquetas JSTL

Como implementación de las librerías de etiquetas estándares (JSTL) utilizamos la que nos proporciona la Apache Software Foundation a través del subproyecto TagLibs de Jakarta.

En nuestra aplicación utilizaremos sólo las librerías `core` y `fmt` de este proyecto. La librería core nos permite realizar cierta lógica como bloques if, iteraciones, manejo de variables y operaciones relacionadas con URL. La librería fmt nos ofrece utilidades de internacionalización que veremos en 6.5.6.6 Internacionalización.

Para utilizar estas librerías es necesaria su configuración en el fichero `web.xml` y la librería `jstl.jar`. Esta librería la copiamos en la carpeta `lib`, en `WEB-INF`, donde se colocarán las librerías necesarias para ejecutar nuestra aplicación. Además crearemos una carpeta `StdTagLibs` donde se copiarán los descriptores de cada una de las librerías estándares utilizadas, en nuestro caso `c.tld` y `fmt.tld`.

Tanto los descriptores de las librerías como la librería `jstl.jar` se proporcionan con cada distribución de JSTL.

```
<!-- web.xml -->
<!-- ... -->

         <!-- Definición de etiquetas estándares JSTL. -->

         <taglib>
         <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
         <taglib-location>/WEB-INF/StdTagLibs/fmt.tld
            </taglib-location>
         </taglib>

         <taglib>
         <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
         <taglib-location>/WEB-INF/StdTagLibs/c.tld</taglib-location>
         </taglib>

<!-- ... -->
```

Bloque de código 6-26: Definición de JSTL en web.xml.

```
<%-- Ejemplo de etiquetas core.
/HTML/FindHolidaysServiceResult.jsp --%>

<%-- Definición de etiquetas JSTL. --%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<%-- <c:choose> en cominación con <c:when> y <c:otherwise>, y la
etiqueta <c:if> nos permite mostrar una sección de la página según una
condición se cumpla o no. --%>
<c:choose> <c:when test="${empty requestScope.holidaysServices}">
<tr><td colspan="5"  style="padding : 2em 0em"><fmt:message
key="find.result.empty"/>.</td></tr>
</c:when>

<c:otherwise>

<%-- La librería core también nos permite trabajar con variables. En el
ejemplo utilizamos una variable local para controlar el estilo de cada
línea en el resultado de una búsqueda. --%>
<c:set var="color" value="0"/>

<%-- La etiqueta <c:forEach> nos permite realizar iteraciones sobre una
variable. En el ejemplo es útil para mostrar los resultados de una
búsqueda en una variable tipo Collections y mostrar cada resultado
iterando por esta variable. --%>
<c:forEach var="holidaysService"
items="${requestScope.holidaysServices}">

<fmt:formatDate pattern="yyyy" value="${holidaysService.period.time}"
var="period"/>

<c:set var="color" value="${!color}"/>
<tr id="${color}">

<td>
<%-- <c:url> nos permite trabajar con una URL y sus parámetros. --%>
<c:url var="userdetailsURL" value="Auth/Adm/GoToLoginAsUser.do">
<c:param name="loginName"
value="${holidaysService.loginName}"/></c:url>
<c:if test="${!(empty sessionScope.administratorName)}">
<a href="<c:out
value='${pageContext.request.contextPath}/${userdetailsURL}'/>"></c:if>
<c:out value="${holidaysService.loginName}"/><c:if
test="${sessionScope.administratorName}"></a></c:if></td>

<td><c:out value="${period}"/></td>

<td><c:out value="${holidaysService.days}"/></td>

<c:if test="${(!(empty sessionScope.administratorName)) &&
(sessionScope.loginName == holidaysService.loginName) }">
```

```
<td style="font-size : smaller;" id="noprint"><c:url var="editURL"
value="Auth/Adm/GoToUpdateHolidaysService.do">
<c:param name="period" value="${period}"/></c:url>
<a href="<c:out
value='${pageContext.request.contextPath}/${editURL}'/>"><fmt:message
key="link.edit"/></a>
</td></c:if>

</tr>
</c:forEach>

<!-- ... -->
```

Bloque de código 6-27: Ejemplo de utilización de JSTL. Librería de etiquetas core.

En la figura F. 6-26: Aplicación de hojas de estilo. Vista Web.puede la vista final que ofrece el Bloque de código 6-27: Ejemplo de utilización de JSTL. Librería de etiquetas core.

6.5.6.4 Librerías de etiquetas de Struts (y Tiles)

Para poder utilizar las librería de etiquetas de Struts en nuestras propias aplicaciones, sólo tendremos que copiar las librerías necesarias en nuestro directorio `/WEB-INF/lib` y copiar los ficheros `lib/struts-*.tld` que ofrece también la distribución de Struts bajo la carpeta `WEB-INF` de nuestra aplicación. Normalmente aunque es configurable se sitúan en una subcarpeta de `WEB-INF` con el nombre de `struts` donde también se suele situar el fichero de configuración `struts-config.xml` y otros ficheros de configuración si se utilizar por ejemplo Validator y Tiles.

El conjunto de librerías de etiquetas que Struts nos proporcionan son las librerías `bean`, que permite imprimir el valor de las propiedades de los Java Beans; `html`, para la generación de código; `logic`, para el control de flujo; y `template`, para la utilización de plantillas. Aunque como hemos visto en apartados anteriores, Struts puede utilizarse con otros frameworks, por lo que también proporciona librería de etiquetas de éstos, por ejemplo `Tiles`. De estas librerías sólo utilizaremos las librerías html y tiles, y utilizaremos la alternativa que ofrecen las librerías de etiquetas estándares a `bean` y a `logic` (que se corresponderían con `core` de JSTL).

Para el correcto funcionamiento de la aplicación, al utilizar las etiquetas `html` y `tiles`, son necesarias las librerías `commons-beanutils.jar`, `commons-digister.jar`, `commons-validator.jar` y standard.jar. Estas librerías están disponibles con la distribución de Struts, y deben colocarse en la carpeta WEB-INF/lib de nuestra aplicación. Además, en WEB-INF/Struts, se copiarán los descriptores de estas librerías struts-html.tld y struts-tiles.tld.

