UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA
EN INFORMÁTICA

PROYECTO FIN DE CARRERA

GCAD: Sistema Cliente/Servidor para la Gestión de Decisiones en Desarrollo Distribuido de Software.

Juan Andrada Romero

Febrero, 2012

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

DEPTO. DE TECNOLOGÍAS,
Y SISTEMAS DE INFORMACIÓN

PROYECTO FIN DE CARRERA

GCAD: Sistema Cliente/Servidor para la Gestión de Decisiones en Desarrollo Distribuido de Software.

Autor: Juan Andrada Romero

Director: Aurora Vizcaíno Barceló

Febrero, 2012

GCAD: Sistema Cliente/Servidor para la Gestión de Decisiones en Desarrollo Distribuido de Software.

TRIBUNAL:

Presidente: _____

Vocal 1: _____

Vocal 2: _____

Secretario: _____

FECHA DE DEFENSA: _____

CALIFICACIÓN: _____

PRESIDENTE	VOCAL 1	VOCAL 2	SECRETARIO
Fdo.:	Fdo.:	Fdo.:	Fdo.:

Resumen

Abstract

... english version for the abstract ...

Agradecimientos

Escribir agredecimientos

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Índice de algoritmos

Capítulo 1

Resultados

En este capítulo se expondrán los resultados obtenidos al aplicar la metodología descrita en el capítulo **??** para el desarrollo del presente PFC.

Se muestra todo el proceso de desarrollo real que se ha seguido hasta la obtención del sistema, desde su fase de inicio hasta su finalización, mostrando su evolución de manera incremental y la evolución de requisitos funcionales que ha sufrido.

Siguiendo la metodología propuesta por el Proceso Unificado de Desarrollo, el ciclo de vida de desarrollo del sistema se ha dividido en las cuatro fases que el PUD propone: Inicio, Elaboración, Construcción y Transición. Cada una de esas fases, a su vez, se ha dividido en una o más iteraciones, donde intervienen las disciplinas de Captura de Requisitos, Análisis, Diseño, Implementación y Pruebas, en mayor o menor medida, según la fase e iteración donde nos encontremos. De este modo, el sistema va incrementando su funcionalidad al terminar cada una de estas iteraciones.

Para evitar que este capítulo sea demasiado extenso, no se profundiza del mismo modo en todas las fases e iteraciones, ni se han incluido todos los diagramas que se han realizado, especialmente en las iteraciones de análisis y diseño, debido a la gran cantidad de diagramas que se obtuvieron. Por otro lado, se han incluido algunos fragmentos de código fuente que tienen especial relevancia e interés en el sistema.

1.1 Fase de Inicio

En esta fase, como se muestra en la Figura 1.1, interviene principalmente la disciplina de Captura de Requisitos. Es por ello que que esta fase es ligeramente diferente al resto de iteraciones, ya que se centra en: la captura e identificación de requisitos, el estudio de viabilidad del sistema y en la creación del plan de iteraciones, por lo que esta fase no se ha dividido en iteraciones.

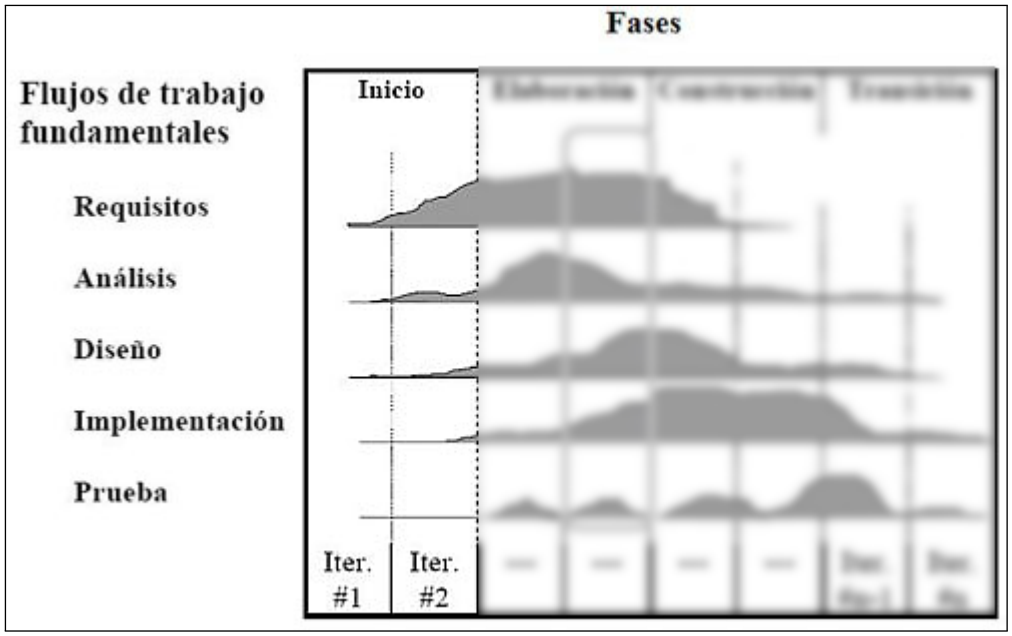

Figura 1.1: Fase de inicio en el PUD

En los siguientes apartados se exponen las tareas realizadas en esta fase.

1.1.1 Captura e identificación de requisitos

Para la identificación de requisitos del sistema, se estudió la literatura existente relacionada con el tema y se fueron extrayendo requisitos funcionales que pudieran resolver o mitigar los desafíos encontrados en la literatura, desarrollados en el capítulo **??**.

Por otra parte, para que el sistema obtenido como resultado del desarrollo de este proyecto pudiera tener una aplicación real, era necesario conocer el funcionamiento de las empresas que siguen el paradigma de GSD y qué información es relevante para los miembros de los equipos de desarrollo. La información que interesaba conocer era cómo se distribuyen los proyectos,

los tipos de roles que participan en el desarrollo, experiencia de los desarrolladores, cómo se distribuyen los equipos en los países, cómo se comunican e intercambian información, etc.

Para poder conocer esta información acerca del dominio de aplicación del sistema, se recurrió a un jefe de proyecto de INDRA Software Labs S.L. Además, también se ha utilizado la propia experiencia del autor y desarrollador del presente PFC en una empresa que cuenta con centros de desarrollo distribuidos en diferentes países.

Una vez estudiada la literatura, conocidos los problemas del GSD y con conocimiento del dominio de la aplicación del sistema, se pasó a elaborar una serie de requisitos funcionales que el sistema debe cumplir para alcanzar los objetivos propuestos en el Capítulo **??**.

1.1.1.1 Requisitos funcionales

A continuación, se enumera el conjunto de requisitos funcionales que el sistema debe cumplir.

- Se debe permitir que diferentes miembros de los equipos de desarrollo de una compañía puedan acceder al sistema, mediante un nombre de usuario y contraseña, proporcionados por la empresa.

