UOC

Matching entre servidor de recursos y consumidores

Trabajo Final de Carrera (TFC)

ALEJANDRO VOLKOFF BAZAREVITCH

Enginyeria Tècnica en Informàtica de Sistemes

Segundo semestre curso 2004-2005

Consultor:

JULIÀ MINGUILLÓN I ALFONSO

Resumen

El **objetivo** del Trabajo Final de Carrera (TFC) fue desarrollar una aplicación que permitiera, por un lado, la definición de una oferta de recursos; por otro, el que unos usuarios-consumidores pudieran apuntarse a dichas ofertas y, por fin, que el sistema fuera capaz de asignar los consumidores a los recursos, a partir de las especificaciones establecidas.

Este TFC tiene el **interés** de servir como base para posibles implementaciones por parte de la UOC, en las áreas de gestión de los TFC, ofertas de distinto tipo que se puedan ofrecer, etc.

Todo el **desarrollo** se realizó sobre plataforma Java, con el interés de ser compatible con la plataforma habitual de la UOC.

En cuanto a los **datos**, se decidió almacenarlos en el Sistema Gestor de Bases de Datos (SGBD) PostgreSQL, por estar considerado el mejor SGBD existente de código abierto.

Por otro lado, mi **interés personal** ha sido adentrarme en el desarrollo con Java para la web (incluyendo la tecnología de Java Server Pages y el servidor Apache Tomcat) y poder experimentar con el SGBD PostgreSQL.

Todo el **software utilizado** ha sido de código abierto, incluyendo el sistema operativo sobre el cual se realizó el desarrollo (GNU / Linux).

Contenido

1 Introducción..... 6

1.1 Especificaciones de la aplicación.....6

1.2 Especificaciones técnicas.....7

1.3 Planificación inicial.....7

2 Análisis y diseño..... 9

2.1 Planteo del problema.....9

2.2 Elementos de la aplicación..... 9

Recursos.....9

Consumidores.....10

Matching.....11

2.3 Casos de uso..... 11

Diagrama UML.....11

Crear recurso.....12

Modificar recurso.....12

Crear matching.....12

Ejecutar matching.....12

Crear consumidor.....13

Completar atributos.....13

2.4 Lógica de la aplicación.....14

2.5 Clases de persistencia.....15

Diagrama UML simplificado.....16

Diagrama UML completo del área de Recursos.....17

Diagrama UML completo del área de Consumidores.....18

Diagrama UML completo del área de Matches.....18

Diagrama UML completo del área de acceso a datos.....19

Descripción de las clases.....19

2.6 Base de datos.....21

Esquema de la base de datos.....21

Definición de las tablas y campos..... 22

2.7 Interfaz de usuario..... 25

Diagrama de los scriplets JSP (formularios web y capa de negocio).....26

Descripción de los scriplets.....26

3 Funcionamiento de la aplicación..... 34

3.1 Procedimiento.....34

3.2 Entrada en la aplicación....35
3.3 Administración de recursos y matching....35
3.4 Consumidores-Usuarios....36
3.5 Gestión de errores....36
4 Plan de trabajo....37
4.1 Análisis....37
4.2 Diseño orientado a objetos....37
4.3 Diseño de la base de datos....37
4.4 Pruebas tecnológicas....37
4.5 Implementación de la capa de datos....38
4.6 Implementación de la capa de negocio....38
4.7 Implementación de la interfaz de usuario....38
5 Tecnología utilizada....39
5.1 Almacenamiento: base de datos relacional....39
¿Por qué base de datos relacional?....39
Otras posibles opciones....39
Adaptación a otras tecnologías....39
5.2 Servidor de base de datos: PostgreSQL....39
¿Por qué PostgreSQL?....39
Otras posibles opciones....40
Adaptación a otras tecnologías....40
5.3 Motor: Clases Java....40
¿Por qué Java?....40
Otras posibles opciones....40
Adaptación a otras tecnologías....40
5.4 Interfaz de usuario: web (JavaServer Pages - JSP)....40
¿Por qué JavaServer Pages?....40
Otras posibles opciones....40
Adaptación a otras tecnologías....41
5.5 Servidor web: Apache Tomcat 4....41
¿Por qué Tomcat?....41
Otras posibles opciones....41
Adaptación a otras tecnologías....41
5.6 Sistema operativo: Debian GNU / Linux....41
¿Por qué GNU / Linux?....41

Otras posibles opciones.....41
Adaptación a otras tecnologías.....42
5.7 Entorno de desarrollo: Sun NetBeans 4.....42
¿Por qué NetBeans?.....42
Otras posibles opciones.....42
Adaptación a otras tecnologías.....42
5.8 Herramienta CASE: Poseidon UML CE.....42
¿Por qué Poseidon?.....42
Otras posibles opciones.....42
Adaptación a otras tecnologías.....42
6 Aspectos no implementados..... 43
Validación de los datos introducidos al crear un recurso.....43
Áreas.....43
Completado de los datos personales de un consumidor.....43
Cambio de contraseña.....43
Gestión de plantillas de matching.....43
Restricciones de puntuación mínima para un recurso.....44
Restricciones de mínimo y máximo de personas apuntadas.....44
Cálculo del matching.....44
Gestión de usuarios administradores.....44
Borrar a los consumidores después de un matching.....44
Almacenamiento de los datos de acceso para la base de datos.....44
Mejor aprovechamiento del servidor de base de datos.....45
Optimización y perfeccionamiento del código.....45
Documentación del código.....45
7 Pruebas realizadas.....46
8 Conclusiones..... 52
Qué cambiaría si tuviera que volver a hacerlo.....52
9 Glosario de la aplicación.....54
10 Bibliografía..... 56

1 Introducción

Dada mi experiencia de algunos años como desarrollador de aplicaciones, para clientes diversos, me costó ubicarme en el punto de vista de desarrollar un TFC y no una aplicación acabada y lista para usar. En el caso del TFC, he comprendido que el interés primario era desarrollar un esquema básico, con una funcionalidad mínima, que pudiera ser ampliado, pero contando con dicha base. En el caso del desarrollo para clientes, el interés primario de estos es un software que funcione, sin importarles el diseño interno de la aplicación. Es evidente que, cuando se parte de un diseño inadecuado, se multiplica la energía que se necesitará para el mantenimiento de la aplicación, pero esto suele considerarse como secundario en el mundo laboral, donde prima la entrega final de un producto "con cara y ojos", máxime cuando dicho producto ha de ser vendido a terceros.

Así pues, mi punto de vista inicial fue desarrollar una aplicación que pudiera estar acabada y funcionando en los plazos improrrogables de entrega; por tanto, ciertas decisiones de diseño e implementación fueron tomadas teniendo en cuenta este aspecto; es decir, ante la posible dificultad que me encontraría utilizando determinadas tecnologías en las cuales tengo poca experiencia (por ejemplo, XML), preferí utilizar tecnologías que conozco mejor (como el trabajo con bases de datos relacionales), para no encontrarme al final del plazo con un producto que no acabara de funcionar.

Con el tiempo, me dí cuenta que quizás hubiera sido mejor adoptar otro punto de partida, intentando diseñar la aplicación "de la mejor manera posible", aunque luego me encontrara con problemas prácticos a la hora de implementarla.

Con todo esto no intento excusarme ante ciertas decisiones de diseño, porque considero que éste es adecuado para el objetivo planteado, pero sí explicar algunas de estas decisiones. Por otro lado, comprendo que tres meses no son quizás tiempo suficiente para desarrollar al completo una aplicación como la pretendida, debiendo priorizar ciertos aspectos por encima de otros.

1.1 Especificaciones de la aplicación

Poder realizar un matching entre dos grupos de elementos, uno ofertado y el otro consumido. Los nombres que dimos a estos elementos han sido "Recursos" y "Consumidores" (o "Usuarios").

A partir de unos recursos ofertados, en los cuales hay unos determinados requisitos, y de unos consumidores, que completan dichos requisitos obteniendo

finalmente una puntuación global, se intenta elaborar un matching que optimice la puntuación global.

Así, cada recurso tendría unos consumidores que se han apuntado, y cada uno de ellos con una puntuación. Al efectuar el matching global, a cada recurso sería asignado uno o más consumidores (en función de unas restricciones), maximizando la puntuación sumada de todos los recursos.

Por ejemplo, en el caso de la UOC o bien de otra organización, se podrían ofertar unos determinados puestos para ocupar (también podrían ser los propios Trabajos Finales de Carrera); en cada uno de estos puestos, habría determinados requisitos imprescindibles (como ciertos conocimientos, vivir en determinado lugar, tener una edad determinada, etc.) y otros requisitos adicionales, que podrían servir para aumentar la puntuación del consumidor.

1.2 Especificaciones técnicas

Se pretende que la aplicación no sea un producto cerrado funcionando, sino más bien una aplicación abierta, que sea capaz de plantear las mejores alternativas para resolver estos problemas, y que sea adaptable a otros entornos, principalmente el entorno de la UOC.

Dado que el entorno de la UOC es Java, ésta será la plataforma de la aplicación. Asimismo, dado que los usuarios interaccionan con la UOC a través del Campus, la interfaz de la aplicación será web.

En cuanto al resto de la aplicación, hay libertad para decidir sobre los aspectos técnicos.

1.3 Planificación inicial

Del	Al	
26 de febrero	7 de marzo	Planificación del trabajo. Planteamiento del problema.
8 de marzo	13 de marzo	Análisis.
14 de marzo	27 de marzo	Diseño de los casos de uso. Diseño de las clases. Diseño de los formularios – interfaz de usuario.
28 de marzo	3 de abril	Diseño de la base de datos.
4 de abril	10 de abril	Instalación y pruebas PostgreSQL.
11 de abril	17 de abril	Instalación y pruebas Apache Tomcat. Instalación y pruebas IDE desarrollo. Creación de la base de datos.
18 de abril	28 de abril	Implementación de las clases de acceso a datos.

Del	Al	
29 de abril	15 de mayo	-- (Ausente por viaje.)
16 de mayo	22 de mayo	Diseño esqueleto de la memoria.
23 de mayo	29 de mayo	Implementación de la capa de negocio. Implementación de la interfaz de usuario.
30 de mayo	5 de junio	Implementación de la capa de negocio. Implementación de la interfaz de usuario. Trabajo con la memoria.
6 de junio	12 de junio	Testing de la aplicación. Mejoras en el código. Trabajo con la memoria.
13 de junio	20 de junio	Testing de la aplicación. Mejoras en el código. Trabajo con la memoria. Presentación de la memoria.

2 Análisis y diseño

Al iniciar el trabajo, se realizó un análisis y diseño; durante el desarrollo del TFC, el diseño inicial sufrió diversos cambios, incluso en los casos de uso, que es el que se refleja a continuación. Estos cambios fueron motivados por la retroalimentación que iba recogiendo al realizar el desarrollo.

2.1 Planteo del problema

El problema final es una función de maximización sobre un grafo bipartito con aristas ponderadas.

Para llegar a esta resolución, es necesario diseñar un sistema que sea capaz de administrar los recursos, los consumidores y finalmente calcular el resultado final (o sea, asignar consumidores a recursos).

Esto plantea diversas cuestiones: formato y almacenamiento de los datos de los recursos y consumidores, cálculo de las puntuaciones de cada consumidor y del matching final, interfaz de administración, tanto de recursos como de matching, e interfaz de consumidor.

2.2 Elementos de la aplicación

Recursos

Un recurso se compone de atributos propios y diversas propiedades. A su vez, cada propiedad se compone de diversos elementos.

Un recurso tiene una *descripción* y unas *observaciones*, que pueden servir como ampliación de la descripción; además pertenece a un *área*, lo cual facilitaría la selección por parte de los consumidores.

Cada recurso tiene asignado un *peso*, que pondera (o normaliza) la puntuación de cada consumidor. Por ejemplo, un recurso con un peso de 1,2 tiene un valor un 20% superior a un recurso con un peso de 1. Esto quiere decir que, a la hora de maximizar el resultado global, se valorará más los puntos del recurso con más peso. Por defecto, el peso para un recurso es 1.

Un recurso puede tener una *puntuación mínima* exigible para que un consumidor le sea asignado (por defecto es 0, lo cual quiere decir que cualquier consumidor le puede ser asignado); asimismo puede tener un número *mínimo y/o máximo de consumidores* para asignarle (por defecto tanto el mínimo como el máximo son 1).

Una **propiedad** tiene un *peso*, que pondera la importancia relativa de la propiedad dentro del recurso. Por defecto, el valor es 1.

Una propiedad tiene asociada una *lista de elementos*, que conforman las opciones que se presentarán a un consumidor. Por ejemplo, en el caso de la edad, se pueden definir franjas de edad; en el caso de conocimiento de idiomas, se pueden definir diferentes posibilidades (conocimiento alto, medio o bajo); etc.

Además, una propiedad se puede considerar *excluyente*. En ese caso, se creará uno o más elementos cuya puntuación sea 0. Si un consumidor, al entrar sus datos, no satisface dicho requerimiento, queda automáticamente excluido de la futura asignación entre recursos y consumidores.

Una propiedad puede permitir seleccionar sólo un elemento, o bien varios elementos ("*multi-lista*").