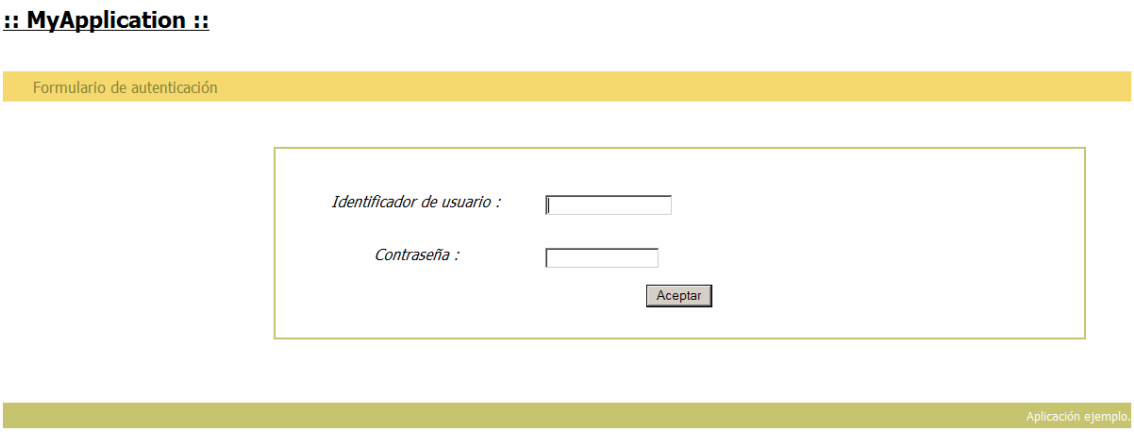
Struts con el grupo de etiquetas `html` nos facilita el URL rewriting. Estas etiquetas que solucionan los problemas de URL rewriting (mantenimiento de sesiones) y gestión de formularios, además de ayudar a generar la vista, corresponden a la aplicación del patrón View Helper.

La etiquetas `<html:form>` tiene asociada una acción definida en el atributo action. Esta acción debe corresponder con una acción en el mapeo de acciones del fichero de configuración de Struts. A su vez, esta acción debe tener asociado un formulario. Las etiquetas `html` de tipo input (`<html:text>`, `<html:password>`, etc) harán referencia a los distintos campos del formulario correspondiente. Esto permite, además de la entrada y salida de datos, el manejo de errores.

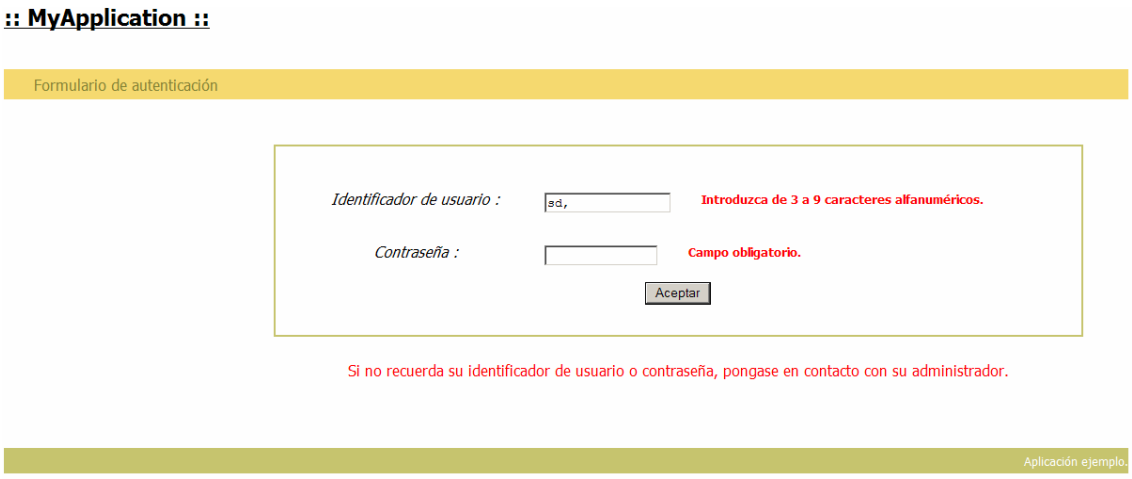
Las etiquetas `<html:errors>` y `<html:error>` permiten mostrar los errores asociados a un campo o a todo el formulario. Puede mostrar mensajes de error originados por el método `validate` del formulario `ActionForm`, por Validator (antes de realizar la acción) o bien errores definidos programáticamente al originados al intentar realizar la acción correspondiente.

```
<!-- Login.jsp -->

<%-- Definición de etiquetas de Struts y JSTL. --%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<%-- Ejemplo de utilización de etiquetas html. Este formulario está
asociado a la acción Login.do --%>
<html:form action="Login.do" focus="loginName">

<fieldset>

<div class="row">
<span class="label"><fmt:message key="userProfile.loginName"/> :
</span>
<%-- Si se ha pasado en la petición, <html:text> toma el valor
existente para el campo loginName, si no, se muestra vacío para
introducir datos. --%>
<span class="value"><html:text property="loginName" size="16"
maxlength="9" styleId="m"/></span>
<%-- Con <html:errors> podemos mostrar los errores asociados a cada
campo, si existen en la petición. --%>
<span class="error"><html:errors property="loginName"/></span>
</div>

<!-- Otros campos del formulario ... -->

<div class="row"><span class="button"><html:submit><fmt:message
key="buttons.accept"/></html:submit></span></div>

</fieldset>
<!-- ... -->
</html:form>
```

Bloque de código 6-28: Ejemplo de utilización de la librería de etiquetas html de Struts.