- Como en los equipos de desarrollo existen diferentes roles entre sus miembros, el sistema debe adaptarse a las acciones que cada miembro del equipo (en adelante, usuario) puede realizar, según su rol.

- Los usuarios pueden participar activamente en las decisiones que se van tomando durante el ciclo de vida de los proyectos en los que participan, de tal modo que la herramienta debe permitir gestionar estas decisiones. Así, se podrán crear, modificar y eliminar dichas decisiones, agrupadas en tres categorías: *Temas*, *Propuestas* y *Respuestas*.

- La herramienta debe mostrar, de una manera gráfica e intuitiva, el conjunto de decisiones que han sido tomadas en un proyecto software. Además, se deberá proporcionar diferente información acerca de esas decisiones: su autor, centro de desarrollo y compañía donde pertenece, detalles de esa decisión, etc.

- La herramienta debe proveer mecanismos de notificación síncrona y asíncrona que faciliten la comunicación de las últimas modificaciones:

  - Se debe crear un sistema de notificaciones o alertas, informando al usuario cuando se producen cambios sobre las decisiones de los proyectos donde trabaja.
  - Se deben mostrar cambios en tiempo real, es decir, si un empleado se encuentra visualizando el conocimiento en la aplicación y, en ese momento, otro empleado de otra localización diferente realiza un cambio sobre el mismo conocimiento, éste debe ser notificado al primer empleado en tiempo real, actualizando su vista.

- Se deben poder dar de alta nuevos proyectos en el sistema, así como modificar los existentes, permitiendo seleccionar los usuarios que en dichos proyectos participarán.

- Los miembros del equipo de desarrollo normalmente trabajan en varios proyectos, por lo que el sistema deberá permitir escoger el proyecto en el cuál se van a gestionar las decisiones.

- Cuando se presenta un nuevo proyecto, se debe poder aconsejar decisiones tomadas en proyectos ya terminados, que tuviesen características similares.

- La herramienta debe poder configurarse y mostrarse en diferentes idiomas. Es decir, debe soportar la *internacionalización*.

- La herramienta debe generar *logs* con las acciones que se van realizando en el sistema.

- Se debe poder exportar todas las decisiones realizadas en un proyecto a un fichero XML, así como toda la información asociada a dichas decisiones.

Una vez identificada esta lista de requisitos a las que el sistema debe dar soporte, éstos se agruparon en diferentes grupos funcionales del sistema, como se muestra en la tabla 1.2.

Junto a los requisitos capturados del sistema, se identificaron los roles de usuario mostrados en la Tabla 1.3, así como los requisitos, o acciones, que cada uno de estos roles podían ejecutar. En la Tabla 1.1 se muestran las acciones que cada usuario puede realizar en el sistema.

1.1.1.2 Requisitos no funcionales

A continuación, se enumera el conjunto de requisitos no funcionales del sistema, dependientes del entorno tecnológico de la aplicación.

Requisito	Empleado	Jefe Proyecto
F1.1	✓	✓
F1.2	✓	✓
F2.1	✓ (excepto *Temas*)	✓
F2.2	✓ (excepto *Temas*)	✓
F2.3	✓ (excepto *Temas*)	✓
F3.1	✓	✓
F3.2	✓	✓
F4.1	✓	✓
F4.2	✓	✓
F4.3	✓	✓
F5.1		✓
F5.2		✓
F5.3	✓	✓
F5.4	✓	✓
F5.5		✓
F6.1	✓	✓
F7.1		✓

Tabla 1.1: Acciones que un usuario puede realizar en el sistema - v1.0

- El sistema debe estar basado en tecnología Java.

- El sistema debe ser distribuido, para facilitar su uso entre los diferentes miembros de los equipos de desarrollo que se encuentran geográficamente deslocalizados. Por tanto, es un sistema cliente-servidor, formado por dos subsistemas.

- Debe existir una base de datos en el servidor que permita almacenar todas las decisiones tomadas en los proyectos, la información de los proyectos, la información de usuarios, etc.

Grupo Funcional	Descomposición de requisitos	Id. Requisito
F1: Acceso Sistema	Login (acceso)	**F1.1**
	Logout (desconexión)	**F1.2**
F2: Gestión Decisiones	Crear decisión	**F2.1**
	Modificar decisión	**F2.2**
	Eliminar decisión	**F2.3**
F3: Visualización información	Visualizar decisiones	**F3.1**
	Visualizar datos empresa	**F3.2**
F4: Gestión Notificaciones	Consultar notificaciones	**F4.1**
	Modificar notificación	**F4.2**
	Eliminar notificación	**F4.3**
F5: Gestión Proyectos	Dar de alta un proyecto	**F5.1**
	Modificar datos proyecto	**F5.2**
	Consultar usuarios	**F5.3**
	Seleccionar proyecto activo	**F5.4**
	Aconsejar decisiones de proyectos	**F5.5**
F6: Gestión Idiomas	Cambiar idioma	**F6.1**
F7: Exportar información	Exportar información	**F7.1**

Tabla 1.2: Grupos funcionales del sistema - v1.0

Rol	Descripción
Empleado	Es un usuario del sistema que representa a un miembro de un equipo de desarrollo de una compañía.
Jefe de Proyecto	Es un usuario del sistema que representa a un jefe de proyecto de un equipo de desarrollo de una compañía.

Tabla 1.3: Roles identificados en el sistema

1.1.2 Modelo de casos de uso

Una vez identificados los requisitos funcionales del sistema, éstos se modelan en un diagrama de casos de uso. Al estar en la fase de inicio, centrada en la captura de requisitos y en una visión de alto nivel de la funcionalidad del sistema, dicho diagrama de casos de uso se modela con un alto nivel de abstracción, solamente mostrando los diferentes grupos funcionales del sistema que se detallaron en la Tabla 1.2. En la siguiente fase, se modelarán los diagramas de casos de uso con un nivel mayor de detalle, mostrando los diferentes casos de uso que modelan los requisitos de cada uno de esos grupos funcionales.

Cabe destacar que el sistema está compuesto de dos subsistemas, cliente y servidor, como se ha comentado en los requisitos no funcionales (ver apartado 1.1.1.2), por lo que, aunque los requisitos funcionales se refieren al sistema global, se han modelado casos de uso para ambos subsistemas.

Los actores que aparecen en los diagramas de casos de uso del cliente se refieren a las personas físicas que pueden utilizar el sistema (empleado y jefe de proyecto) y al subsistema servidor. Por otro lado, en el servidor aparece como actor el subsistema cliente y el actor que representa la base de datos.