Si una propiedad sólo permite elegir un elemento de entre varios, cada elemento tendrá una puntuación entre 0 y 1. Si un consumidor selecciona un elemento que vale 1, querrá decir que el consumidor satisface dicha propiedad al 100%; inversamente, si selecciona un elemento cuyo valor es 0, querrá decir que el consumidor no satisface dicha propiedad, y si además esa propiedad es excluyente, el consumidor quedará excluido de ser seleccionable para dicho recurso. Por ejemplo, una propiedad podría ser: "¿Tiene usted carnet para conducir automóviles?"; en este caso, los elementos podrían ser 2: "sí" (valor 1) o "no" (valor 0). Otro ejemplo sería: "Conocimiento de inglés" y sus elementos serían "alto" (valor 1), "medio" (valor 0,7), "bajo" (valor 0,3) y "ninguno" (valor 0).

Si una propiedad permite la selección de más de un elemento, cada elemento tendrá una puntuación entre 0 y 1, debiendo la suma total de elementos ser igual a 1. En este caso, la puntuación de cada elemento seleccionado se suma para conformar la puntuación del consumidor para esa propiedad.

En cualquiera de las situaciones anteriores, un consumidor acabará teniendo una puntuación entre 0 y 1 para cada propiedad.

Consumidores

Un consumidor (que en definitiva será el usuario del sistema) se compone de una serie de datos personales (nombre, documento, domicilio, etc.); de hecho, estos datos lo que hacen es darlo de alta ante el sistema (además, ya podría ser usuario del sistema para otros aspectos, en cuyo caso estos datos personales estarían excluidos de esta aplicación).

Para que un usuario se convierta en consumidor, debe al menos apuntarse a un

recurso. En ese momento, contestará una serie de preguntas (o una serie de ítems) que determinarán, en primer lugar, si es seleccionable para ser asignado a dicho recurso, y en segundo lugar su puntuación.

Un consumidor puede apuntarse a todos los recursos que desee, pero será asignado o no a ellos en función de los otros consumidores que se hayan apuntado y en función de las restricciones de cada recurso.

Matching

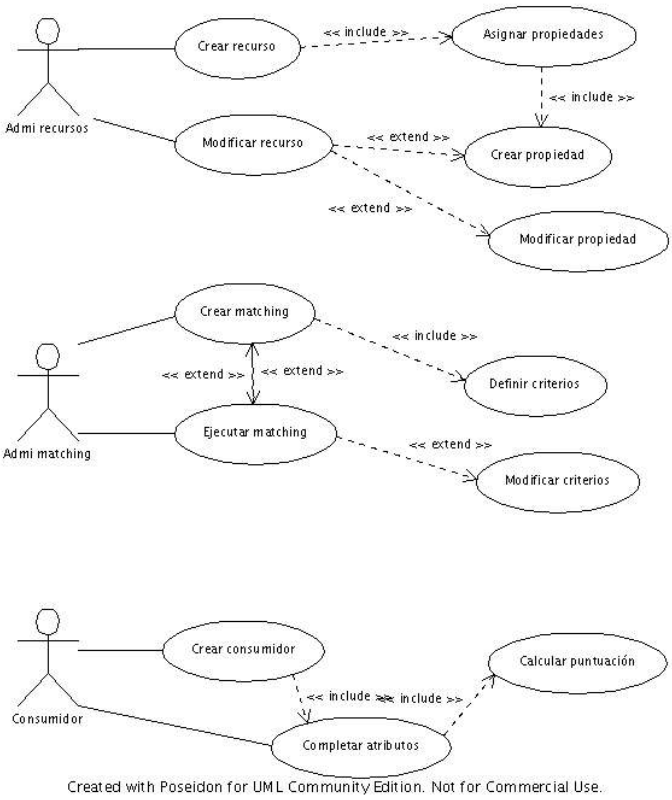
Un matching es una asignación de consumidores a recursos. Se realiza en momentos puntuales (por ejemplo, se puede publicar unos recursos durante 3 o 6 meses para que los consumidores se vayan apuntando y, en una fecha determinada, se efectúa la asignación).

El matching toma en cuenta las puntuaciones de cada consumidor apuntado a cada recurso, las restricciones propias de cada recurso (en cuanto a número de consumidores y puntuación mínima) y, opcionalmente, puede definir unas restricciones globales que se sumen a las anteriores (por ejemplo, determinar una puntuación mínima para todos los recursos, o determinar el máximo de recursos a los cuales se puede apuntar un consumidor).

2.3 Casos de uso

En primer lugar, se diseñó un diagrama de casos de uso, con el objetivo de tener claro cuáles serían los roles (o actores), y cuáles serían las atribuciones de cada rol.

Diagrama UML



Crear recurso

Precondición: El sistema debe estar instalado y funcionando.

La creación de un nuevo recurso implica definir algunos de sus atributos (descripción, área, observaciones, etc.) y, sobre todo, la asignación de propiedades. Para esto, es necesario crear cada una de las propiedades, y dentro de cada una de ellas crear los elementos que compondrán la propiedad.

Postcondición: Se ha creado un nuevo recurso, que puede estar disponible para que los consumidores se apunten.

Modificar recurso

Precondición: El recurso debe estar creado y ser editable.

Una vez que un recurso ha sido creado, mientras éste sea editable (es decir, pueda ser modificable; esto quiere decir que el recurso aún no está disponible para los consumidores), se lo puede modificar. Este sería un caso de uso poco habitual, porque lo más probable sería que, al crear un recurso, éste ya estuviera completamente definido, y en ese caso se lo podría "cerrar" (es decir,

convertir en no editable), poniéndolo a disposición de los consumidores.

Postcondición: Se ha modificado un recurso, que puede haber quedado disponible para que los consumidores se apunten.

Crear matching

Precondición: El sistema debe estar instalado y funcionando.

Un matching no es más que la asignación de los consumidores a los recursos. Dicho matching puede tener ciertas restricciones o criterios; este caso de uso contempla la creación de una plantilla de matching, independiente de los recursos y consumidores actualmente activos, con sus restricciones, para ser almacenado y ser ejecutado posteriormente.

Postcondición: Se ha creado una plantilla de matching, que podrá ser utilizada cuando se ejecute el matching.

Ejecutar matching

Precondición: Deben existir recursos y consumidores apuntados en ellos.

Esto es la asignación de consumidores a recursos. Es la fase final de toda la aplicación, donde cada recurso queda emparejado con sus consumidores.

Se puede ejecutar un matching previamente almacenado, con sus criterios, o bien definir el matching y ejecutarlo en un mismo paso. Este seguramente sería el caso más común.

Postcondición: Se ha efectuado la asignación de consumidores a recursos, maximizando la puntuación global.

Crear consumidor

Precondición: El sistema debe estar instalado y funcionando.

Este caso de uso es dar de alta un consumidor, cosa que realiza él mismo. Un consumidor se da de alta en el sistema, completando unos datos personales, y a partir de ese momento ya puede apuntarse a los recursos que le interesen y estén disponibles.

Postcondición: El consumidor se ha dado de alta en el sistema.

Completar atributos

Precondición: Deben existir recursos activos y no editables (o cerrados).

Una vez que un consumidor está dado de alta en el sistema, puede apuntarse a

cualquier recurso disponible que desee; para ello, deberá completar una serie de atributos o propiedades, que variarán en función del recurso que ha elegido. Cada atributo tendrá una serie de elementos asociados, de los cuales el consumidor deberá elegir uno o más (dependiendo de la definición de la propiedad como "multi-lista"); cada uno de estos elementos tendrá asociada una puntuación.

Al completar todos los atributos, el sistema calculará la puntuación de dicho consumidor correspondiente a dicho recurso. Podría ocurrir que el consumidor no se considere apto para ser asignado a un recurso (porque incumple alguna propiedad definida como excluyente), en cuyo caso se le asignará una puntuación 0.

El **proceso de puntuación** es el siguiente:

1) Recorrer todos los elementos indicados por el consumidor, de todas las propiedades del recurso. Verificar en cada propiedad si es excluyente, y en ese caso mirar si el consumidor la cumple. Si no la cumple, asignar una puntuación global de 0 para ese consumidor en ese recurso, y finalizar el proceso. Si la propiedad no es excluyente o bien el consumidor no ha sido excluido, calcular el total de puntos obtenidos en esa propiedad, y multiplicarlo por el peso de dicha propiedad.

2) Si el consumidor no ha sido excluido, sumar el valor total que ha obtenido en cada una de las propiedades; dividir la puntuación obtenida por el consumidor entre el peso total de las propiedades del recurso, para obtener el porcentaje de cumplimiento (por ejemplo, si el peso total de las propiedades es 8 y el consumidor ha obtenido una puntuación total de 7, 7 / 8 = 0,875, que equivale al 87,5% de la puntuación del recurso). Finalmente, multiplicar esta cifra obtenida por el peso del recurso, para obtener la puntuación final, que es la que se utilizará a la hora de calcular el matching.

Postcondición: El consumidor está apuntado a un recurso o bien está excluido de la selección, y no puede volver a apuntarse.

2.4 Lógica de la aplicación

Podemos decir que la aplicación se componen de *clases de almacenamiento* (se componen de atributos solamente), *clases que implementan parte de la lógica relacionada con el acceso a la base de datos, scriplets* no visuales ***que implementan el resto de la lógica***, que son a los que se accede desde la interfaz de usuario, y por último *scriplets que implementan la interfaz de usuario* HTML.

Clases de almacenamiento:

- Area,
- Recurso,
- Propiedad,
- ElemLista,
- Persona,
- Residencia,
- Doc,
- Consumidor,
- Atributo,
- Match,
- MatchGlobal y
- Criterio.

Clases de acceso a datos:

- DatosBdTFC,
- GestorBd,
- GestorRecursos,
- GestorConsumidores,
- GestorUsuarios y
- GestorMatches.

Scriplets que implementan la capa de negocio:

- login.jsp,
- borrar_rec.jsp,
- activar_rec.jsp,
- borrar_prop.jsp,
- guardar_rec.jsp,
- guardar_elem_prop.jsp,
- guardar_prop.jsp,
- borrar_elem_prop.jsp,
- guardar_cons.jsp,
- borrar_cons_rec.jsp y
- apuntar_cons_rec.jsp

Scriplets que implementan la interfaz de usuario:

- index.jsp,
- index_admin.jsp,
- modif_rec.jsp,
- modif_prop.jsp,
- index_cons.jsp,
- nuevo_cons.jsp,
- modif_cons.jsp,
- intro_atrib.jsp,
- asignar.jsp y
- error.jsp.

2.5 Clases de persistencia

La aplicación se puede dividir en tres grandes áreas: gestión de recursos, gestión de consumidores y gestión de matching.

En el diseño de las clases, estas tres áreas están subdivididas en dos bloques cada una: clases de persistencia y clases de acceso a datos.

La gestión de la persistencia en una base de datos suele tener dos variantes: o bien las propias clases de persistencia acceden a la base de datos (o el sistema que haya sido elegido para la persistencia) o bien se crean clases adicionales que implementan estos accesos. En este caso se ha elegido la segunda opción, que facilita que todo el acceso a datos se haga con unas pocas clases.

Asimismo, la capa de negocio (implementada con scriplets JSP) se encarga de utilizar las clases de persistencia y las clases de acceso a datos, interactuando además con la interfaz de usuario.