F. 6-28: Ejemplo de utilización de la librería html de Struts para formularios.

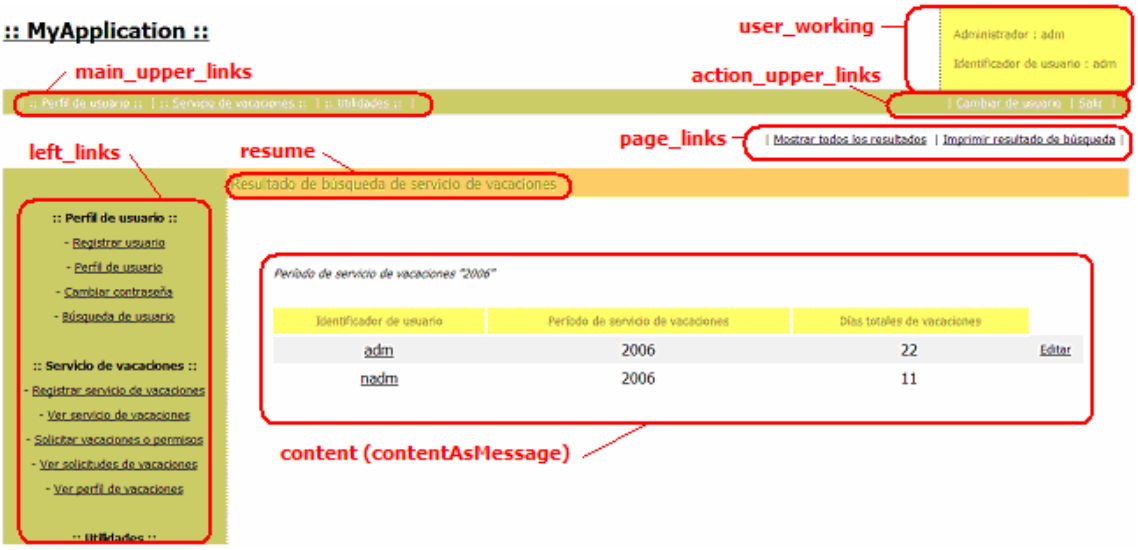


F. 6-29: Ejemplo de utilización de la libreria html de Struts para formularios. Entrada de datos con errores.

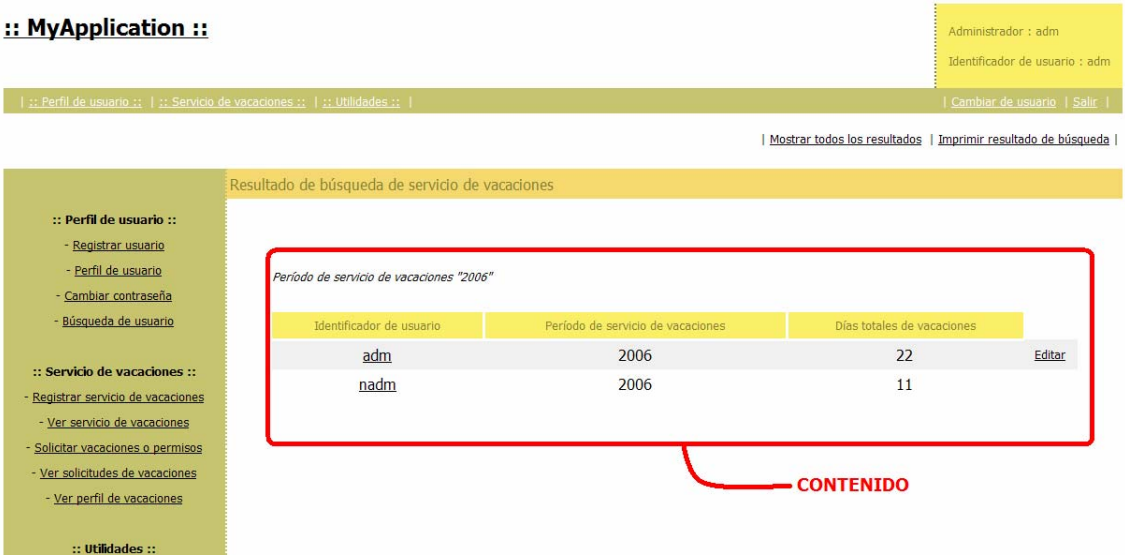
En las figuras anteriores se muestra una página utilizando Tiles para su composición. Sin embargo, las etiquetas de Tiles pueden utilizarse independientemente si se utiliza `TilesRequestProcessor` o no.

En ambos casos, la estructura general de la página se definirán en las páginas JSP que servirán de plantilla. En las plantillas se utilizarán etiquetas `<tiles:get>` y `<tiles:useAsAttribute>` para definir los objetos que cada página debe especificar. Si no se utilizara Tiles como plug-in, sino sólo sus etiquetas, por cada página definida en el fichero `tiles-def.xml`, sería necesario crear una página JSP con etiquetas `<tiles:put>` y, evidentemente, este fichero no sería necesario. Esta opción tiene la desventaja de requerir un mayor tiempo de desarrollo y mantenimiento al aumentar el número de páginas JSP.

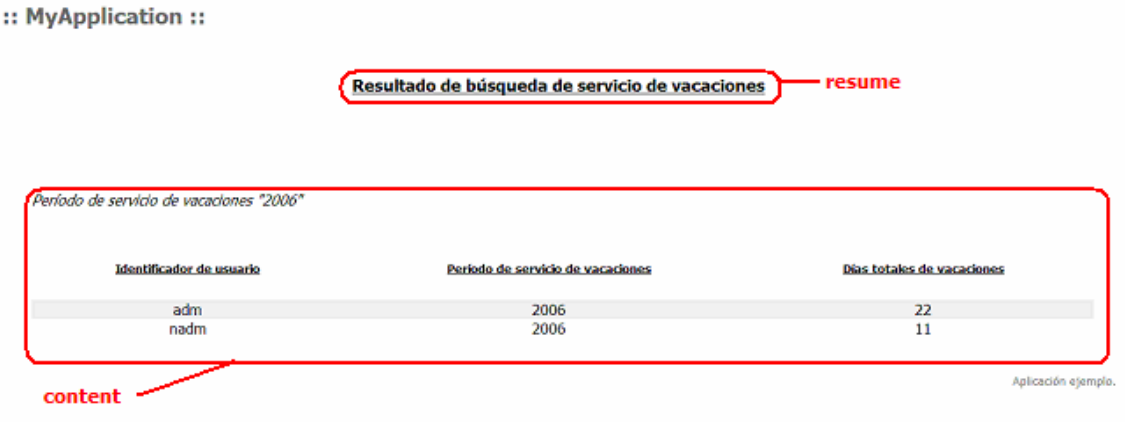
Si utilizamos Tiles como plug-in, también es necesario la creación de las plantillas correspondientes, sin embargo, el resto de páginas JSP serán sólo contenido (u otras subplantillas) de las definidas en `tiles-def.xml`. Esto permite una mayor reutilización de estas páginas. Por ejemplo, una misma página JSP puede utilizarse para una vista normal de una página en la Web y para una vista de impresión con el mismo contenido (teniendo en cuenta que es contenido dinámico). En estos casos, normalmente se utilizan además distintos tipos de hojas de estilo asociadas a cada plantilla.



F. 6-30: Elementos de la plantilla DefaultTemplate.jsp.



F. 6-31: Reutilización de contenido. Información a mostrar.



F. 6-32: Ejemplo plantilla PrintTemplate.jsp. Reutilización de contenido. Vista de impresión.

Como puede verse, gracias a la utilización de Tiles que permite la reutilización de contenido en distintas plantillas, y la utilización de hojas de estilos, que permiten mostrar un mismo contenido con distinto formato, podemos fácilmente mostrar vistas distintas de la misma información. En los ejemplos anteriores utilizamos dos plantillas distintas, pero además, cada plantilla tiene asociada una hoja de estilo diferente.

```
<%-- BasicTemplate.jsp. Ejemplo de plantillas utilizando Tiles. --%>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>

<html>
<%-- Elementos que se obtendrán como atributos (texto). --%>
<tiles:useAttribute name="title"/>
<tiles:useAttribute name="resume"/>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1"><title><fmt:message key="${title}"/></title>
<LINK REL=StyleSheet
HREF="${pageContext.request.contextPath}/CSS/Default.css"
TYPE="text/css">
</head>

<body>
<div class=logo><a
href="${pageContext.request.contextPath}/Auth/MainPage.do"> ::
MyApplication :: </a></div>
<div class="resume" style="{clear:both}"><fmt:message
key="${resume}"/></div>
<div class=content><tiles:get name="content"/></div>
<div class="footer"><fmt:message key="content.footer"/>.</div>
<body>
</html>
```

Bloque de código 6-29: Ejemplo plantilla BasicTemplate.jsp utilizando Tiles.