1.1.2.1 Modelo de casos de uso para el subsistema cliente

En la Figura 1.2 se muestra el diagrama de casos de uso de alto nivel para el subsistema cliente. Todos los casos de uso en este subsistema tienen un mecanismo común de funcionamiento: se solicitan al usuario los datos que correspondan a la funcionalidad de ese caso de uso (datos del proyecto, datos de una decisión, etc.) y se envían dichos datos al servidor, esperando su respuesta y mostrando los resultados. Por esta razón, los actores que aparecen en el diagrama

de casos de uso de este subsistema son los diferentes roles del sistema, y el servidor.

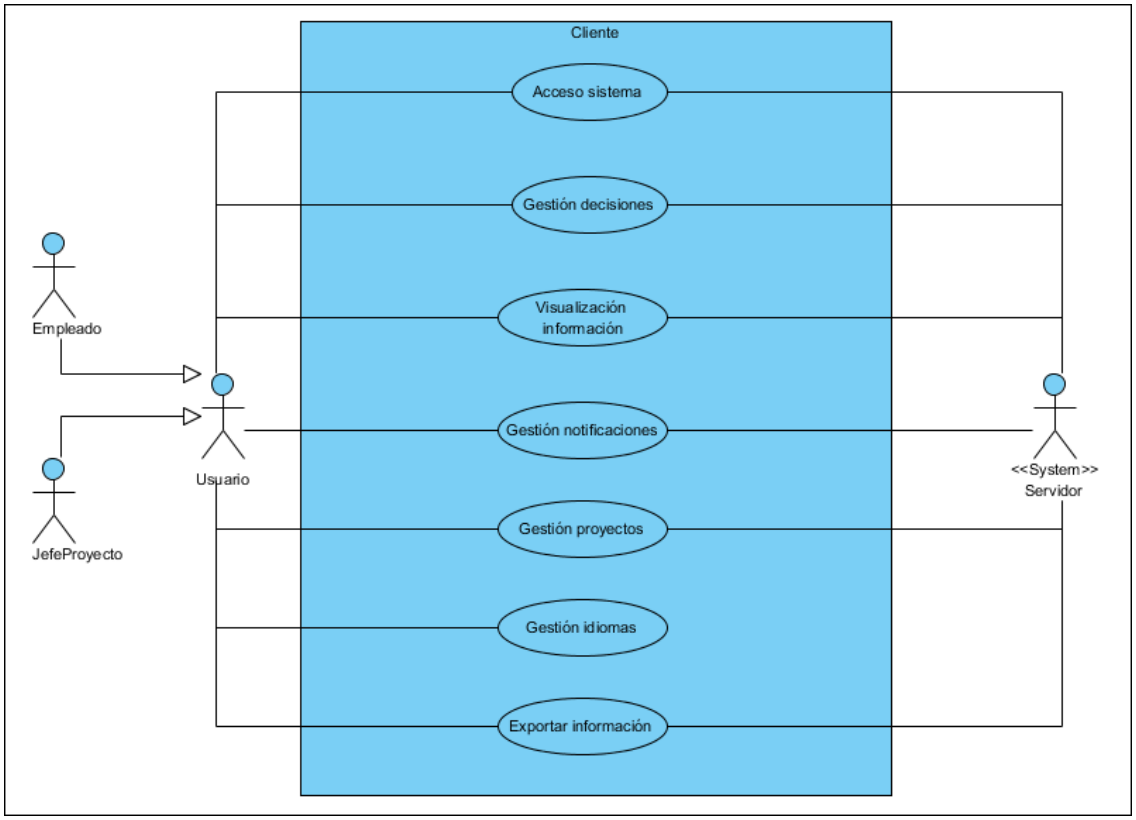

Figura 1.2: Diagrama de casos de uso - Cliente - v1.0

1.1.2.2 Modelo de casos de uso para el subsistema servidor

En la Figura 1.3 se puede observar el diagrama de casos de uso de alto nivel para el servidor. En este subsistema, el modo de proceder es el siguiente: se reciben las peticiones del subsistema cliente, se opera con la lógica de control y de dominio, accediendo a la base de datos y se envía la respuesta al cliente, notificando los posibles cambios. Por esta razón, los actores que aparecen en el diagrama de casos de uso de este subsistema son el subsistema cliente y el SGDB.

En cuanto a los casos de uso, aparecen los mismos que en el subsistema anterior, salvo dos casos de uso adicionales, descritos a continuación. Todos los casos de uso del servidor incluyen la funcionalidad de estos casos de uso, pero no se ha representado en el diagrama para facilitar la legibilidad.

- **Actualizar ventanas**: este caso de uso se utiliza para que el servidor pueda notificar

cambios al cliente, producidos por otros clientes.

- **Actualizar log**: este caso de uso se utiliza para almacenar y reflejar en el servidor todas las acciones realizadas por los clientes, creando *logs*.

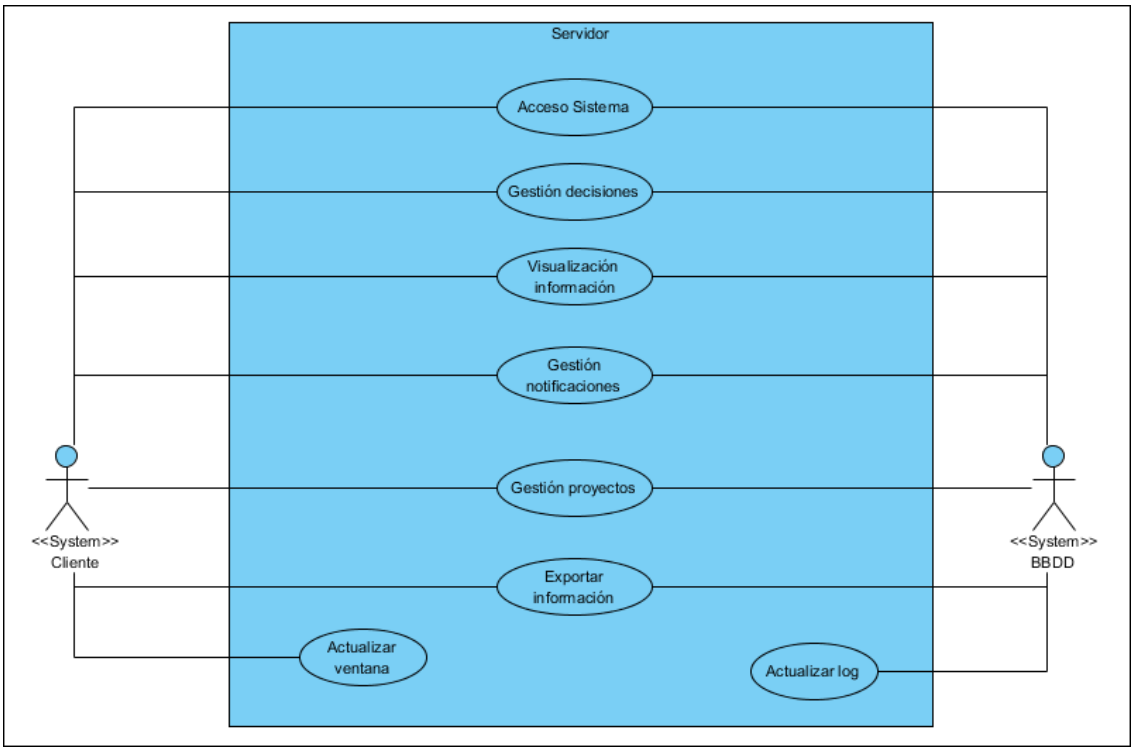

Figura 1.3: Diagrama de casos de uso - Servidor - v1.0

1.1.3 Glosario de términos

El glosario de términos está compuesto por los términos que fueron definidos en la sección **??**.