Diagrama UML simplificado

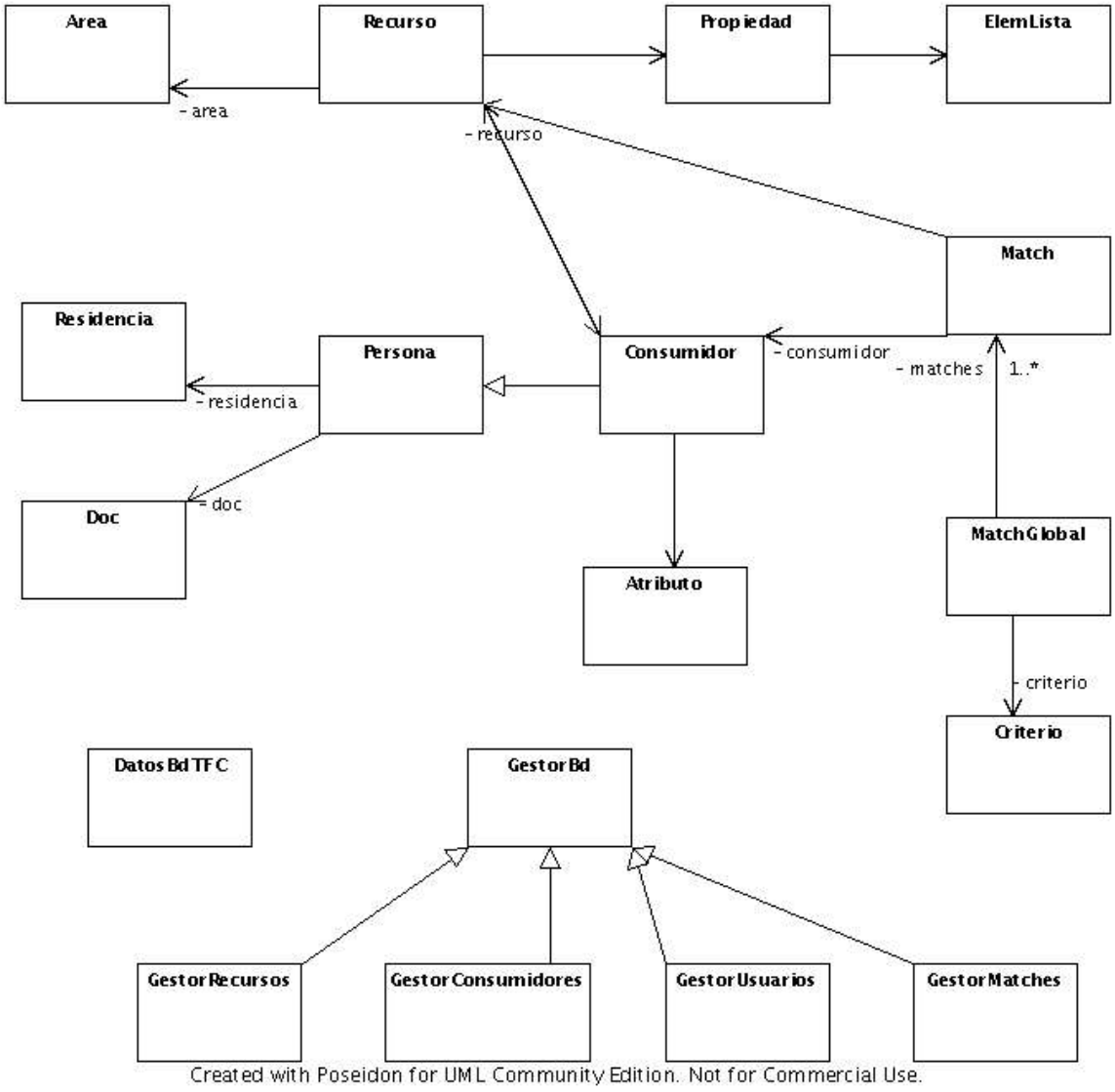


Diagrama UML completo del área de Recursos

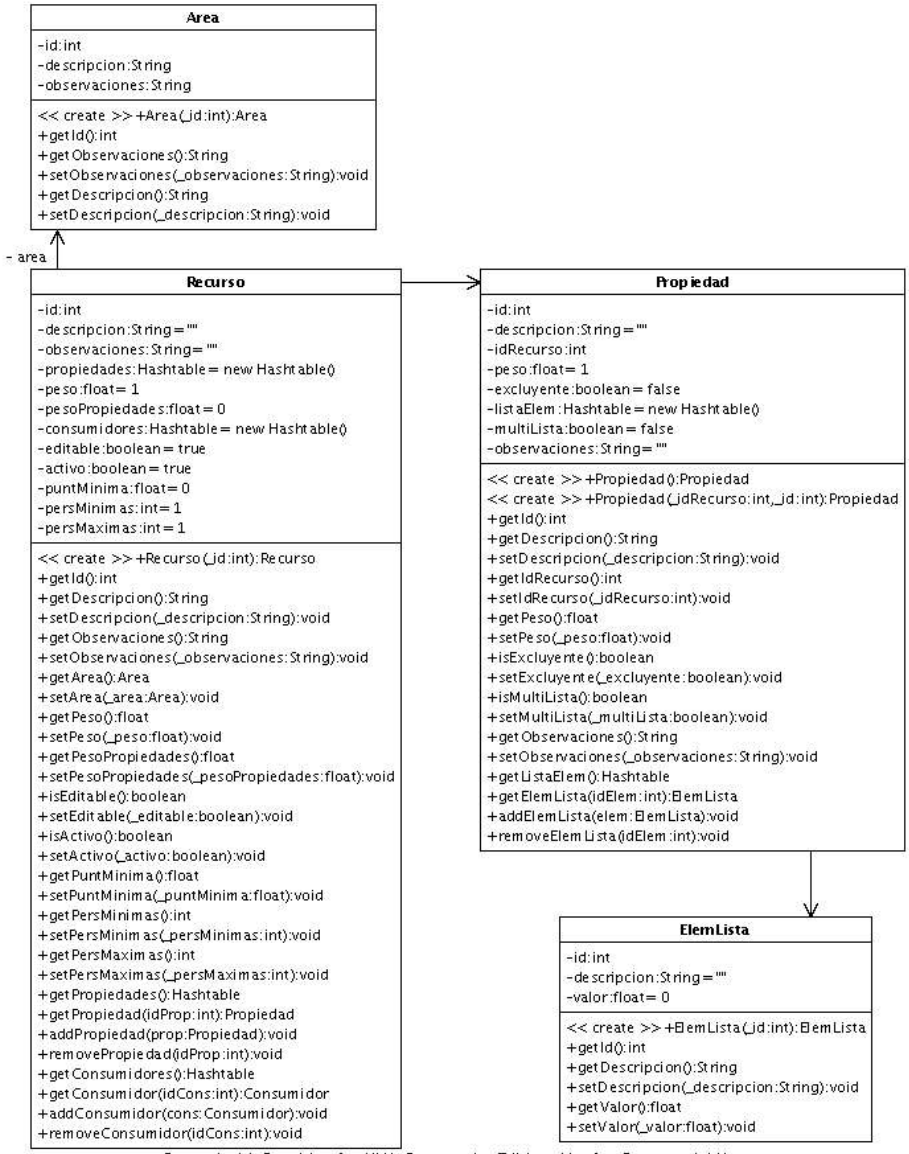


Diagrama UML completo del área de Consumidores

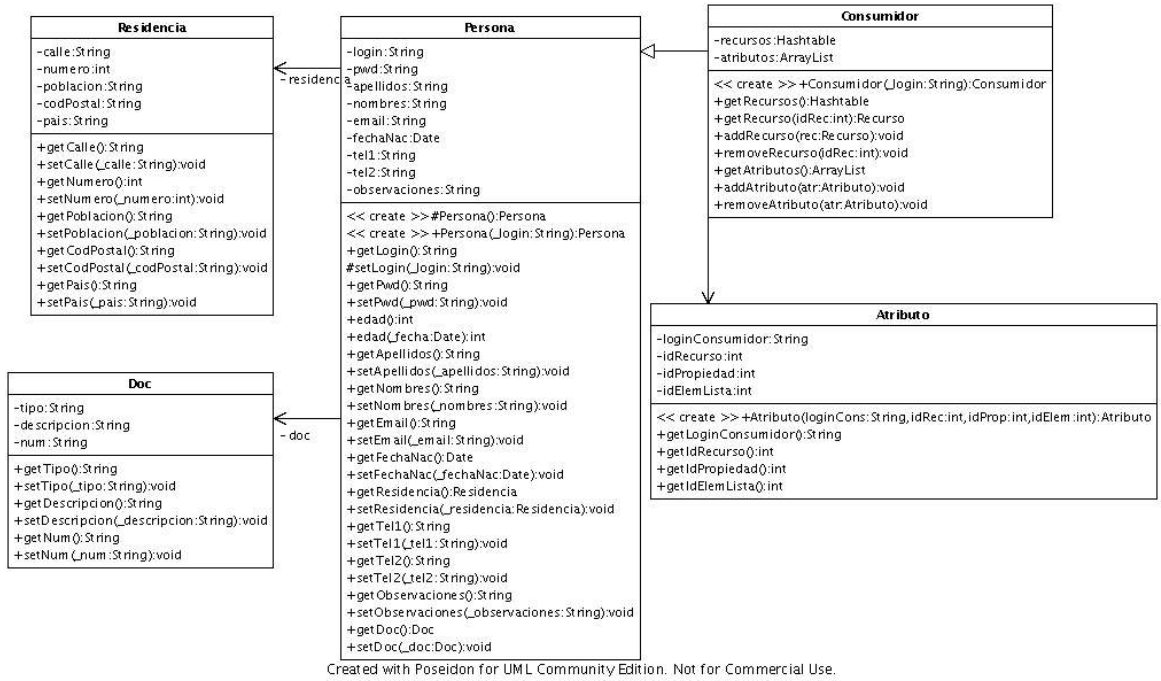


Diagrama UML completo del área de Matches

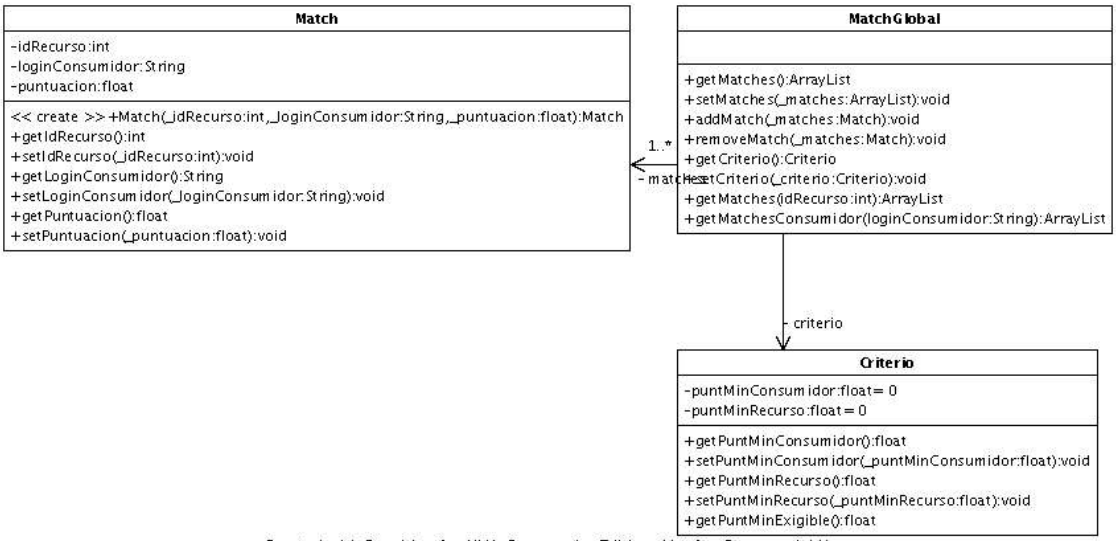
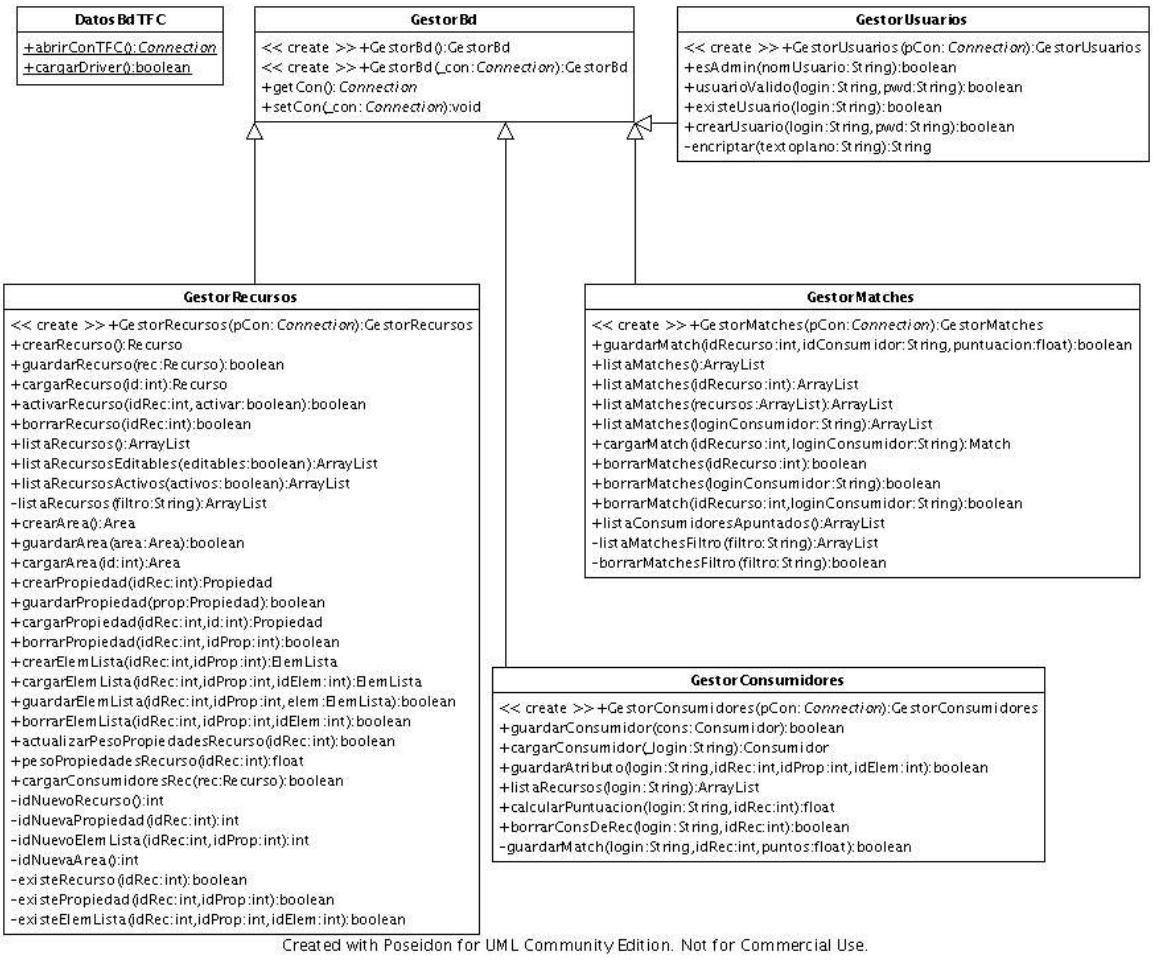


Diagrama UML completo del área de acceso a datos



Descripción de las clases

Area

Clase que puede almacenar los datos de un área. Sólo tiene atributos y sus métodos accesores. Dado que la gestión de áreas no se ha implementado, se utiliza poco.

Recurso

Clase que puede almacenar los datos de un recurso, incluido en un área. Se compone de atributos y métodos accesores, incluyendo el acceso a algunas listas. Algunos de sus atributos no se utilizan, pero podrían ser útiles en otra forma de implantación (lista de consumidores del recurso).