6.5.6.5 Etiquetas personalizadas

En la aplicación no utilizamos este tipo de etiquetas, sin embargo, para la ampliación que se realizará posteriormente se intentará proporcionar un ejemplo que sí utilice etiquetas personalizadas o Custom Tags.

6.5.6.6 Internacionalización

La internacionalización hece referencia a la posibilidad de mostrar datos en un lenguaje, con un formato o con cualquier otra característica, en función de una variable llamada `locale`. `Locale` es una variable formada por un par `[idioma, país]`. Donde idioma y país son valores de dos letras según la norma ISO-639 para los idiomas y según la norma ISO-3166 para los países. Localización es utilizar y configurar estas características para un idioma o país determinado.

Aunque tanto Struts como JSTL nos ofrecen funcionalidades de internacionalización, en nuestra aplicación no utilizamos internacionalización propiamente, ya que no hacemos uso de la variable locale. Sin embargo, utilizaremos algunas de las características que las librerías orientadas a internacionalización proporcionan.

F. 6-33: Códigos de idiomas y países para localización (ejemplo).

En este punto es importante hablar del fichero Messages.properties. Este fichero, contiene un conjunto de pares clave=valor que nos permite asociar a una serie de claves su valor para el idioma por defecto. En nuestra aplicación, el valor por defecto es Español. Si quisiéramos aprovechar las características de localización, podríamos tener tantos ficheros Messages.properties_xx_XX, donde xx y XX son los códigos de según las normas ISO para un idioma y país específico. Todos estos ficheros tendrían el mismo conjunto de claves, sin embargo, cada clave tendría valores distintos en distintos idiomas. Veamos un ejemplo en el Messages.properties de nuestra aplicación.

```
# ...

errorsMessage.incorrect.holidaysrequest = Fecha de solicitud
incorrecta
errorsMessage.serviceNotAvaliable = El usuario no está dado de alta en
este servico para el periodo solicitado
errorsMessage.requestAlreadyExist = Periodo ya solicitado
anteriormente

# DEFAULT

default.yes = Sí
default.no = No

buttons.accept = Aceptar
buttons.cancel = Cancelar
buttons.update.state = Actualizar estado
buttons.reject = Rechazar
buttons.reset = Reset

content.noContentAsMessage
content.footer = Aplicación ejemplo

administratorRequired.content = <p class="internalError">Necesita
      permisos de administrador para realizar la operación
# ...
```

F. 6-34: Internacionalización. Ejemplo Messages.properties.

Para que tanto Struts, como JSTL, reconozcan este fichero con las fuentes de localización es necesario configurarlo en `web-xml` para JSTL y en `struts-config.xml` para Struts. Veamos a continuación cómo lo hemos configurado en nuestra aplicación. Normalmente, este fichero se situa dentro de la carpeta `WEB-INF/classes`, directamente o en alguna subcarpeta.

```
<!-- web.xml -->
<web-apps>

<!-- Otros parámetros de configuración. -->
      <!-- Configuración de localización para JSTL. -->

         <context-param>
         <!-- Parámetro que necesitamos configurar. -->
         <param-name>javax.servlet.jsp.jstl.fmt.localizationContext
            </param-name>
         <!-- Valor del parámetro, bajo la carpeta clases y con la
         extensión properties. -->
         <param-value>Messages</param-value>
         </context-param>
<!-- Otros parámetros de configuración. -->
</web-apps>
```

Bloque de código 6-30: Configuración de localización para JSTL.

```
<!-- struts-config.xml -->
<struts-config>
<!-- Otros parámetros de configuración. -->
<Configuración de localización para Struts. Busca un fichero bajo la
carpeta classes, con extensión properties. -->
<message-resources parameter="Messages" null="false"/>
<!-- Otros parámetros de configuración. -->
</struts-config>
```

Bloque de código 6-31: Configuración de localización para Struts.

Con la configuración de los ejemplos, JSTL y Struts utilizan los mismos ficheros. Sólo es necesario configurarlo para el fichero por defecto. Si se utilizan ficheros de localización para distintos idiomas, deben encontrarse en la misma carpeta que Messages.properties, pero con lo nomenclatura indicada más anteriormente.

Cada aplicación puede manejar la variable locale según los objetivos. Por defecto, cada navegador utiliza un locale configurado en las propiedades del navegador, de este modo,

automáticamente, se presentarán las fuentes de localización en el lenguaje especificado en el navegador, y en el caso de no encontrar el fichero Messages.properties_xx_XX para esa localización en concreto, se mostrará en el lenguaje por defecto.

Otra forma para manejar idiomas puede ser la elección de un locale específico mostrando la opción al usuario desde la propia aplicación con enlaces a distintos idiomas. Estos enlaces sólo tendrán que asignar a locale el valor correspondiente para que se muestren las fuentes en ese idioma.

También podrían guardarse las preferencias de idioma de un usuario en una cookie o en alguna base de dato. Sólo sería necesario guardar el locale elegido por el usuario y recuperarlo cuando el usuario inicie una sesión.

Nuestra aplicación no implementa localizaciones distintas al lenguaje español, sin embargo, la utilización del fichero Messages.properties y las etiquetas correspondientes permitirían que una ampliación a más localizaciones se realizara de un modo sencillo y rápido.

Una vez configuradas las fuentes de localización existen varios modos de acceder a ellas. Para la vista, en nuestra aplicación utilizamos las etiquetas de JSTL `<fmt:message>`. Esta etiqueta recibe como atributo la clave que se quiere recuperar, y según el locale, se obtendrá su valor en el idioma correspondiente. Aunque no lo utilizamos, Struts proporciona esta funcionalidad con la etiqueta `<bean:message>`.