1.1.4 Gestión del riesgo

Para realizar un estudio de viabilidad del proyecto, se analizaron los aspectos técnicos y las tecnologías disponibles para la realización de dicho proyecto, así como un estudio del tiempo y duración del proceso.

Uno de los puntos más importantes detectados en este análisis previo fue la necesidad de crear un sistema cliente-servidor, donde el cliente fuera totalmente independiente de la

implementación del servidor. Además, otro aspecto destacado es que la interfaz gráfica del cliente debe ser flexible y adaptable a las operaciones que los diferentes usuarios puedan realizar en el sistema.

Analizando las tecnologías existentes, se comprobó que era factible realizar estos objetivos, gracias al uso de **RMI** y del API de reflexión de Java. Además, con el resto de tecnologías utilizadas, se podía cubrir el resto de objetivos del sistema.

Por otra parte, se realizó una estimación previa de la duración del proyecto y se comprobó que podía terminarse en el tiempo estimado, teniendo en cuenta posibles retrasos debidos a imprevistos y vacaciones.

1.1.5 Plan de iteraciones

Dado que el PUD es centrado y dirigido por casos de uso, se utiliza el diagrama de casos que representa los grupos funcionales de alto nivel para poder crear una primera versión del plan de proyecto (o plan de iteraciones) y llevar a cabo el desarrollo de dichos grupos funcionales en las iteraciones de las fases posteriores. De este modo, se obtienen las iteraciones mostradas en la Tabla 1.7.

1.2 Fase de Elaboración

En esta fase intervienen la Captura de Requisitos, Análisis, Diseño, Implementación y Pruebas, con mayor o menor influencia, según la iteración donde nos encontremos (ver Figura 1.4).

Esta fase se ha divido en dos iteraciones: la primera de ellas se centra de nuevo en la captura de requisitos, en el análisis y en algunas tareas de diseño, como la definición de la arquitectura del sistema. La segunda iteración se centra en tareas de diseño y de implementación, principalmente de la comunicación de los subsistemas en una arquitectura cliente-servidor.

1.2.1 Iteración 1

En esta iteración se abordan las siguientes tareas:

Fase	Iteración	Objetivos
Inicio	Preliminar	Realizar un estudio del sistema a desarrollar, capturando e identificando sus requisitos funcionales.
Elaboración	1	Validar el estudio del sistema e identificar posibles nuevos requisitos, modelando los requisitos del sistema con mayor detalle y realizando la definición de su arquitectura.
	2	Analizar e identificar los objetos de dominio que forman parte del sistema, a partir de los casos de uso detallados, y diseñar e implementar la comunicación entre sistemas en la arquitectura cliente-servidor definida.
Construcción	3	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F1**.
	4	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F3**.
	5	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F2** y **F4**
	6	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F5**.
	7	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F6** y **F7**.
Transición	8	Obtener la versión entregable del sistema, así como su documentación y manuales de usuario.

Tabla 1.4: Primera versión del plan de iteraciones

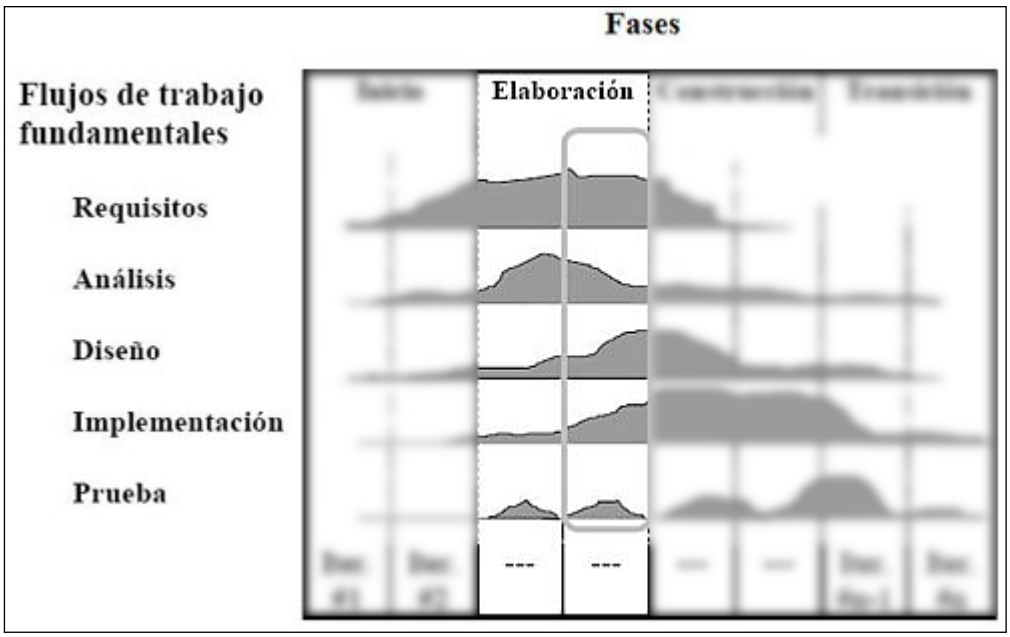

Figura 1.4: Fase de elaboración en el PUD

- Identificación de nuevos requisitos funcionales.
- Elaboración del modelo de casos de uso definitivo.
- Elaboración del plan de proyecto definitivo.
- Definición de la arquitectura del sistema.

1.2.1.1 Identificación de requisitos

Al terminar la iteración de la fase anterior, obteniendo un primer modelo de casos de uso con los requisitos identificados, se llevó a cabo una reunión de seguimiento, para revisar la planificación realizada en el plan de proyecto y poder identificar nuevos requisitos funcionales, si los hubiera.

Como resultado de esta reunión, se identificaron nuevos requisitos para añadir al sistema, y que servirán para mitigar o resolver más problemas que aparecen al aplicar el paradigma de GSD. Estos nuevos requisitos fueron:

- Además de poder mostrar las decisiones tomadas en un proyecto, se debe poder cambiar el estado de éstas, aceptando o rechazando dichas decisiones, para posteriormente,

cuando el proyecto llegue a su fin, poder conocer qué notificaciones fueron de ayuda y cuáles no.