Propiedad

Clase que puede almacenar los datos de una propiedad, incluida en un recurso. Se compone de atributos y métodos accesores, incluyendo el acceso a la lista de elementos que pertenecen la propiedad.

ElemLista

Clase que puede almacenar los datos de un elemento de lista, incluido en una propiedad. Se compone de atributos y métodos accesores.

Consumidor

Hereda de persona. Clase que puede almacenar algunos datos específicos, relacionado con sus atributos y recursos. Se compone de atributos y métodos accesores. No todo lo implementado se utiliza (listas de atributos y recursos del consumidor).

Persona

Clase que puede almacenar los datos personales de una persona. Se compone de atributos y métodos accesores.

Residencia

Clase que puede almacenar los datos de residencia de una persona. Se compone de atributos y métodos accesores.

Doc

Clase que puede almacenar los datos de documento de una persona. Se compone de atributos y métodos accesores.

Atributo

Clase que puede almacenar un elemento perteneciente a una propiedad de un recurso, elegido por un consumidor. Se compone de atributos y métodos accesores.

Match

Clase que puede almacenar la puntuación de un consumidor, obtenida al apuntarse a un recurso. Se compone de atributos y métodos accesores.

MatchGlobal

Clase que puede almacenar datos de matches y criterios. Se compone de atributos y métodos accesores. Esta clase no se utiliza, dado que no se ha implementado la gestión de plantillas de matching.

Criterio

Clase que puede almacenar la definición de las restricciones de un matching. Se compone de atributos y métodos accesores.

DatosBdTFC

Contiene todos los datos necesarios para poder conectarse a la base de datos utilizada, incluyendo el controlador, el nombre de la base de datos, el usuario y contraseña. Asimismo, se conecta y devuelve una conexión ya establecida. Si se deseara cambiar de SGBD, ésta es la clase en la cual se deberían hacer cambios.

GestorBd

Clase padre de las demás que acceden a la base de datos (excepto DatosBdTFC).

Contiene un objeto conexión.

GestorRecursos

Clase que gestiona el acceso a la base de datos para el área de recursos. Permite crear, cargar y guardar los datos de un recurso con sus propiedades y elementos, borrar un recurso, una propiedad o un elemento, cargar listas de recursos editables o activos, etc.

GestorConsumidores

Clase que gestiona el acceso a la base de datos para el área de consumidores. Permite cargar, guardar o borrar un consumidor y sus atributos, así como calcular la puntuación obtenida.

GestorUsuarios

Clase que gestiona todo lo relacionado con los usuarios del sistema, incluyendo el acceso a la base de datos. Valida y crea los usuarios, e identifica al administrador.

GestorMatches

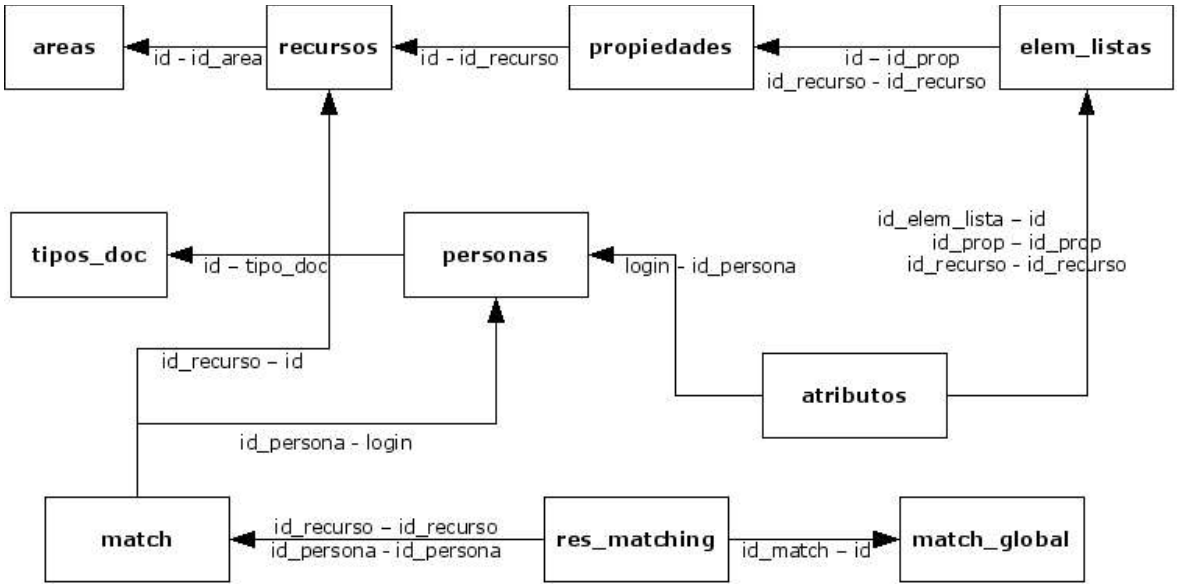
Clase que gestiona el acceso a la base de datos para el área de matches. Permite cargar, guardar o borrar matches, así como obtener listas de matches según diferentes criterios.

2.6 Base de datos

La base de datos ha sido diseñada teniendo en cuenta que fuera apta para los requerimientos, aunque algunas funcionalidades no fueran a ser implementadas durante el TFC.

Esquema de la base de datos

En el esquema figuran las tablas y sus relaciones de claves foráneas. La flecha indica la dirección de la relación (el origen es la tabla que tiene definida la clave foránea), figurando los nombres de los campos relacionados (el orden de los campos varía según la posición de las tablas).



Definición de las tablas y campos

areas

Campo	Tipo	Tamaño	PK	FK	Observaciones
id	int	2	x		
descripcion	varchar	50			
observaciones	text				

Áreas habilitadas. Se utilizan para agrupar los recursos.

recursos

Campo	Tipo	Tamaño	PK	FK	Observaciones
id	int	2	x		
descripcion	varchar	50			
observaciones	text				
id_area	int	2		areas.id	Área a la que pertenece el recurso.
peso	float	2			Peso relativo de cada recurso.
peso_prop	float	4			Peso total de las propiedades del recurso.
punt_minima	float	4			Puntuación mínima requerida para ser asignado a este recurso.
activo	bool	1			Indica si el recurso está activo.

Campo	Tipo	Tamaño	PK	FK	Observaciones
editable	bool	1			Indica si el recurso está disponible para los consumidores.
pers_minimas	int	2			Mínimo de consumidores a asignar a este recurso.
Pers_maximas	int	2			Máximo de consumidores a asignar a este recurso.

Lista de recursos datos de alta.

propiedades

Campo	Tipo	Tamaño	PK	FK	Observaciones
id	int	2	x		
descripcion	varchar	50			
observaciones	text				
id_recurso	int	2	x	recursos.id	Recurso al que pertenece.
peso	float	2			Indica la importancia de la propiedad.
excluyente	bool	1			Indica si esta propiedad es excluyente.
multi_lista	bool	1			Indica si esta propiedad admite más de un valor seleccionado.

Propiedades asignadas a cada recurso.

elem_listas

Campo	Tipo	Tamaño	PK	FK	Observaciones
id	int	2	x		
descripcion	varchar	50			
valor	float	2			Valor de este elemento. Para obtener la puntuación de un consumidor, se suman estos valores.
id_prop	int	2	x	propiedades.id	Propiedad a la que pertenece.
id_recurso	int	2	x	propiedades.id_recurso	Recurso al que pertenece la propiedad.

Elementos que componen cada propiedad de cada recurso.

personas

Campo	Tipo	Tamaño	PK	FK	Observaciones
num_doc	varchar	15			Número de documento.
tipo_doc	varchar	3		tipos_doc.id	Tipo de documento.
apellidos	varchar	50			
nombres	varchar	30			
email	varchar	255			
fecha_nac	date				
login	varchar	12	x		Login de acceso al sistema. Identifica al usuario.
contrasenya	varchar	32			Contraseña encriptada.
calle	varchar	60			
numero	varchar	10			
poblacion	varchar	30			
cod_postal	varchar	10			
pais	varchar	30			
tel1	varchar	15			
tel2	varchar	15			
tel3	varchar	15			
observaciones	text				

Datos personales de las personas dadas de alta en el sistema, posibles consumidores.

tipos_doc

Campo	Tipo	Tamaño	PK	FK	Observaciones
id	varchar	3	x		
descripcion	varchar	20			

Tipos de documento: DNI, Pasaporte, NIE, etc.

atributos

Campo	Tipo	Tamaño	PK	FK	Observaciones
id_persona	varchar	12	x	personas.login	Identificación del consumidor.
id_recurso	int	2	x	elem_listas.id_recurso	Recurso al que pertenece.
id_prop	int	2	x	elem_listas.id_prop	Propiedad a la que pertenece.
id_elem_lista	int	2	x	elem_listas.id	Identificación del elemento de la lista.

Contiene, para cada consumidor de cada recurso, los elementos seleccionados por dicho consumidor. Si la propiedad no es multi_lista, sólo debería haber 1 id_prop para ese recurso de esa persona. Si la propiedad sí es multi_lista, puede haber más de un id_elem_lista distinto para cada id_prop de cada recurso de cada persona.

match

Campo	Tipo	Tamaño	PK	FK	Observaciones
id_persona	varchar	12	x	personas.login	Identifica el consumidor.
id_recurso	int	2	x	recursos.id	Recurso al que pertenece.
puntuacion	float	2			Puntuación del consumidor para este recurso.

Almacena la puntuación de cada persona para cada recurso. Es la tabla que se utiliza para calcular el matching (asignar los consumidores a los recursos).

match_global

Campo	Tipo	Tamaño	PK	FK	Observaciones
id	int	1	x		
punt_min_recurso	int	2			Puntuación mínima exigida que debe cumplir un consumidor para ser asignado a este recurso.
punt_min_persona	int	2			Puntuación mínima exigible a un consumidor para ser asignado a cualquier recurso.

Almacena la definición de una clasificación o matching. Su utilidad principal consiste en guardar la configuración de un matching, aunque estos criterios se podrían definir en el momento de realizar el matching. Permite guardar diferentes matching con distintos criterios.

res_matching

Campo	Tipo	Tamaño	PK	FK	Observaciones
id_match	int	1	x	match_global.id	Identificación de la definición del matching.
id_persona	varchar	12	x	match.id_persona	Identifica la persona.
id_recurso	int	2	x	match.id_recurso	Identifica el recurso.

Almacena el resultado de un matching, donde cada consumidor ha sido asignado a un recurso.

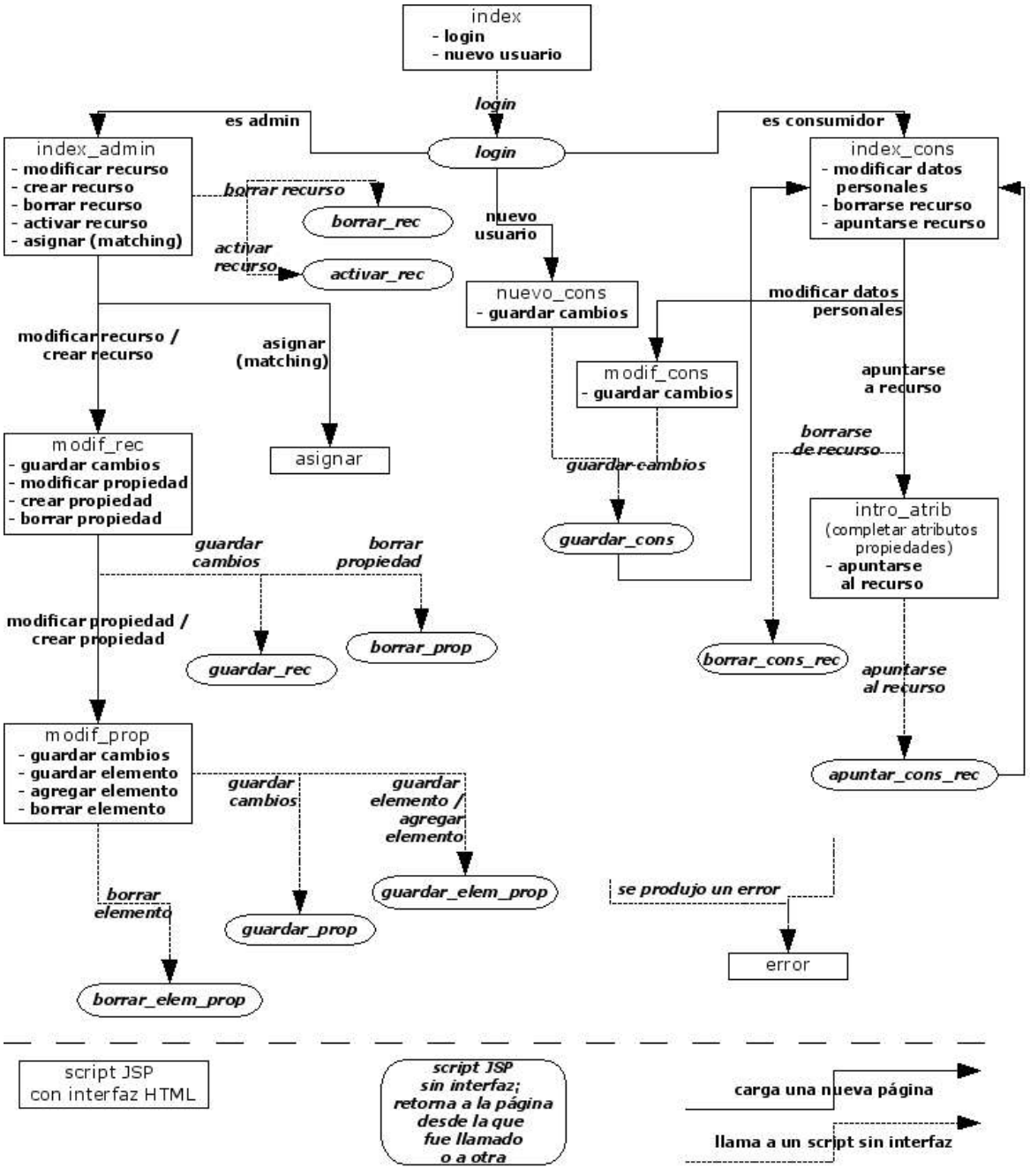
2.7 Interfaz de usuario

Al tratarse de una aplicación web, la interfaz de usuario está contenida en archivos JSP que interactúan entre ellos.