Validator también proporciona distintas claves para que en caso de error de validación sea recuperado el valor del mensaje de error en el idioma correspondiente. Igualmente, con la utilización de la clase `org.apache.struts.action.ActionMessage`, podemos recuperar mensajes de error con la etiqueta `<html:errrors>` de Struts de la localización correspondiente.

```
<!-- Login.jsp -->
<%-- Librerías de etiquetas. --%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<html:form action="Login.do" focus="loginName">
<fieldset>
<div class="row"><span class="label">
<%-- Utilización de librería fmt para obtener valores de localización.
--%>
<fmt:message key="userProfile.loginName"/> : </span>
<span class="value"><html:text property="loginName" size="16"
maxlength="9" styleId="m"/></span>
<%-- Con <html:errors> obtiene el error en el lenguaje adecuado según
la localización. --%>
<span class="error"><html:errors property="loginName"/></span>
</div>
<br>
<div class="row">
<span class="label"><fmt:message key="userProfile.password"/> :
</span>
<span class="value"><html:password property="password" size="16"
maxlength="9"/></span>
<span class="error"><html:errors property="password"/></span>
</div>
<br>
<div class="row"><span class="button"><html:submit><fmt:message
key="buttons.accept"/></html:submit></span></div>
</fieldset>

<c:set var="errors"
value="<%=request.getAttribute("org.apache.struts.action.ERROR")%>"/>
<c:if test="${!(empty errors)}"><p class="error"><fmt:message
key="error.action.login"/></p></c:if>

</html:form>
```

Bloque de código 6-32: Utilización de etiquetas de localización.

```
<!-- validator.xml -->
<!-- loginForm -->
<form name="loginForm">
    <field property="loginName" depends="required, mask">
    <var><var-name>mask</var-name>
        <var-value>^\w{3,9}$</var-value></var>
    <!—Si existe errores de validación por no cumplir el patrón
especificado, se busca el valor de la clave “errorsMessage.mask.login”
según localización. Este valor contiene {0}, que será sustitido por el
valor correspondiente de la clave “userProfile.loginName”. -->
    <msg name="mask" key="errorsMessage.mask.login"/>
    <arg0 name="mask" key="userProfile.loginName"/>
    </field>

    <field property="password" depends="required, mask">
    <var><var-name>mask</var-name>
        <var-value>^\w{3,9}$</var-value></var>
    <msg name="mask" key="errorsMessage.mask.login"/>
    <arg0 name="mask" key="userProfile.password"/>
    </field>
</form>
```

Bloque de código 6-33: Localizacón con Validator.

```
package application.http.controller.actions.userprofile;
//imports

import org.apache.struts.action.ActionMessages;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.DynaValidatorForm;

import application.util.exceptions.InstanceNotFoundException;
import application.util.exceptions.IncorrectPasswordException;
import application.util.exceptions.InternalErrorException;

import application.util.controller.struts.DefaultAction;

/**
 * Log in Action.
 */
public class LoginAction extends DefaultAction {

    /** ...
     */
```

```
    public ActionForward doExecute(ActionMapping mapping, ActionForm
            form, HttpServletRequest request, HttpServletResponse
            response) throws IOException, ServletException,
            InternalErrorException {

        /* Obtiene Datos. */
        DynaValidatorForm loginForm = (DynaValidatorForm) form;
        String loginName = (String) loginForm.get("loginName");
        String password = (String) loginForm.get("password");

        /* Preparamos errores a capturar. */
        ActionMessages errors = new ActionMessages();
        try {
            SessionManager.login(request, loginName, password);
        } catch (InstanceNotFoundException e) {
            //se proporciona el la clave para posteriormente obtener
            //el valor en el fichero de localización correspondiente.
            errors.add("loginName", new
                ActionMessage("errorsMessage.notFound.loginName"));
        } catch (IncorrectPasswordException e) {
            errors.add("password", new
                ActionMessage("errorsMessage.incorrect.password"));
        }

        if (errors.isEmpty()) {
            return mapping.findForward("LoginAction");
        } else {
            saveErrors(request, errors);
            return new ActionForward(mapping.getInput());
        }
    }
}
```

Bloque de código 6-34: Internacionalización. Clase ActionMessage de Struts.

Además estos frameworks proporcionan utilidades de formateo de datos, útil para cifras y fechas. Nosotros sólo utilizamos `<fmt:formatDate>` para formateo de algunas fechas.

```
<!-- FindHolidaysRequestResult.jsp -->
<!-- ... -->
<c:set var="forUpdate" value="0"/>
<html:form action="/Auth/Adm/MapUpdatesHolidaysRequest.do">
<c:forEach var="holidays" items="${requestScope.holidayss}">

<fmt:formatDate pattern="dd-MM-yyyy"
value="${holidays.holidaysRequestVO.stateDate.time}"
var="formattedStateDate"/>
<fmt:formatDate pattern="dd-MM-yyyy"
value="${holidays.holidaysRequestVO.startDate.time}"
var="formattedStartDate"/>
<fmt:formatDate pattern="dd-MM-yyyy"
value="${holidays.holidaysRequestVO.finalDate.time}"
var="formattedFinalDate"/>

<!-- ... -->

<td><c:out value="${formattedStateDate}"/></td>
<td><c:out value="${formattedStartDate}"/></td>
<td><c:out value="${formattedFinalDate}"/></td>

<!-- ... -->
```

Bloque de código 6-35: Formateo de fechas con JSTL.

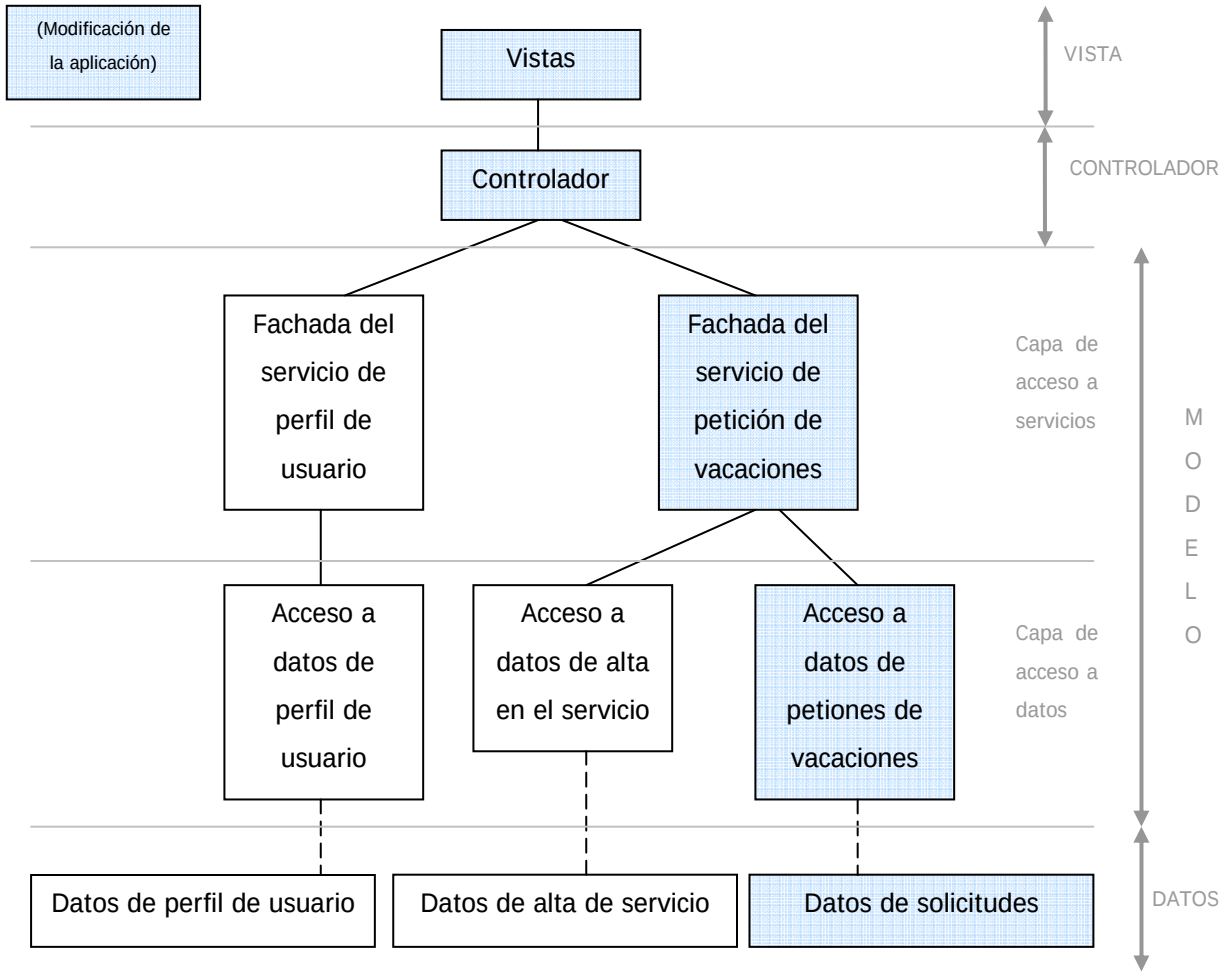
6.6 POSIBLES AMPLIACIONES DE LA APLICACIÓN