- Para aumentar la información que se añade al crear nuevas decisiones de un proyecto, se debe poder adjuntar ficheros a dichas decisiones. Del mismo modo, se deben poder descargar esos archivos adjuntos, si los hubiera.
- Se deben poder generar de manera automática estadísticas que ayuden a llevar un control sobre los proyectos que se están desarrollando por los diferentes centros de desarrollo software de una compañía. También, se podrán generar estadísticas acerca de los miembros de los equipos.
- Se deben poder generar informes de las decisiones tomadas en un proyecto, en forma de tabla y de manera automática. Dicho informe debe generarse como un documento PDF.

Con estos nuevos requisitos, se actualizan los grupos funcionales creados en la fase de inicio, las acciones que pueden realizar los diferentes roles de usuarios y el modelo de casos de uso de alto nivel, explicado en el siguiente apartado.

En la Tabla 1.5 se muestran los nuevos grupos funcionales actualizados, incluyendo los nuevos requisitos que se han identificado en esta fase.

Por otra parte, en la Tabla 1.6 se muestran las acciones que se pueden realizar en el sistema según el rol del usuario.

1.2.1.2 Modelo de casos de uso

En la Figura 1.5 se muestra el diagrama de casos de uso que representa los grupos funcionales actualizados en el subsistema cliente. Del mismo modo, la Figura 1.6 refleja el diagrama de casos de uso de alto nivel actualizado del subsistema servidor.

Una vez identificados y modelados los nuevos requisitos, obteniendo los diagramas de casos de uso de alto nivel actualizados, se profundiza y aumenta el detalle en los diagramas de casos de uso, representando, para cada grupo funcional del sistema, el conjunto de casos de uso que modelan los requisitos que se engloban en dicho grupo, como se ha detallado en la Tabla 1.5. Por tanto, en en las secciones posteriores, se muestran estos diagramas de casos de uso detallados para cada grupo funcional, formando diferentes vistas del sistema global.

Funcionalidad	Descomposición de requisitos	Id. Requisito
F1: Acceso Sistema	Login (acceso)	**F1.1**
	Logout (desconexión)	**F1.2**
F2: Gestión Decisiones	Crear decisión	**F2.1**
	Modificar decisión	**F2.2**
	Eliminar decisión	**F2.3**
	Adjuntar ficheros	**F2.4**
	Aceptar o rechazar decisiones	**F2.5**
F3: Visualización datos empresa	Visualizar decisiones	**F3.1**
	Visualizar compañía	**F3.2**
	Descargar ficheros adjuntos	**F3.3**
F4: Gestión Notificaciones	Consultar notificaciones	**F4.1**
	Modificar notificación	**F4.2**
	Eliminar notificación	**F4.3**
F5: Gestión Proyectos	Dar de alta un proyecto	**F5.1**
	Modificar datos proyecto	**F5.2**
	Consultar usuarios	**F5.3**
	Seleccionar proyecto activo	**F5.4**
	Aconsejar decisiones de proyectos	**F5.5**
F6: Generación informes	Generación informes PDF	**F6.1**
F7: Generación estadísticas	Generación gráficos estadísticos	**F7.1**
F8: Gestión Idiomas	Consultar idiomas	**F8.1**
F9: Exportar información	Exportar información	**F9.1**

Tabla 1.5: Grupos funcionales del sistema - v2.0

Requisito	Empleado	Jefe Proyecto
F1.1	✓	✓
F1.2	✓	✓
F2.1	✓ (excepto *Temas*)	✓
F2.2	✓ (excepto *Temas*)	✓
F2.3	✓ (excepto *Temas*)	✓
F2.4	✓	✓
F2.5		✓
F3.1	✓	✓
F3.2	✓	✓
F3.3	✓	✓
F4.1	✓	✓
F4.2	✓	✓
F4.3	✓	✓
F5.1		✓
F5.2		✓
F5.3	✓	✓
F5.4	✓	✓
F5.5		✓
F6.1		✓
F7.1		✓
F8.1	✓	✓
F9.1		✓