Algunos de estos archivos JSP definen específicamente diferentes páginas web

que son los formularios de interacción del usuario con la aplicación, mientras que otros scriplets JSP pertenecen en realidad a la capa de negocio, aunque interactúan directamente con los scriplets de interfaz.

Diagrama de los scriplets JSP (formularios web y capa de negocio)

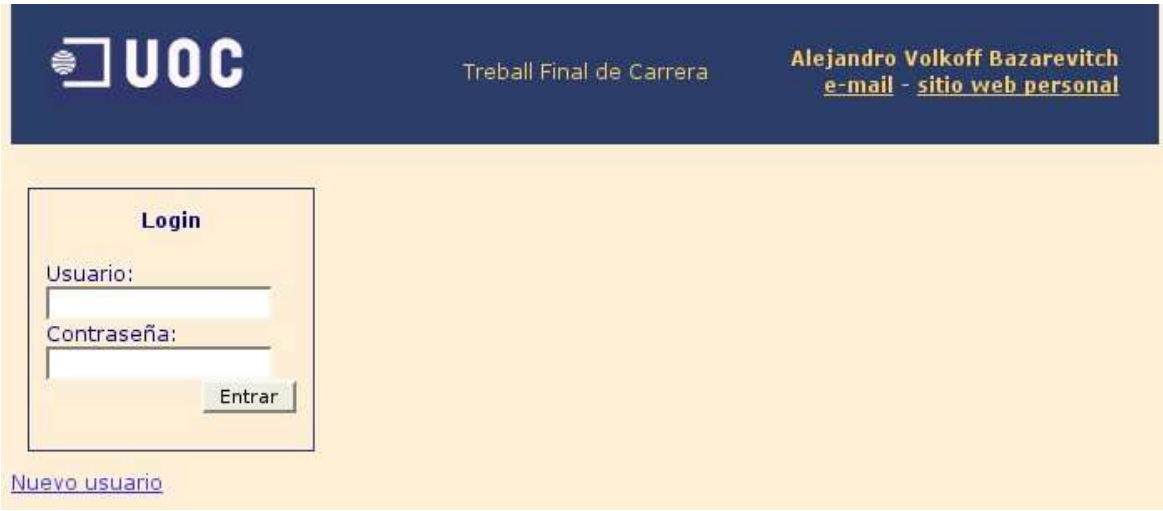


Descripción de los scriplets

index.jsp

Página de entrada. Permite identificarse como usuario o darse de alta como

usuario nuevo.



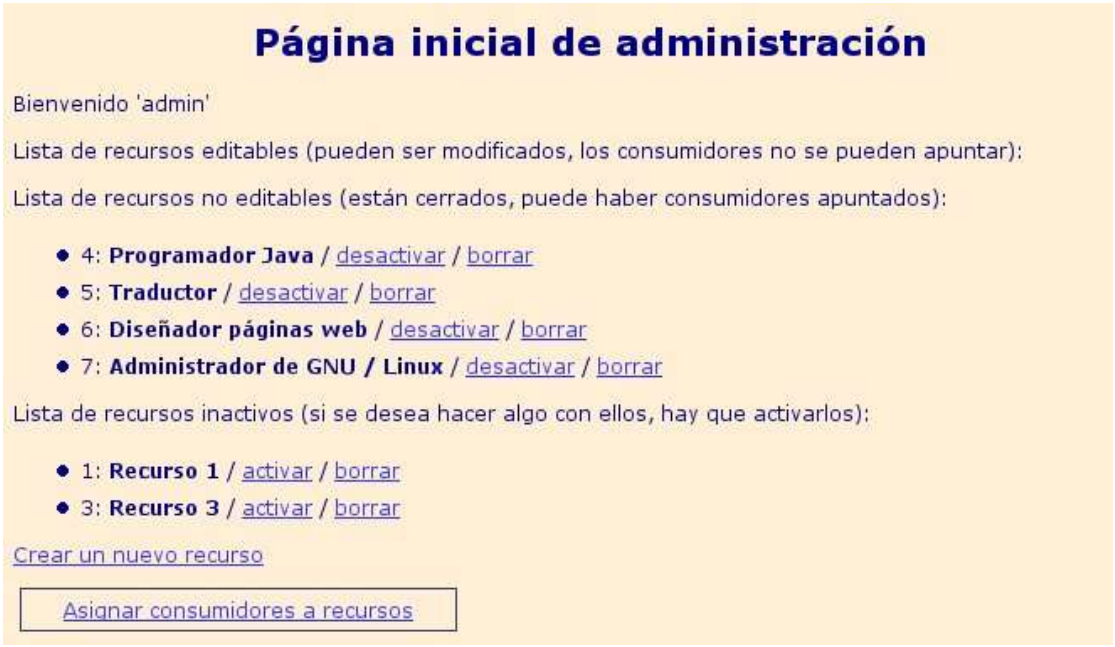
login.jsp *(sin interfaz)*

Valida el login y contraseña del usuario contra la base de datos. Si no es válido, devuelve a la página principal. Si es válido, mira si es administrador o consumidor.

Si es administrador, lo envía a la página de administración; si es consumidor, lo envía a la página de usuarios.

index_admin.jsp

Página de administración. Muestra todos los recursos disponibles, clasificados en: “editables”, “no editables” e “inactivos”. Permite modificar, borrar o desactivar los primeros, borrar o desactivar los segundos y borrar o activar los inactivos.



borrar_rec.jsp *(sin interfaz)*

Borra un recurso de la base de datos, y devuelve a la página de administración. Según como se implementen las restricciones en la base de datos, al borrar un recurso puede ocurrir que se borren todas sus propiedades y consumidores apuntados (borrado en cascada) o bien, si hay consumidores apuntados, que no se permita borrar el recurso.

activar_rec.jsp *(sin interfaz)*

Activar o desactivar (según un parámetro) un recurso en la base de datos. Devuelve a la página de administración.

modif_rec.jsp

Página para modificar un recurso, o bien para crearlo (según parámetros). Muestra una lista con todos los atributos del recurso (descripción, peso, etc.) más una lista de sus propiedades. Permite cambiar cualquier atributo y guardar los cambios, modificar o borrar una propiedad, o bien crear una propiedad nueva.

guardar_rec.jsp *(sin interfaz)*

Guarda un recurso en la base de datos, con los datos del formulario que ha llamado a este scriplet. Devuelve a la página de modificación de recursos.

borrar_prop.jsp *(sin interfaz)*

Borra una propiedad de un recurso, junto con sus elementos asociados. Devuelve a la página de modificación de recursos.

modif_prop.jsp

Página para modificar o crear una propiedad (según parámetros). Muestra los atributos básicos de una propiedad (descripción, peso, etc.) más una lista de los elementos que la componen. Permite cambiar cualquier atributo y guardar los cambios.

Los elementos se pueden modificar, crear o borrar en la misma página, con un botón que guarda los cambios.

guardar_prop.jsp *(sin interfaz)*

Guarda los datos de una propiedad en la base de datos. Devuelve a la página de modificación de propiedades.

guardar_elem_prop.jsp *(sin interfaz)*

Guarda los datos de un elemento de una propiedad. Devuelve a la página de modificación de propiedades.

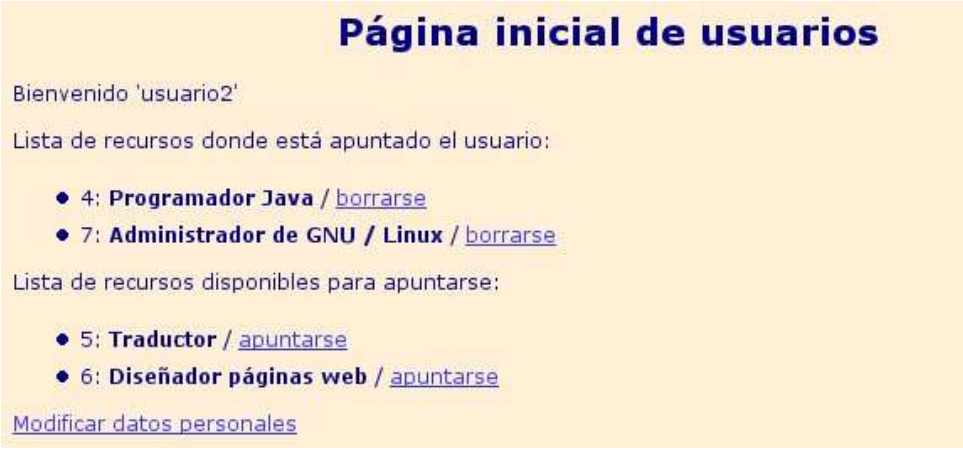
borrar_elem_prop.jsp *(sin interfaz)*

Borra un elemento de una propiedad. Devuelve a la página de modificación de propiedades.

index_cons.jsp

Página de usuarios. Muestra algunos datos del usuario, los recursos en los cuales está apuntado y una lista de los recursos disponibles para apuntarse.

Permite modificar los datos personales, borrarse de un recurso al cual está apuntado o apuntarse a un nuevo recurso. En este último caso, es llevado a una página donde deberá completar los requerimientos de ese recurso.



nuevo_cons.jsp

Se utiliza para dar de alta un usuario en el sistema. Muestra un formulario con los datos personales del usuario, para que éste los complete.

modif_cons.jsp

Muestra un formulario con los datos personales del usuario. Permite cambiarlos y guardar los cambios.

guardar_cons.jsp *(sin interfaz)*

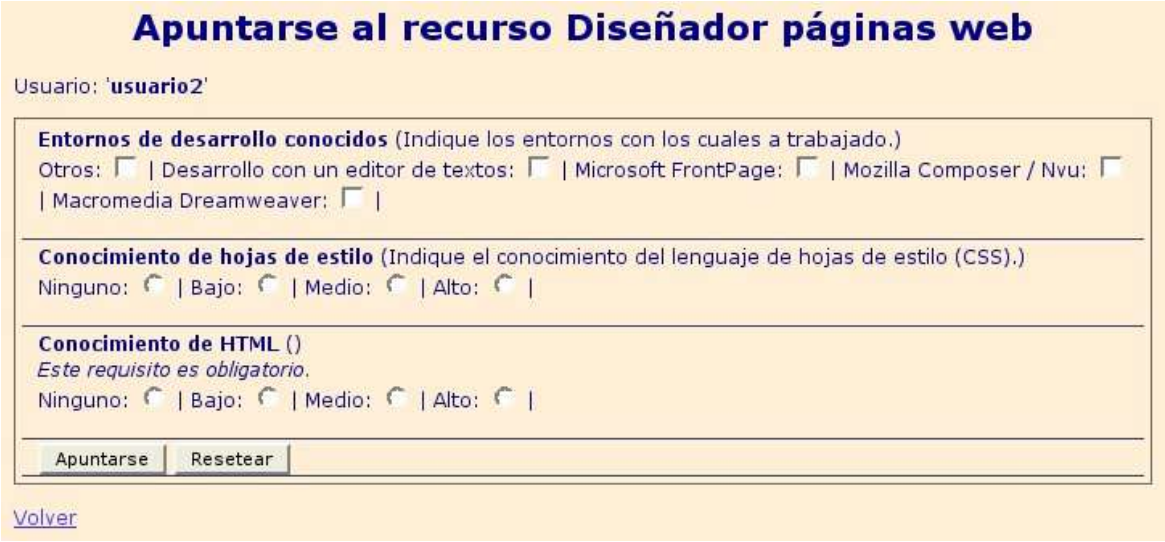
Guarda los cambios en los datos personales de un usuario (ya sea existente o nuevo). Devuelve a la página de usuarios.

borrar_cons_rec.jsp *(sin interfaz)*

Borra un consumidor de un recurso. Devuelve a la página de usuarios.

intro_atrib.jsp

Muestra un formulario con todas las propiedades que componen un recurso, y sus elementos respectivos. El usuario debe ir eligiendo los elementos que cumple de las opciones que se le presentan, y finalmente confirmar que desea apuntarse al recurso.