Como se ha comentada desde el inicio, el mayor beneficio que las tecnologías utilizadas en el diseño de la aplicación es la escalabilidad. De este modo, actualizar la aplicación, y desarrollar nuevas ampliaciones se puede realizar de una forma cómoda y rápida.

Una de las ampliaciones que se podría proponer sería la **implementación del servicio de petición de días de permisos**, ya que en nuestro ejemplo sólo hemos abarcado la solicitud de días de vacaciones. Quizás esta sea una de las ampliaciones más complejas, ya que supondría la ampliación del modelo de la aplicación. Las ampliaciones necesarias afectarían

desde las capas de datos hasta la vista pero sólo en aquellas partes pertenecientes al servicio de vacaciones.

Sería necesario realizar modificaciones desde la capa de datos, por ejemplo, se podría incluir una nueva tabla de registros de solicitud de días de permisos o bien añadir un nuevo atributo en las registros que represente si la solicitud corresponde a vacaciones o permisos, y en este último caso que tipo de permiso. Para ello se podría utilizar un campo de valor un número entero que identifique cada tipo de solicitud. Si nos decantamos por la segunda solución sólo necesitaremos realizar modificaciones en la rama de la aplicación orienta a la solicitud de vacaciones para tener en cuenta este atributo. Se podrían definir filtros de búsquedas basados en el tipo de solicitud y añadir información asociada al perfil de vacaciones (por ejemplo añadiendo un atributo que muestre los días de permiso tomados por el usuario teniendo en cuenta que estos días no descuentan de los días totates de vacaciones).



F. 6-35: Modifcaciones necesarias para la ampliación del servicio de solicitud de días de permiso.

Otra ampliación aún más interesante es la **integración con herramientas de calendario** que facilitaran la tramitación de estas solicitudes. En el ejemplo hemos considerado laborables todos los días del año que no son sábado o domingo, sin embargo, podría modificarse la ampliación para que la clase JworkingDay pudiera cargar distintos calendarios según el año. También se podría utilizar de algún tipo de calendario visual, ya sea perteneciente a la misma aplicación o

algún calendario externo. Para ello se propone se crearían utilidades para trabajar con el estandar iCalendar(RFC 2445). Este éstandar es el utilizado por herramientas de calendario como la reciente GoogleCalendar, y permitiría fácilmente la importación y exportación de datos de calendario.

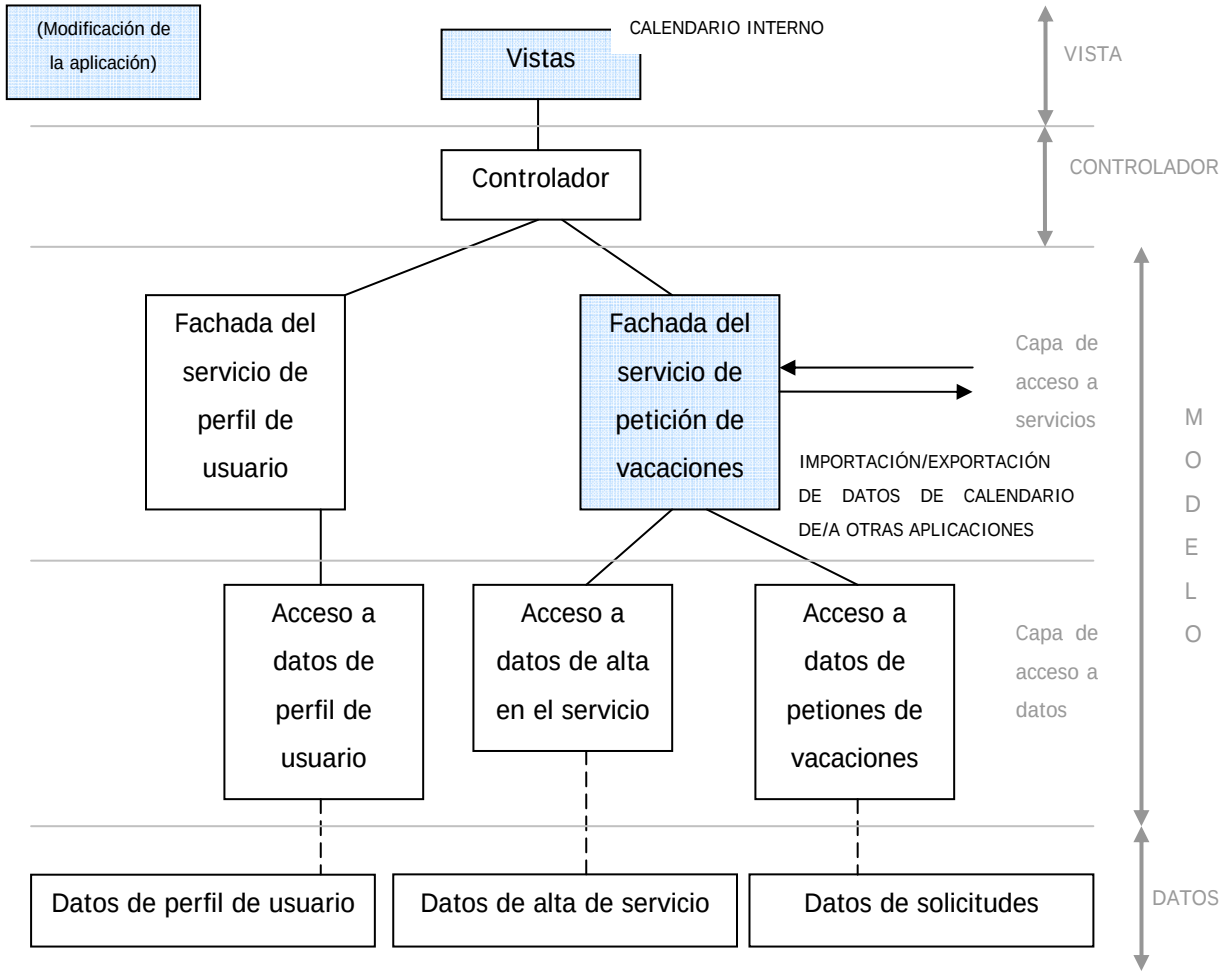
```
BEGIN:VCALENDAR
PRODID:-//Ximian//NONSGML Evolution Calendar//EN
VERSION:2.0
METHOD:PUBLISH
BEGIN:VTIMEZONE
TZID:/softwarestudio.org/Olson_20011030_5/Europe/Helsinki
X-LIC-LOCATION:Europe/Helsinki
BEGIN:DAYLIGHT
TZOFFSETFROM:+0200
TZOFFSETTO:+0300
TZNAME:EEST
DTSTART:19700329T030000
RRULE:FREQ=YEARLY;INTERVAL=1;BYDAY=-1SU;BYMONTH=3
END:DAYLIGHT
BEGIN:STANDARD
TZOFFSETFROM:+0300
TZOFFSETTO:+0200
TZNAME:EET
DTSTART:19701025T040000
RRULE:FREQ=YEARLY;INTERVAL=1;BYDAY=-1SU;BYMONTH=10
END:STANDARD
END:VTIMEZONE
BEGIN:VEVENT
UID:20030821T212557Z-2393-500-1-2@somehost.example.com
DTSTAMP:20030821T212557Z
DTSTART;TZID=/softwarestudio.org/Olson_20011030_5/Europe/Hel
DTEND;TZID=/softwarestudio.org/Olson_20011030_5/Europe/Helsi
20030818T113000
SUMMARY:foo
SEQUENCE:1
LAST-MODIFIED:20030821T212558Z
END:VEVENT
END:VCALENDAR
```