Tabla 1.6: Acciones que un usuario puede realizar en el sistema - v2.0

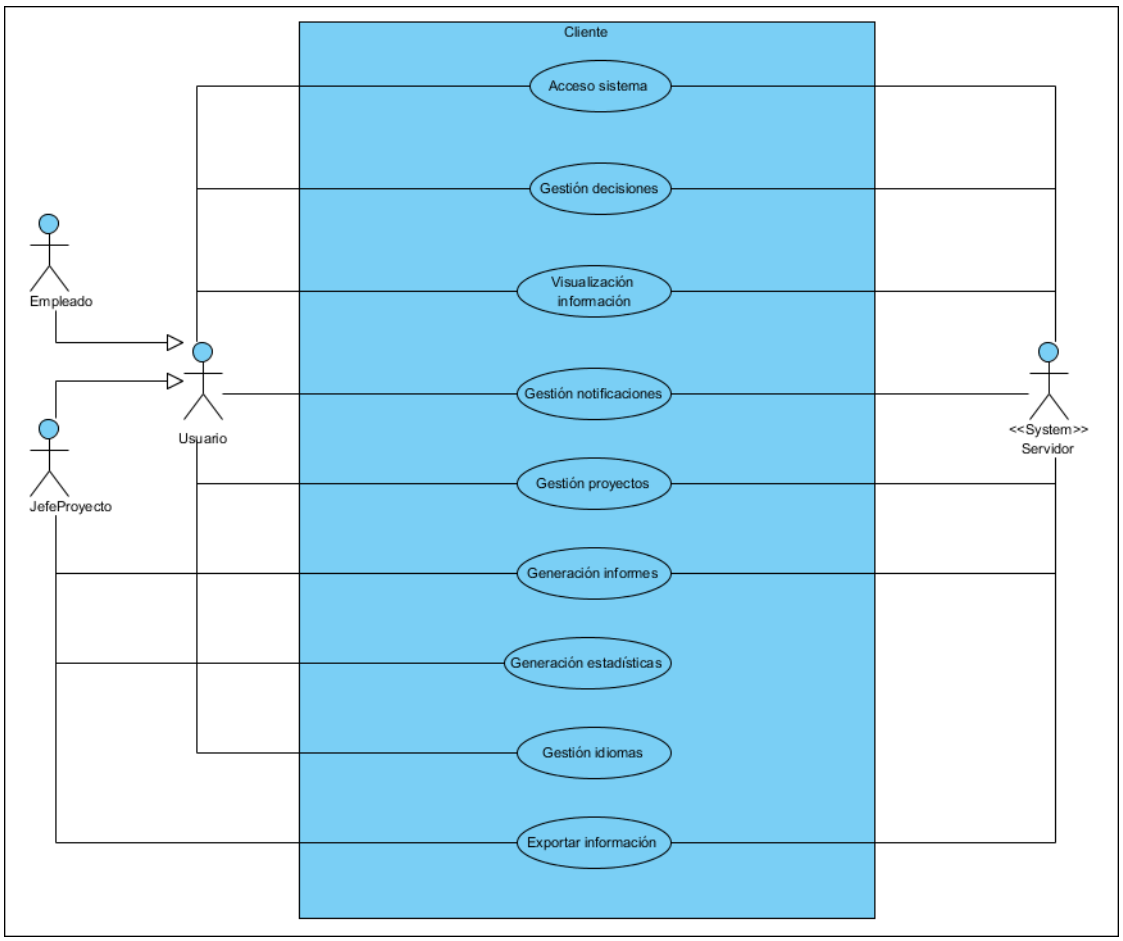

Figura 1.5: Diagrama de casos de uso - Cliente - v2.0

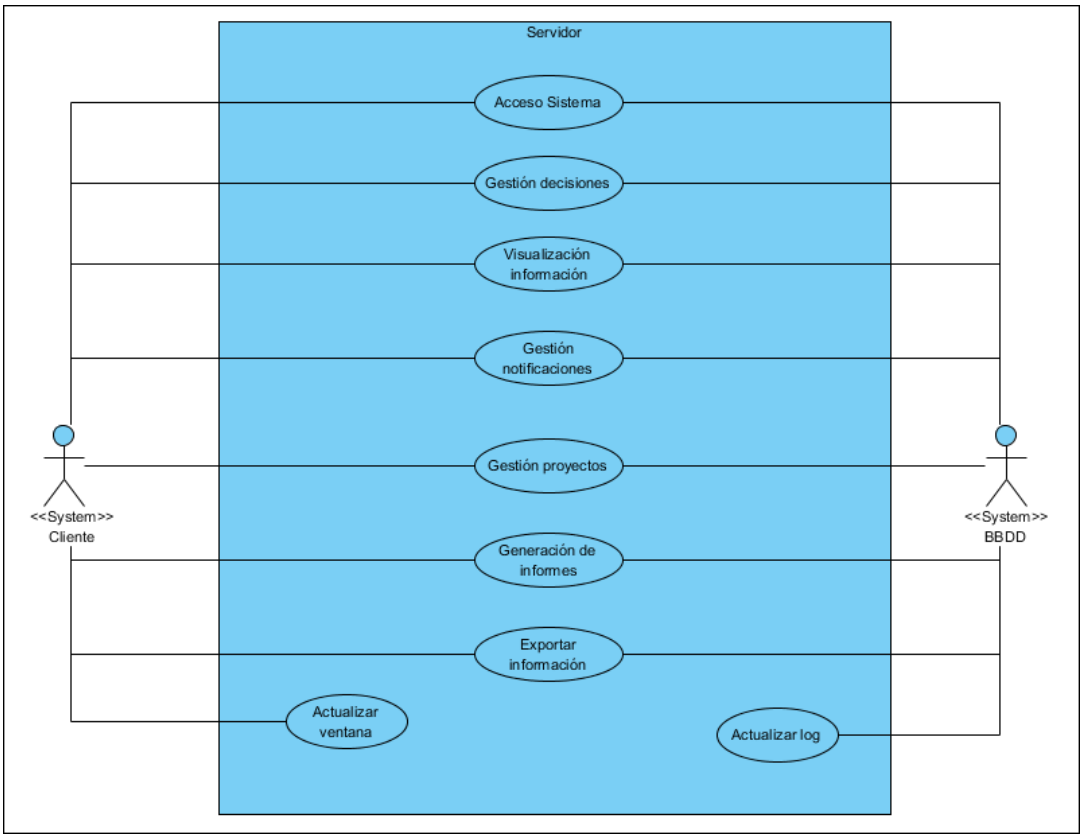

Figura 1.6: Diagrama de casos de uso - Servidor - v2.0

1.2.1.2.1 Acceso al sistema

En la Figura 1.7 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 1.8 se muestra el diagrama de casos de uso para el servidor.

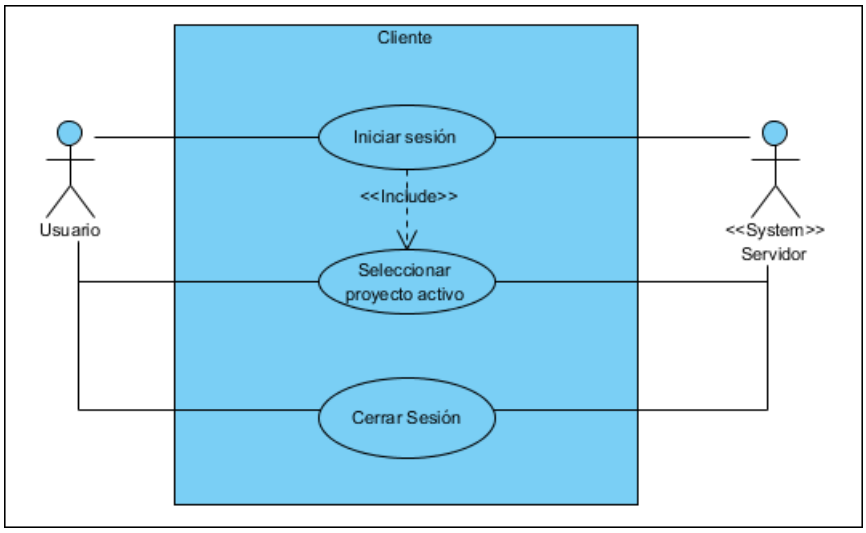

Figura 1.7: Diagrama de casos de uso - Cliente - Acceso al sistema

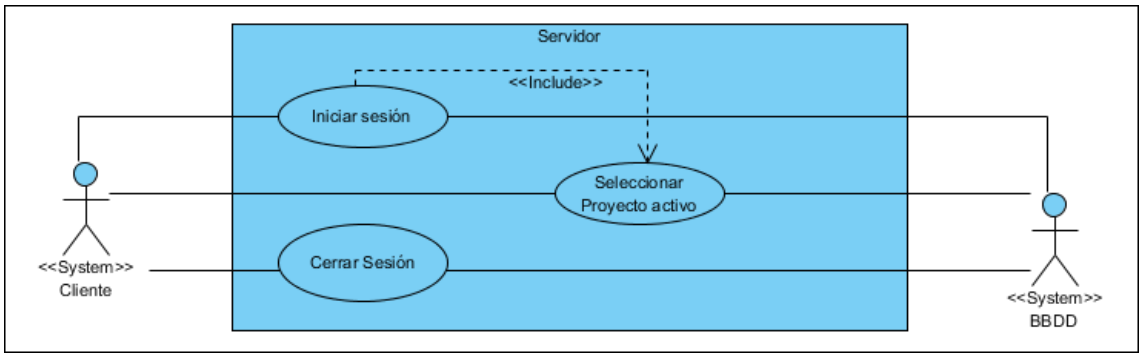

Figura 1.8: Diagrama de casos de uso - Servidor - Acceso al sistema

1.2.1.2.2 Gestión de decisiones

En la Figura 1.9 se muestra el diagrama de casos de uso para el cliente, mientras que en Figura 1.10 refleja el diagrama de casos de uso para el servidor.