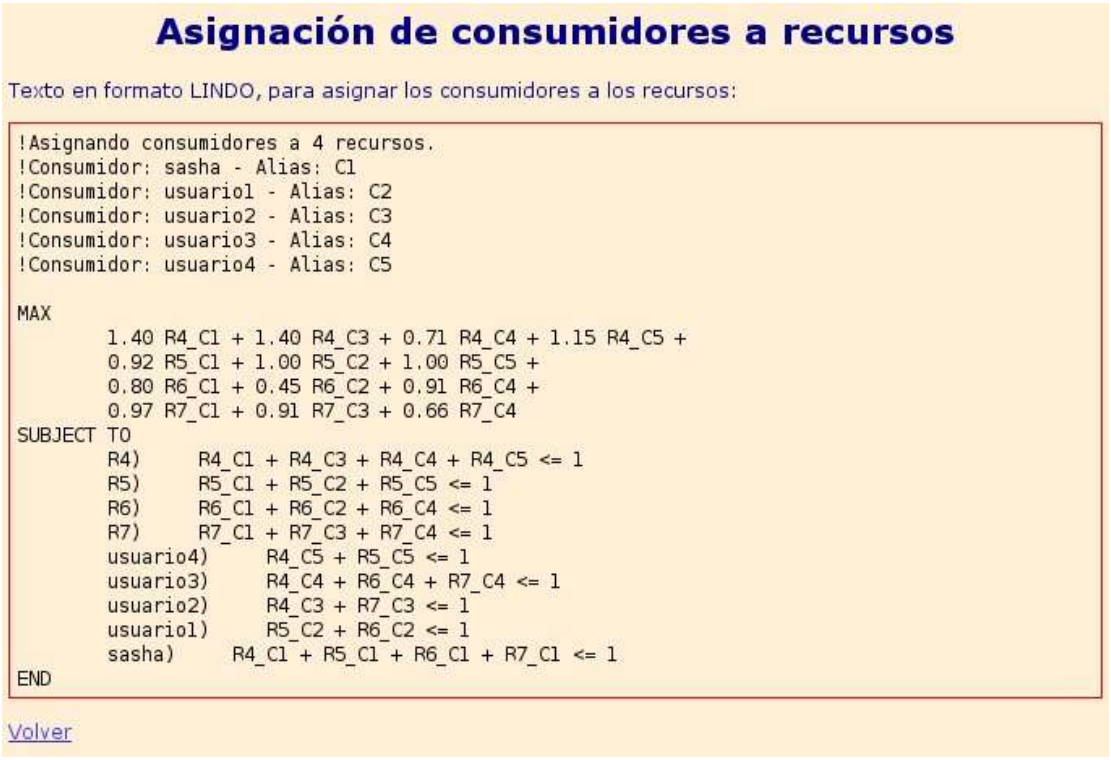


apuntar_cons_rec.jsp *(sin interfaz)*

Apunta un usuario a un recurso. Primero verifica las opciones introducidas por el usuario, para ver si debe ser excluido. En caso que no lo sea, calcula la puntuación del usuario y almacena este dato en la base de datos. También almacena todas las opciones elegidas por el usuario. Devuelve a la página de usuarios.

asignar.jsp

Elabora un texto en formato Lindo para ser procesado y calculada la función maximizadora, según las restricciones existentes, a partir de los consumidores que hay apuntados a los recursos.



error.jsp

Clase genérica para mostrar errores. Cuando se produce algún error, se invoca a esta clase con un parámetro, indicando el mensaje a mostrar.

3 Funcionamiento de la aplicación

3.1 Procedimiento

1) Un **administrador de recursos** define unos recursos, que son los que se intentarán completar con los consumidores-usuarios.

Para ello, primer debe crear el recurso y, en ese mismo acto, puede crear una o más propiedades para dicho recurso.

Posteriormente se puede modificar el recurso así como sus propiedades.

Cada vez que se modifique las propiedades de un recurso, el sistema debe sumar el peso de todas las propiedades y almacenarlo junto con los datos del recurso; este peso de las propiedades se utilizará posteriormente, cuando los consumidores se vayan apuntando, para normalizar la puntuación del consumidor para dicho recurso.

Una vez que se ha finalizado la definición del recurso, se lo debe marcar como "no editable", para que pase a estar disponible para los consumidores.

Cuando se desea quitar un recurso, se lo puede eliminar (eliminándose todas sus propiedades) o bien marcarlo como "inactivo". Esto permite mantener almacenada su definición, para un uso posterior.

2) Un **consumidor-usuario**, se da de alta en el sistema con sus datos personales.

A continuación, puede apuntarse a alguno de los recursos ofrecidos, definiendo una serie de atributos. Estos atributos serán solicitados por el sistema, según sea necesario para los recursos ofrecidos; por ejemplo, datos personales (edad, lugar de residencia, sexo, etc.), intereses (le gusta el deporte, le gustan los niños, etc.), capacidades (sabe conducir, sabe informática, conoce un idioma, etc.).

Al finalizar la entrada de los atributos, se calculará una puntuación (entre 0 y 100%), mediante la cual se va a valorar la relación entre cada consumidor y cada recurso. Esta puntuación es la que luego se va a utilizar para establecer el matching.

Si el recurso tiene alguna propiedad marcada como excluyente, y el consumidor no la satisface, su puntuación global será 0, quedando excluido de la futura selección.

Durante la puntuación, el sistema va vinculando cada atributo de cada

consumidor con cada propiedad de cada recurso. Por ejemplo, si hay un recurso ofrecido cuyas propiedades son: población (= Barcelona), edad (entre 20 y 40 años) y área (salud), al consumidor se le preguntará su lugar de residencia (o bien, en este caso, se le preguntará si reside en Barcelona, como condición excluyente), su edad (o bien, igualmente, si tiene una edad entre 20 y 40 años) y si le interesa el área de la salud.

3) Un **administrador de matching** (que en la implementación actual es el mismo que el administrador de recursos), una vez puntuadas las relaciones entre consumidores y recursos, ejecutará un matching, atendiendo a los criterios definidos. Estos criterios incluyen el número de consumidores mínimo y máximo de cada recurso (por defecto 1), el número de recursos en los cuales se puede apuntar a un consumidor, y una puntuación mínima exigible para cada recurso.

3.2 Entrada en la aplicación

Al entrar a la página principal (*index.jsp*), se presenta un formulario donde el usuario debe validarse con su nombre y contraseña. Según se trate de un usuario normal o un administrador, será redirigido al área de usuarios-consumidores (*index_cons.jsp*) o al área de administración (*index_admin.jsp*).

3.3 Administración de recursos y matching

El área de administración (*index_admin.jsp*) presenta una lista con todos los recursos, subdivididos en tres áreas: recursos editables (o modificables), recursos no editables (o cerrados) y recursos inactivos.

En el caso de los recursos editables, éstos se pueden modificar o borrar.

En el caso de los recursos no editables, éstos se pueden desactivar o borrar.

En el caso de los recursos inactivos, éstos se pueden activar o borrar.

Además, esta página presenta un acceso a la asignación de consumidores y recursos (o sea, a la realización del matching) (*asignar_rec.jsp*).

Si se pretende modificar un recurso, se accede a una página (*modif_rec.jsp*) donde se presentan todos los atributos básicos del recurso, y la lista de propiedades que tiene asignada. Los atributos básicos se pueden modificar directamente. En cuanto a las propiedades, éstas pueden ser modificadas o borradas, o bien se puede crear una nueva propiedad.

Si se desea crear o modificar una propiedad, se accede a una página (*modif_prop.jsp*) donde se presentan los atributos básicos de esa propiedad y una lista de los elementos que la componen. Tanto los atributos básicos como

los elementos pueden ser modificados directamente, o también se puede insertar directamente un nuevo elemento.

3.4 Consumidores-Usuarios

El área de consumidores (*index_cons.jsp*) presenta una página con una lista de los recursos a los cuales el consumidor está apuntado, y otra lista de los recursos a los cuales se puede apuntar.

Además, hay un enlace para modificar los datos personales del usuario (*modif_cons.jsp*).

Al lado de cada recurso al cual el consumidor está apuntado hay un enlace que permite borrarse de ese recurso.

Al lado de cada recurso disponible hay un enlace que permite apuntarse a ese recurso (*intro_atrib.jsp*).

En la página para apuntarse a un recurso, se presenta la lista de propiedades de dicho recurso, con los elementos de cada una de ellas, para que el usuario los complete y se apunte al recurso.

3.5 Gestión de errores

En las clases que acceden a la base de datos, todos los errores que se producen son lanzados hacia el nivel superior.

En los scriplets no visuales, que implementan la capa de negocio, se controlan estos errores, redirigiéndose a un scriplet genérico de gestión de errores (*error.jsp*), que muestra un mensaje de error.

Adicionalmente, cada scriplet visual muestra un mensaje si se le pasa como parámetro. Este mensaje puede ser un error en los datos de un formulario, un aviso o una confirmación de la acción realizada.

4 Plan de trabajo

El plan de trabajo se desarrolló según etapas esquemáticas, aunque algunas de estas se superpusieron, y a veces fue necesario volver atrás para resolver problemas que se fueron presentando durante la implementación.

4.1 Análisis

Primero se efectuó un análisis de lo que implicaba la aplicación, definiendo sus aspectos más relevantes: almacenamiento de los datos, áreas principales, elementos que componen el sistema, tecnologías a utilizar.

Posteriormente, se hizo un análisis de los casos de uso, esquematizando el resultado del análisis anterior.

También se hizo un esquema elemental de cuál sería el funcionamiento de la interfaz de usuario, partiendo del diagrama de clases de uso.

4.2 Diseño orientado a objetos

En esta etapa se hizo el diseño de las clases que compondrían el sistema, en las tres grandes áreas: recursos, consumidores y matching.

Todo esto se plasmó en la primera versión de un diagrama de clases UML, realizado con el programa PoseidonUML. En sucesivas versiones, este diagrama fue sufriendo pequeños cambios.

4.3 Diseño de la base de datos

El diseño de la base de datos se realizó partiendo de las clases especificadas, duplicando prácticamente los atributos de esas clases.

4.4 Pruebas tecnológicas

Esta fue una etapa bastante dura, debido a mi desconocimiento de la plataforma Java para web, del servidor Apache Tomcat y de PostgreSQL.

Todas las pruebas se realizaron en un sistema Debian Linux, versión Sarge.

La implementación de la base de datos sobre PostgreSQL no representó dificultades, aprovechando la interfaz del programa pgAdmin que facilitó la tarea.

Después instalé el servidor Apache Tomcat 4 y verifiqué su correcto funcionamiento.

A continuación instalé el entorno de desarrollo NetBeans 4. Allí comprobé que este entorno ignoraba mi servidor Tomcat ya instalado, y en cambio instaló el suyo propio (versión 5), del cual se sirve.

Realicé unas pruebas básicas del conjunto (página JSP que se compila, se ejecuta en el servidor Tomcat y puede acceder a la base de datos PostgreSQL) y comprobé que todo funcionaba.

4.5 Implementación de la capa de datos

Partiendo del diagrama UML, el programa PoseidonUML me permitió guardar el código resultante, el cual sirvió de base para el resto de la implementación.

Lamentablemente, el PoseidonUML no contempla ciertas clases de la API de Java (principalmente las de acceso SQL) y en general produce ciertos defectos en el código resultante, el cual hubo de ser bastante retocado.

Para la implementación del acceso a la base de datos tuve una dura batalla con la gestión de usuarios de PostgreSQL, aunque al final conseguí que funcionara (sin saber bien cómo ni por qué).

Las clases de persistencia no representaron mucho problema dado que se componen básicamente de atributos y sus métodos de acceso.

Las clases de acceso a datos representaron más trabajo, aunque nada fuera de lo común.

4.6 Implementación de la capa de negocio

Esta etapa y la siguiente han corrido casi paralelas, ya que he ido aprendiendo JSP a medida que iba avanzando.

En realidad la capa de negocio es bastante fina, dado que esta aplicación se basa en un buen diseño de la base de datos y en unos cuantos formularios web de interfaz con el usuario (tanto administrador como consumidor).

4.7 Implementación de la interfaz de usuario

Me hubiera gustado descubrir una forma de separar claramente el código HTML de la lógica de la aplicación, pero no me fue posible (no sé si por mi desconocimiento de JSP o porque realmente no es fácil).

En cuanto al diseño en sí de las páginas no tuve ningún problema.

5 Tecnología utilizada

Esta es la tecnología utilizada para desarrollar la aplicación y hacer las pruebas. No obstante, y atendiendo al diseño, estas decisiones no son las únicas posibles. En algunos casos, sería fácil cambiar una tecnología por otra (por ejemplo, en el caso del SGBD o el servidor web).

Entre las alternativas, he contemplado únicamente software libre o, al menos, gratuito.

5.1 Almacenamiento: base de datos relacional

¿Por qué base de datos relacional?

Dado el tiempo de que disponía para el TFC, elegí trabajar con base de datos porque es lo que mejor conozco y menos problemas me causaría. Además, porque funciona perfectamente para esta aplicación.

Otras posibles opciones

XML. Esta hubiera sido una excelente alternativa, ya que el volumen de datos no es grande, no se han utilizado funciones específicas de base de datos como procedimientos almacenados o disparadores (aunque se podrían utilizar) y las claves foráneas se podrían haber gestionado directamente desde el código.

No elegí esta opción porque he tenido experiencia en el manejo de archivos XML y lo he encontrado bastante farragoso (mi experiencia no es con Java sino con C# y la plataforma .NET). Hubiera necesitado mucho tiempo para hacer funcionar partes del código, que tal vez no fueran las más importantes desde el punto de vista del TFC, que era el diseño del sistema.

Adaptación a otras tecnologías

Si se deseara trabajar con XML, habría que diseñar todos los formatos necesarios, aunque se podría aprovechar tal vez el diseño de la base de datos. En cuanto al código, habría que reescribir todas las clases que acceden a datos.