F. 6-36: Ejemplo de definición de datos de calendario con iCalendar.

(Ejemplo procedente de http://en.wikipedia.org/wiki/VCalendar)

Como puede observarse en la figura este estándar es tan sólo un fichero de texto con unos campos definidos (algunos pueden tener valor nulo). De este modo la integración con calendarios se puede realizar de un modo sencillo y sin afectar al modelo de nuestra aplicación.

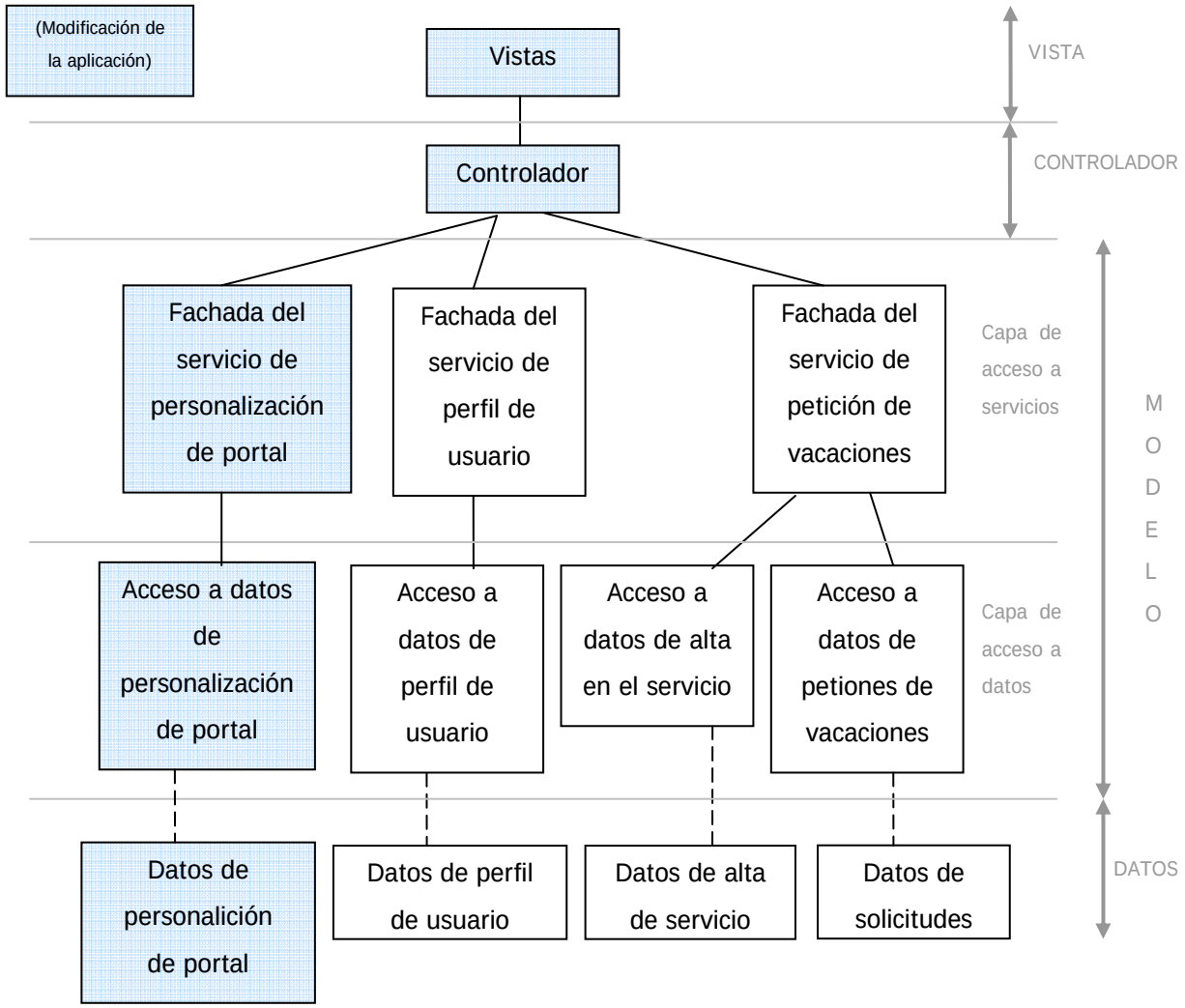
Por otra parte, la creación de un calendario en la misma aplicación es mucho más sencillo. Basta con crear una página JSP que realice un búsqueda sobre todas las solicitudes creadas y marque los días en los que exista alguna solicitud y el nombre de usuario.