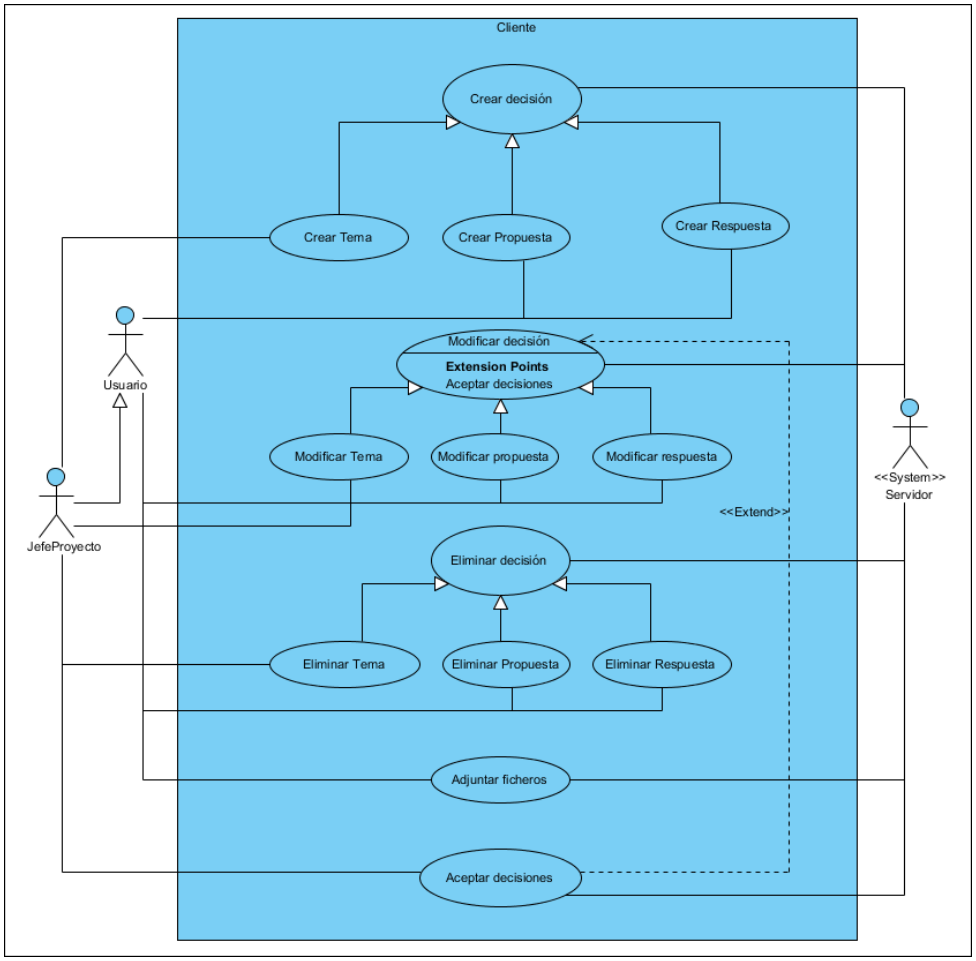

Figura 1.9: Diagrama de casos de uso - Cliente - Gestión decisiones

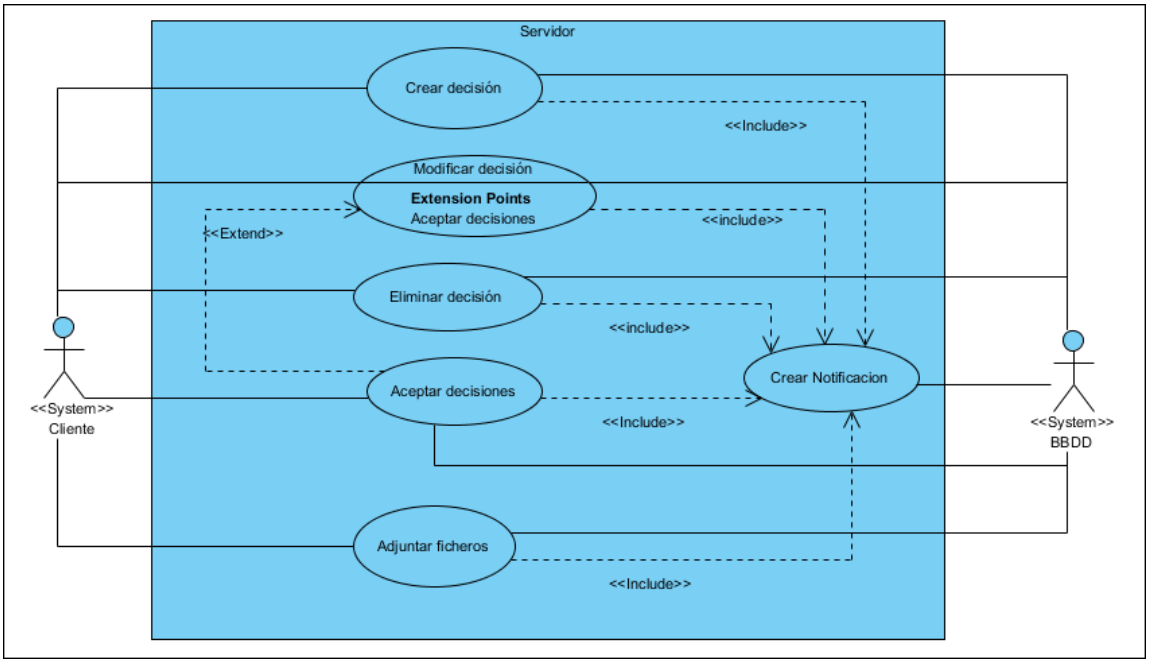

Figura 1.10: Diagrama de casos de uso - Servidor - Gestión decisiones

1.2.1.2.3 Visualización de información

En la Figura 1.11 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 1.12 se observa el diagrama de casos de uso para el servidor.