5.2 Servidor de base de datos: PostgreSQL

¿Por qué PostgreSQL?

Elegí PostgreSQL por diversas razones: no lo conocía y quería aprender; siempre he oído que es un excelente servidor, y tiene todas las funcionalidades que se esperan de un buen servidor de base de datos: permite definir restricciones en

las tablas y en las relaciones con claves foráneas, tiene procedimientos almacenados y disparadores.

Otras posibles opciones

La principal alternativa era MySql, pero carece de algunas funcionalidades (al menos las versiones que yo conozco): definir restricciones en las tablas y en las relaciones con claves foráneas, procedimientos almacenados y disparadores.

Adaptación a otras tecnologías

Para adaptar la aplicación a otros servidores de base de datos no debería haber mayor problema: el acceso a la base de datos está incluido en una sola clase (DatosBdTFC), con lo cual sólo modificando allí sería suficiente.

En cuanto a la creación de las tablas, junto con el TFC se entregan las sentencias SQL necesarias, siendo necesario hacer algunas modificaciones en el tipo de datos, cuyo nombre varía de un servidor a otro.

Las sentencias SQL que se utilizan en el código de la aplicación son muy básicas y conformes al estándar ANSI.

5.3 Motor: Clases Java

¿Por qué Java?

En primer lugar, porque es la plataforma utilizada en la UOC. En segundo lugar, porque puede correr en varios sistemas operativos, porque está orientado a objetos, porque tiene una excelente API...

Otras posibles opciones

En este caso, el motor va muy ligado a la interfaz de usuario, por lo que habría que buscar alternativas a ambas.

Adaptación a otras tecnologías

Sería necesario reescribir todo el código, aunque se podría aprovechar el diseño de las clases. Dada la similitud del lenguaje Java con C# , tal vez se podría aprovechar parte del código.

5.4 Interfaz de usuario: web (JavaServer Pages - JSP)

¿Por qué JavaServer Pages?

Los motivos son los mismos que en la elección del motor Java.

Otras posibles opciones

Dejando de lado el hecho de que Java sea la plataforma elegida por la UOC, se podría haber utilizado PHP o ASP. Con PHP tengo bastante experiencia y seguramente me hubiera costado menos la implementación que con Java y JSP. En cuanto a ASP, no lo conozco mucho, aunque mi experiencia con las plataformas de Microsoft es bastante negativa.

Otro motivo para elegir Java o PHP es que ambos se complementan perfectamente con el servidor web Apache, que también es multiplataforma, además de ser el servidor web más utilizado actualmente.

Adaptación a otras tecnologías

Dada la similitudes entre JSP y ASP, quizás se podría reaprovechar parte del código, ya que el código HTML quedaría prácticamente sin cambios.

Si se quisiera pasar a PHP, habría que reescribir todo el código.

En cuanto al aspecto estético de la web, la mayoría está dictado por hojas de estilos almacenadas aparte, por lo que cambiando estas se podría cambiar el aspecto general.

5.5 Servidor web: Apache Tomcat 4

¿Por qué Tomcat?

Porque es el servidor web más utilizado y más fiable en la actualidad. Además es el que viene con el entorno de desarrollo NetBeans.

Otras posibles opciones

No conozco mucho el mundo de los servidores web con soporte para JSP, aunque hay algunos comerciales.

Adaptación a otras tecnologías

En principio, no debería haber dificultades en cambiar a otro servidor web, dado que la aplicación no utiliza ninguna función especial de Tomcat.

5.6 Sistema operativo: Debian GNU / Linux

¿Por qué GNU / Linux?

Porque es el sistema operativo que utilizo en casa, y por tanto el más fácil para hacer el trabajo. Además, es un SO reconocido y fiable, y sobre todo es software

libre.

Otras posibles opciones

Cualquier otra distribución de GNU / Linux, o Windows.

Adaptación a otras tecnologías

Dado que todo el desarrollo utiliza Java, PostgreSQL y Tomcat, se podría adaptar perfectamente porque hay entornos de ejecución Java para múltiples plataformas, así como versiones de PostgreSQL y Tomcat.

5.7 Entorno de desarrollo: Sun NetBeans 4

¿Por qué NetBeans?

Porque es un entorno gratuito multiplataforma, fácil de instalar y utilizar, que ayuda mucho en el desarrollo.

Otras posibles opciones

Principalmente Eclipse o Borland JBuilder. Alternativamente, se podría haber trabajado con cualquier editor de textos.

Adaptación a otras tecnologías

Sin dificultades.

5.8 Herramienta CASE: Poseidon UML CE

¿Por qué Poseidon?

Porque es la herramienta CASE de distribución libre que mejor conozco. De hecho, utilicé la versión Community Edition porque es la única gratuita.

Otras posibles opciones

ArgoUML... No conozco otras gratuitas o libres.

Adaptación a otras tecnologías

No creo que los diagramas realizados en Poseidon sirvan en otras herramientas CASE.

6 Aspectos no implementados

En términos generales, la implementación obedece principalmente a la demostración de la factibilidad del sistema. Dado el límite de tiempo y el objetivo primario del TFC, se ha puesto el acento en implementar los aspectos más esenciales, dejando para el final los aspectos secundarios. No obstante, en el diseño se ha intentado tener en cuenta también a éstos.

A continuación detallo algunos aspectos que no han sido implementados por falta de tiempo, o bien mejoras que se podrían introducir en la implementación actual.

Validación de los datos introducidos al crear un recurso.

Al crear o modificar un recurso y sus prioridades, el sistema debería poder validar los datos introducidos, sobre todo verificar que el tipo de dato sea correcto (por ejemplo, si se espera un valor numérico decimal, validarlo). Esto es especialmente importante en el caso de las interfaces web, porque los formularios web no permiten definir un tipo de datos. Debería ser parte de la interfaz de usuario, bien implementándolo con JavaScript que se ejecuta en el lado del cliente, o bien creando un scriplet JSP o una clase Java que lo hiciera y devolviera el resultado; personalmente, descartaría el JavaScript para evitar incompatibilidades entre los navegadores y lo ejecutaría en el lado del servidor, con un scriplet JSP.

Áreas

Sólo se ha definido un área y no se ha implementado la administración de éstas. Las áreas permitirían agrupar temáticamente a los recursos, facilitando su selección por parte de los consumidores. En el caso de existir pocos recursos, el área no parece un factor importante, pero si hubiera decenas de recursos, sí sería bueno poder agruparlos por áreas temáticas.

Completado de los datos personales de un consumidor.

Dado que este aspecto es secundario para el objetivo del TFC, he preferido priorizar la dedicación de tiempo a otros aspectos, y por eso algunos datos personales del consumidor (su residencia, por ejemplo) son ignorados.

Cambio de contraseña.

No se ha implementado la posibilidad del cambio de contraseña, tanto de los usuarios/consumidores como del administrador. En cualquier sistema con gestión de usuarios, esto debería hacerse.

Gestión de plantillas de matching.

Debería ser una opción más dentro de la pantalla de administración. Actualmente el sistema permite definir las restricciones del matching en el momento de calcularlo.

Esta posibilidad no se implementó por falta de tiempo, y porque se la considera una opción interesante pero no prioritaria ni imprescindible.

Restricciones de puntuación mínima para un recurso.

La idea era que se pudiera definir, para cada recurso, una puntuación mínima que debe tener un consumidor para poder ser asignado. El objetivo era evitar que se pueda asignar un consumidor a un recurso con una puntuación muy baja, sólo por el hecho de que no hay otro mejor. Esto no ha sido implementado.

Restricciones de mínimo y máximo de personas apuntadas.

En la implementación actual, se toma el número de consumidores que debe haber apuntados a un recurso, aunque se podría perfeccionar indicando un mínimo y un máximo; cuando estos fueran iguales, se actuaría igual que en la implementación actual. Por otro lado, no se ha llegado a tiempo para incluir la gestión de estos valores en la interfaz de usuario, y por tanto se utilizan los valores por defecto almacenados en la base de datos.

Cálculo del matching.

El sistema implementado lo que hace es recoger los datos almacenados de consumidores y recursos, y crear un texto con formato Lindo para que éste programa calcule la función maximizadora. Lo ideal hubiera sido utilizar la API de Lindo (o un algoritmo equivalente) para calcular el matching y almacenar el resultado.

Gestión de usuarios administradores.

El sistema implementa una gestión básica de usuarios consumidores, pero no así de administradores. Se ha definido un único usuario administrador, cuya contraseña está almacenada en la base de datos y sí podría ser modificada, pero no así el nombre de usuario, ni se permite la creación de otros usuarios administradores. En realidad, es un esquema similar al existente en Unix con el usuario *root*.

Borrar a los consumidores después de un matching.

La aplicación finaliza con la asignación de los consumidores a los recursos, creando un fichero en formato LINDO para ser calculado. Faltaría un sistema que permita eliminar a los consumidores ya apuntados, aunque desactivando los recursos, o bien borrándolos, se consigue un efecto parecido.

Almacenamiento de los datos de acceso para la base de datos

Para el acceso a la base de datos se ha creado un usuario, y se ha creado una clase que conecta directamente con la base de datos. Dentro de esta clase (compilada) están guardados los datos de la base de datos (driver, nombre de la base de datos) así como los datos del usuario (nombre y contraseña).

Lo ideal hubiera sido almacenar estos datos en un archivo externo que fuera leído por la aplicación en el momento de ejecutarse, para facilitar el mantenimiento. El problema es que este archivo debería quedar fuera de la estructura de la web, para que no fuera accesible desde afuera, y además la contraseña debería guardarse encriptada.

Dada mi poca experiencia con JSP, no he tenido tiempo para implementar esto. En el caso de haber elegido PHP como lenguaje y motor de scripts, sí lo hubiera hecho, porque tengo amplia experiencia con esta tecnología y sé cómo guardar y acceder a archivos que estén fuera del árbol accesible desde la web.

Mejor aprovechamiento del servidor de base de datos

El PostgreSQL ofrece ciertas funcionalidades comunes a otros servidores comerciales (Oracle, Informix, DB2, SQL Server, etc.) que no han sido utilizadas, pero que podría mejorar el funcionamiento de la aplicación: procedimientos almacenados y disparadores. Algunas de las cosas implementadas en el código de la aplicación podrían serlo directamente en el servidor de base de datos.

Optimización y perfeccionamiento del código

Con un poco más de tiempo, se podría haber optimizado el rendimiento de algunas funciones del código para evitar tantos accesos a la base de datos, optimizado el consumo de recursos de memoria, e incluso perfeccionado en algunos casos, buscando mejores algoritmos.

Documentación del código

La documentación del código está incompleta. Los algoritmos en sí están bastante bien documentados, pero la documentación de atributos y métodos es irregular, faltando en algunos casos. Esto se refleja en la documentación de las clases generada por *javadoc*.

7 Pruebas realizadas

Durante la fase de desarrollo se han ido haciendo pruebas constantemente, verificando el correcto funcionamiento de los distintos componentes. Esto permitió detectar errores y también ciertas carencias en el diseño de las clases o de la interfaz, todo lo cual fue corregido.

Para la prueba final y definitiva, creé 4 recursos y 5 consumidores.

Los recursos son:

1) Programador Java (peso 1.4)

Propiedades (peso total: 3.4)

- Conocimiento teórico de Java (excluyente, peso 1.2)
  - Alto (valor 1)
  - Medio (valor 0.6)
  - Bajo (valor 0.3)
  - Ninguno (valor 0)
- Experiencia en desarrollo de clases Java (peso 1.2)
  - Más de 2 años (valor 1)
  - Menos de 2 años (valor 0.5)
  - Ninguna (valor 0)
- Otros conocimientos (multi-lista, peso 1)
  - JSP (valor 0.6)
  - HTML (valor 0.4)

2) Traductor (peso 1.0)

Propiedades (peso total: 3.8)

- Conocimiento de inglés (excluyente, peso 1.0)
  - Alto (valor 1)
  - Medio (valor 0.6)
  - Bajo (valor 0.3)
  - Ninguno (valor 0)
- Dominio del castellano (excluyente, peso 1.0)
  - Alto (valor 1)
  - Medio (valor 0.6)
  - Bajo (valor 0.3)
  - Ninguno (valor 0)
- Conocimiento de términos informáticos (peso 1.0)

    - Alto (valor 1)
    - Medio (valor 0.6)
    - Bajo (valor 0.3)
    - Ninguno (valor 0)
- Otros idiomas (multi-lista, peso 0.8)
    - Catalán (valor 0.6)
    - Francés (valor 0.2)
    - Alemán (valor 0.2)

3) Diseñador páginas web (peso 1.0)

Propiedades (peso total: 3.4)

- Conocimiento de HTML (excluyente, peso 1.2)
    - Alto (valor 1)
    - Medio (valor 0.6)
    - Bajo (valor 0.3)
    - Ninguno (valor 0)
- Conocimiento de hojas de estilo (peso 1.2)
    - Alto (valor 1)
    - Medio (valor 0.6)
    - Bajo (valor 0.3)
    - Ninguno (valor 0)
- Entornos de desarrollo conocidos (multi-lista, peso 1)
    - Macromedia Dreamweaver (valor 0.3)
    - Mozilla Composer / Nvu (valor 0.2)
    - Microsoft FrontPage (valor 0.1)
    - Desarrollo con un editor de textos (valor 0.3)
    - Otros (valor 0.1)

4) Administrador de GNU / Linux (peso 1.2)

Propiedades (peso total: 4.2)