F. 6-37: Modificación de la aplicación para la inclusión de servicio de calendario externo o interno.

Hasta ahora, las ampliaciones presentadas, suponen modificación de alguno de los servicios existentes. Supongamos que queremos ofrecer un **servicio de personalización del portal**. Este servicio podría permitir al usuario distinta gama de colores para la aplicación, posibilidad de distintas plantillas, y quizás un menú personalizado de enlaces rápidos. Para ello sería necesaria la creación de nuevos objetos de datos relacionados con este servicio, así como la capa de acceso a estos datos y la fachada de operaciones disponibles sobre estos datos. En ningún caso afectaría a la aplicación ya existente en capas del modelo, aunque habría que añadir nuevos servicios en la capa controlador y vista.

También podría incluirse, como se comentó en el apartado anterior, una posible ampliación que podría consistir en ofrecer mejores características de **internacionalización**. Para ello, sólo sería necesaria la creación de nuevos ficheros Messages.prorperties_xx_XX para distintas localizaciones y realizar algunas modificaciones en el controlador para el manejo de la localización, ya sea por enlaces a distintos idiomas o por preferencias del usuario.



F. 6-38: Modificación de aplicación para una ampliación de personalización de portal.

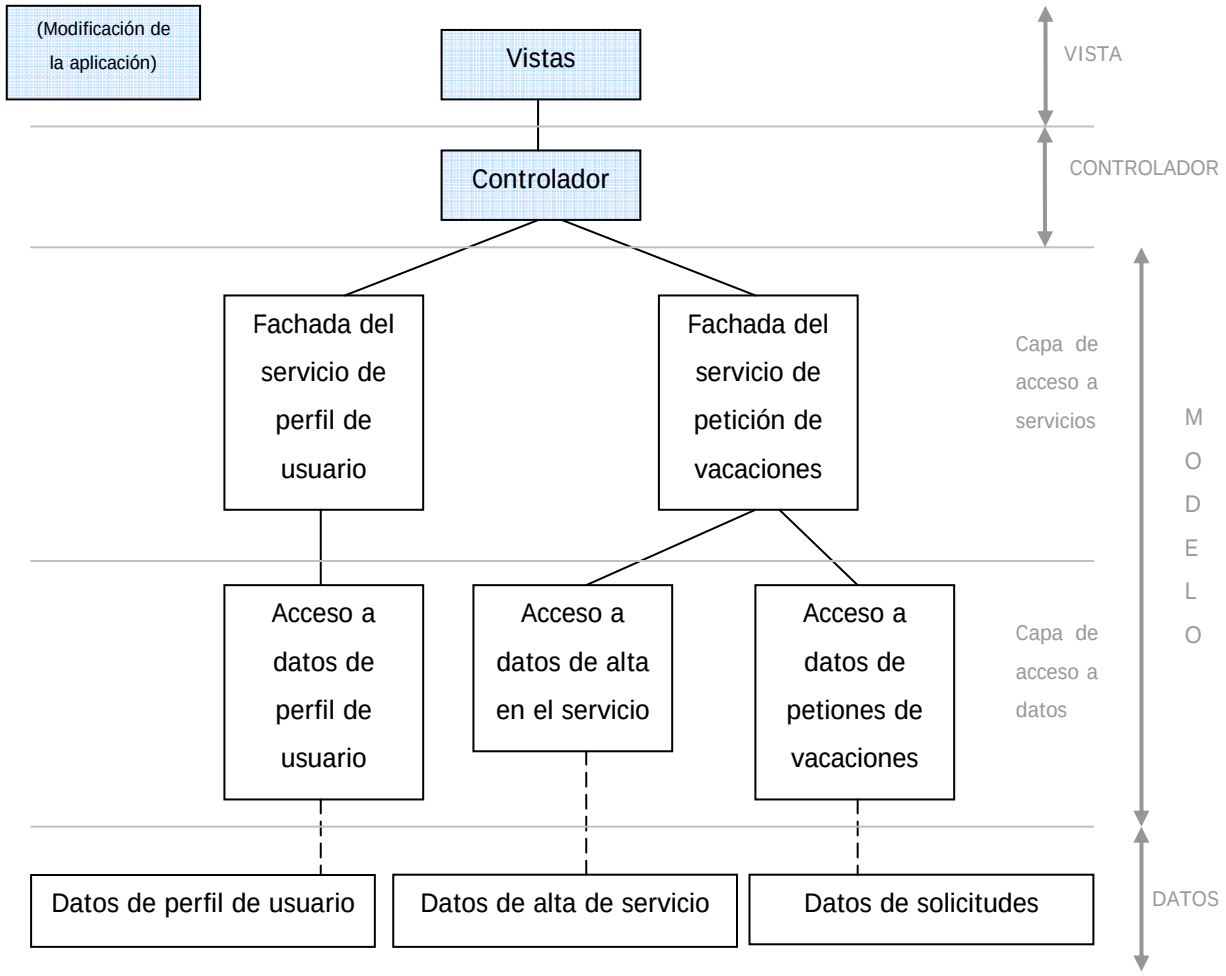
Por último presentamos una ampliación más acorde con el tema que nos abarca: MOVILIDAD. Desde el primer capítulo nos hemos centrados en aplicaciones empresariales con movilidad. Por ello, nos centramos en las aplicaciones Web, ya que son las aplicaciones que mayor movilidad ofrecen al usuario al poder acceder a la aplicación desde cualquier ordenador con conexión a Internet. Ahora proponemos la ampliación de los servicios actuales para proporcionar aún

mayor movilidad al usuario; ampliaremos los servicios existentes para proporcionar estos mismos **servicios vía SMS**.

Esta ampliación es también una muestra de la escabilidad de la aplicación. Pretendemos ofrecer un servicio ya existente, por lo que no será necesaria la modificación del modelo, sin embargo, para ofrecer un servicio de SMS, necesitaremos una pasarela de SMS y gestionar la comunicación de ésta con nuestra aplicación. Normalmente, esta pasarela podrá comunicarse por HTTP, como lo hacemos desde un navegador, pero hay que tener en cuenta que el establecimiento de sesión por parte del usuario no tiene sentido en la utilización de SMS.

La comunicación a través de SMS tiene más sentido en un entorno petición/respuesta que en un entorno de sessión. Por este motivo, para ofrecer servicios de SMS será necesario añadir componentes en el controlador que manejen el envio y recepción de mensajes. Además, la vista utilizada en Web no será muy útil para un SMS, así pues, crearemos nuevas vistas orientas a mensajes que presentarán una vista de sólo texto.

Puesto que esta ampliación, no es más sencilla, ni más complicada que las ampliaciones anteriores, sin embargo, debido al carácter de movilidad que presenta, se desarrollará e implementará con más detalles en los siguientes capítulos.



F. 6-39: Modificaciones de la aplicación para ofrecer servicios vía SMS.