- Experiencia en GNU / Linux (excluyente, peso 1.2)
    - Más de 2 años (valor 1)
    - Menos de 2 años (valor 0.5)
    - Ninguna (valor 0)
- Conocimiento de redes (peso 1.0)
    - Alto (valor 1)
    - Medio (valor 0.6)
    - Bajo (valor 0.3)

    - Ninguno (valor 0)
- ¿Te gusta programar scripts de shell? (peso 1.0)
    - Mucho (valor 1)
    - Poco (valor 0.3)
    - No entiendo la pregunta (valor 0)
- Distribuciones de GNU / Linux con las que has trabajado bastante (multi-lista, peso 1)
    - Debian / Ubuntu (valor 0.3)
    - RedHat (valor 0.2)
    - Mandriva (valor 0.2)
    - Suse (valor 0.2)
    - Otras (valor 0.1)

La siguiente tabla resume en qué recursos se han inscrito los usuarios, y los elementos elegidos por cada uno. En la fila del título del recurso, se indica la puntuación obtenida para dicho recurso (la puntuación máxima es el peso del recurso).

	sasha	usuario1	usuario2	usuario3	usuario4
1) Programador Java (peso 1.4)	1,4		1,4	0,7	1,15
Conocimiento teórico de Java (excluyente, peso 1.2)					
Alto (valor 1)	X		X		X
Medio (valor 0.6)				X	
Bajo (valor 0.3)					
Ninguno (valor 0)					
Experiencia en desarrollo de clases Java (peso 1.2)					
Más de 2 años (valor 1)	X		X		
Menos de 2 años (valor 0.5)				X	X
Ninguna (valor 0)					
Otros conocimientos (multi-lista, peso 1)					
JSP (valor 0.6)	X		X		X
HTML (valor 0.4)	X		X	X	X
2) Traductor (peso 1.0)	0,91	1			1
Conocimiento de inglés (excluyente, peso 1.0)					
Alto (valor 1)	X	X			X
Medio (valor 0.6)					
Bajo (valor 0.3)					
Ninguno (valor 0)					
Dominio del castellano (excluyente, peso 1.0)					

	sasha	usuario1	usuario2	usuario3	usuario4
Alto (valor 1)	X	X			X
Medio (valor 0.6)					
Bajo (valor 0.3)					
Ninguno (valor 0)					
Conocimiento de términos informáticos (peso 1.0)					
Alto (valor 1)	X	X			X
Medio (valor 0.6)					
Bajo (valor 0.3)					
Ninguno (valor 0)					
Otros idiomas (multi-lista, peso 0.8)					
Catalán (valor 0.6)	X	X			X
Francés (valor 0.2)		X			X
Alemán (valor 0.2)		X			X
3) Diseñador páginas web (peso 1.0)	0,8	0,45		0,91	
Conocimiento de HTML (excluyente, peso 1.2)					
Alto (valor 1)	X			X	
Medio (valor 0.6)		X			
Bajo (valor 0.3)					
Ninguno (valor 0)					
Conocimiento de hojas de estilo (peso 1.2)					
Alto (valor 1)				X	
Medio (valor 0.6)	X	X			
Bajo (valor 0.3)					
Ninguno (valor 0)					
Entornos de desarrollo conocidos (multi-lista, peso 1)					
Macromedia Dreamweaver (valor 0.3)	X			X	
Mozilla Composer / Nvu (valor 0.2)	X				
Microsoft FrontPage (valor 0.1)		X			
Desarrollo con un editor de textos (valor 0.3)	X			X	
Otros (valor 0.1)				X	
4) Administrador de GNU / Linux (peso 1.2)	0,97		0,91	0,65	
Experiencia en GNU / Linux (excluyente, peso 1.2)					
Más de 2 años (valor 1)	X		X	X	
Menos de 2 años (valor 0.5)					
Ninguna (valor 0)					
Conocimiento de redes (peso 1.0)					
Alto (valor 1)			X		
Medio (valor 0.6)	X			X	

	sasha	usuario1	usuario2	usuario3	usuario4
Bajo (valor 0.3)					
Ninguno (valor 0)					
¿Te gusta programar scripts de shell? (peso 1.0)					
Mucho (valor 1)	X				
Poco (valor 0.3)			X	X	
No entiendo la pregunta (valor 0)					
Distribuciones de GNU / Linux (multi-lista, peso 1)					
Debian / Ubuntu (valor 0.3)	X		X		
RedHat (valor 0.2)	X				
Mandriva (valor 0.2)			X	X	
Suse (valor 0.2)			X		
Otras (valor 0.1)	X				

El resultado generado por la aplicación fue este:

```
!Asignando consumidores a 4 recursos.
!Consumidor: sasha - Alias: C1
!Consumidor: usuario1 - Alias: C2
!Consumidor: usuario2 - Alias: C3
!Consumidor: usuario3 - Alias: C4
!Consumidor: usuario4 - Alias: C5

MAX
        1.40 R4_C1 + 1.40 R4_C3 + 0.71 R4_C4 + 1.15 R4_C5 +
        0.92 R5_C1 + 1.00 R5_C2 + 1.00 R5_C5 +
        0.80 R6_C1 + 0.45 R6_C2 + 0.91 R6_C4 +
        0.97 R7_C1 + 0.91 R7_C3 + 0.66 R7_C4
SUBJECT TO
        R4)     R4_C1 + R4_C3 + R4_C4 + R4_C5 <= 1
        R5)     R5_C1 + R5_C2 + R5_C5 <= 1
        R6)     R6_C1 + R6_C2 + R6_C4 <= 1
        R7)     R7_C1 + R7_C3 + R7_C4 <= 1
        usuario4)     R4_C5 + R5_C5 <= 1
        usuario3)     R4_C4 + R6_C4 + R7_C4 <= 1
        usuario2)     R4_C3 + R7_C3 <= 1
        usuario1)     R5_C2 + R6_C2 <= 1
        sasha)     R4_C1 + R5_C1 + R6_C1 + R7_C1 <= 1
END
```

Y el resultado obtenido, después de procesar este texto en LINDO, fue el siguiente:

```
 LP OPTIMUM FOUND AT STEP     12

        OBJECTIVE FUNCTION VALUE

        1)      4.280000
```

```
  VARIABLE        VALUE          REDUCED COST
     R4_C1         0.000000          0.060000
     R4_C3         1.000000          0.000000
     R4_C4         0.000000          0.860000
     R4_C5         0.000000          0.000000
     R5_C1         0.000000          0.390000
     R5_C2         0.000000          0.000000
     R5_C5         1.000000          0.000000
     R6_C1         0.000000          0.000000
     R6_C2         0.000000          0.040000
     R6_C4         1.000000          0.000000
     R7_C1         1.000000          0.000000
     R7_C3         0.000000          0.000000
     R7_C4         0.000000          0.420000


       ROW   SLACK OR SURPLUS     DUAL PRICES
        R4)         0.000000          1.150000
        R5)         0.000000          1.000000
        R6)         0.000000          0.490000
        R7)         0.000000          0.660000
 USUARIO4)          0.000000          0.000000
 USUARIO3)          0.000000          0.420000
 USUARIO2)          0.000000          0.250000
 USUARIO1)          1.000000          0.000000
    SASHA)          0.000000          0.310000

 NO. ITERATIONS=      12
```

Como se puede observar, LINDO asignó el puesto de Programador Java (R4) a “usuario2”, el puesto de Traductor (R5) a “usuario4”, el puesto de Diseñador web (R6) a “usuario3” y el puesto de Administrador de GNU / Linux a “sasha”.

8 Conclusiones

El objetivo inicial era poder diseñar un sistema que fuera capaz de gestionar una serie de recursos, por una parte; que ofreciera a los usuarios la posibilidad de apuntarse a dichos recursos, por otra; y que con estos dos requisitos fuera capaz de construir una salida que permitiera la asignación de los consumidores a los recursos, maximizando la puntuación global.

Este objetivo ha sido conseguido, y además ha sido posible implementarlo, aunque sólo sea en una parte mínima indispensable.

Además, el sistema ha sido diseñado y construido de forma que sea independiente de la plataforma, y con el único requisito por parte de los usuarios (sean administradores o consumidores) de disponer de un navegador web para su utilización.

La aplicación puede ser ampliada con nuevas funcionalidades, y mejorada en su implementación, pero creo que partiendo de la base actual ello es factible.

No obstante, la sensación que me queda es que me hubiera gustado disponer de más tiempo para evaluar determinadas tecnologías antes de decidirme por alguna de ellas, y por otro lado, para poder implementar algunas de las funcionalidades de que carece actualmente, así como de mejorar la documentación.

Qué cambiaría si tuviera que volver a hacerlo.

Todo proyecto implica un aprendizaje, y más en este caso, que he trabajado con tecnologías que prácticamente desconocía. Debe ser por esto que se suele decir que, cuando se termina la primera versión de una aplicación, al desarrollador le gustaría volver a escribirla desde cero. En este caso, no volvería a comenzar desde el inicio, pero sí haría algunos cambios de enfoque.

Intentaría almacenar la información en archivos XML, que permitirían replicar mejor la estructura de la información (recursos que pertenecen a un área, que se componen a su vez de propiedades, y éstas de elementos). La estructura del archivo de recursos podría ser así:

```
<area 1>
     <recurso 1>
          <propiedad 1>
               <elemento 1></elemento 1>
               <elemento n></elemento n>
          </propiedad 1>
          <propiedad n>
     </recurso 1>
     <recurso n>
```

```
</area 1>
<area n>
...
```

Como ya comenté más arriba, descarté XML por el temor a no poder acabar el proyecto a tiempo, dado que mi experiencia con la gestión de archivos XML ha sido negativa, y dado que no conozco qué posibilidades aporta Java en este sentido. Debería haber dedicado buena parte del tiempo inicial del proyecto en investigar este campo, y el tiempo de implementación se hubiera reducido drásticamente.

En cuanto a la lógica de la aplicación, en lugar de utilizar scriplets no visuales, utilizaría JavaBeans. La decisión de utilizar scriplets vino dada por mi inexperiencia en el trabajo con aplicaciones Java para web, y una vez avanzado el proyecto, era difícil cambiar. De todos modos, a efectos de rendimiento el resultado es el mismo, la diferencia estaría en que utilizando JavaBeans el diseño sería más pulcro y fácil de entender.

9 Glosario de la aplicación

Recurso

Oferta que se pone a disposición de los usuarios del sistema, para que éstos puedan apuntarse y tener la posibilidad de ser seleccionados. Un recurso puede ser un puesto de trabajo, un determinado TFC, un puesto de voluntario en una ONG, incluso una persona que busca pareja!

Un recurso pertenece a un área, y está compuesto por atributos propios (descripción, peso, etc.) y propiedades.

Propiedad

Componente de un recurso. Contiene atributos propios (descripción, peso, etc.) y elementos. Puede ser una propiedad excluyente, y puede permitir la selección de uno o más elementos, según su definición.

Elemento de propiedad

Son las diferentes opciones que ofrece una propiedad. Cada elemento tiene una descripción y un valor, y cuando es elegido por un consumidor, este valor se suma al resto de elementos para conformar la puntuación de dicho consumidor para un recurso.

Consumidor

Es el usuario del sistema, que cuando se apunta a uno o más de los recursos ofrecidos, se transforma en un consumidor de dicho recurso, elegible para el momento de la selección final (matching).

Atributo

Relación que se establece cuando un consumidor elige un elemento de una propiedad de un recurso. Por tanto, un atributo se compone de un consumidor, un recurso, una propiedad de ese recurso y un elemento de esa propiedad. Por ejemplo, se puede decir que el consumidor “usuario1” contiene el atributo “Alto” de la propiedad “Conocimiento de Java” del recurso “Programador de Java”.

Peso

Importancia relativa de un recurso o bien de una propiedad dentro de un recurso. Se utiliza para multiplicar la puntuación parcial o total. Cuando el peso es superior a 1, la puntuación aumenta; cuando el peso es inferior a 1, la puntuación disminuye; si el peso es igual a 1, no tiene incidencia.

Puntuación

Cuando un consumidor elige un elemento de una propiedad, suma a su puntuación el valor de dicho elemento. La suma de todos los elementos de una propiedad se multiplica por el peso de esa propiedad, y la suma de todos los

puntos de las propiedades, de divide por el total de puntos de las propiedades (para obtener un porcentaje) y luego se multiplica por el peso del recurso, para obtener la puntuación final, que es la que se va a utilizar al asignar consumidores a recursos (ejecutar el matching).

Match

Relación que se establece entre un recurso, un consumidor y su puntuación.

Matching

Asignación de los consumidores apuntados a los recursos. Esta asignación persigue maximizar la puntuación global del sistema.

Criterio

Definición de una plantilla de matching, conteniendo básicamente el número de consumidores que se puede apuntar a cada recurso (por defecto es 1). Quizás un término más afortunado hubiera sido *Restricción*.

10 Bibliografía

Material de *Enginyeria de programari I* (UOC).

Programació en Java (UOC).

Internet:

- Documentación de Java Server Pages:
  http://java.sun.com/products/jsp/docs.html
- Documentación de Java 1.4.2:
  http://java.sun.com/docs/
- PostgreSQL 7.4: http://www.postgresql.org/docs/7.4/interactive/
- Apache Tomcat 4: http://jakarta.apache.org/tomcat/tomcat-4.1-doc/
- Algunos tutoriales de JSP:
  http://www.etse.urv.es/EngInf/assig/sob/
  http://geneura.ugr.es/~jmerelo/JSP/
  http://www.jsptut.com/