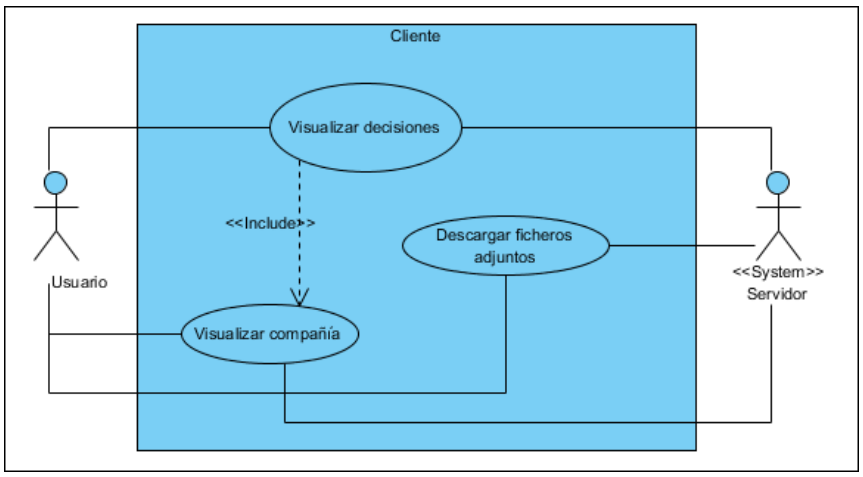

Figura 1.11: Diagrama de casos de uso - Cliente - Visualización información

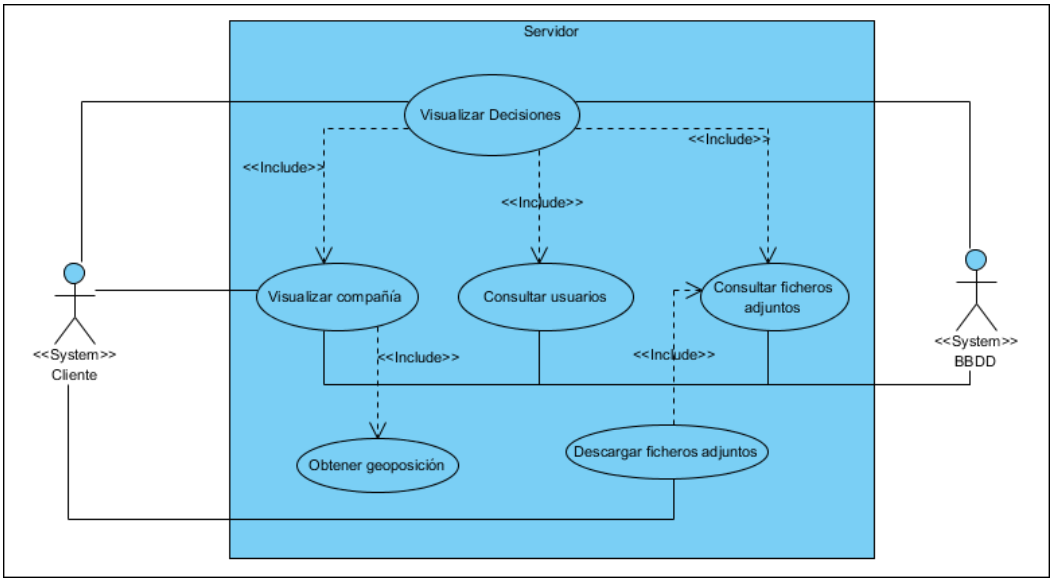

Figura 1.12: Diagrama de casos de uso - Servidor - Visualización información

1.2.1.2.4 Gestión de notificaciones

En la Figura 1.13 se muestra el diagrama de casos de uso para el cliente, mientras que la Figura 1.14 refleja el diagrama de casos de uso para el servidor.

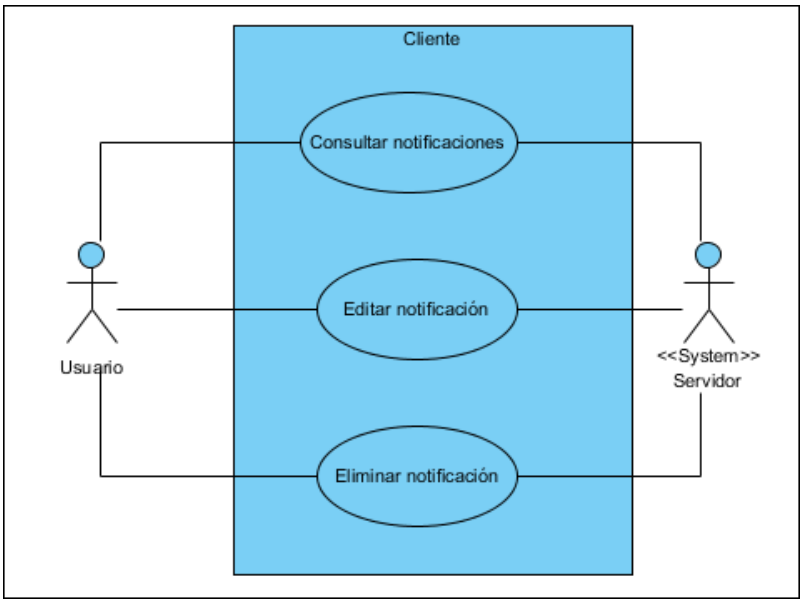

Figura 1.13: Diagrama de casos de uso - Cliente - Gestión notificaciones

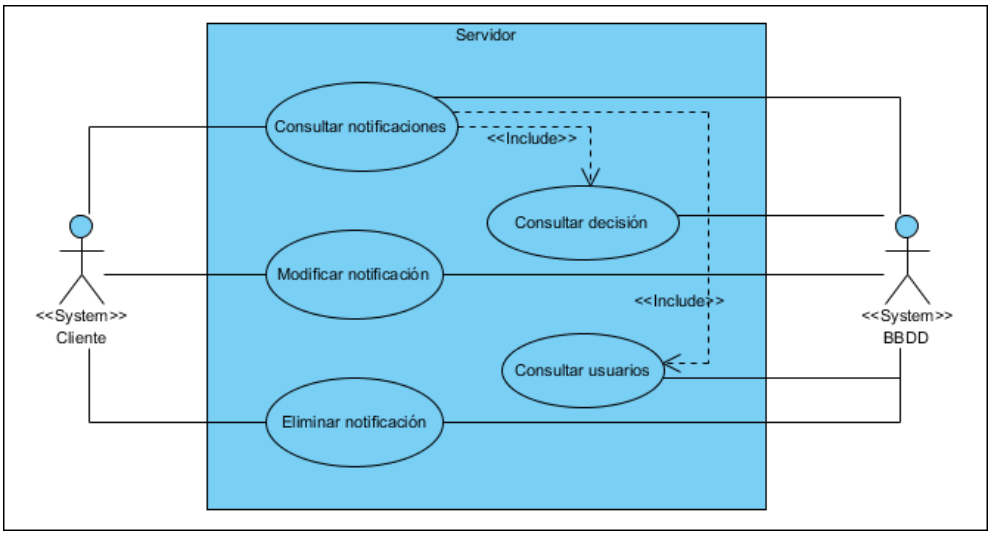

Figura 1.14: Diagrama de casos de uso - Servidor - Gestión notificaciones

1.2.1.2.5 Gestión de proyectos

En la Figura 1.15 se muestra el diagrama de casos de uso para el cliente, mientras que la Figura 1.16 refleja el diagrama de casos de uso para el servidor.

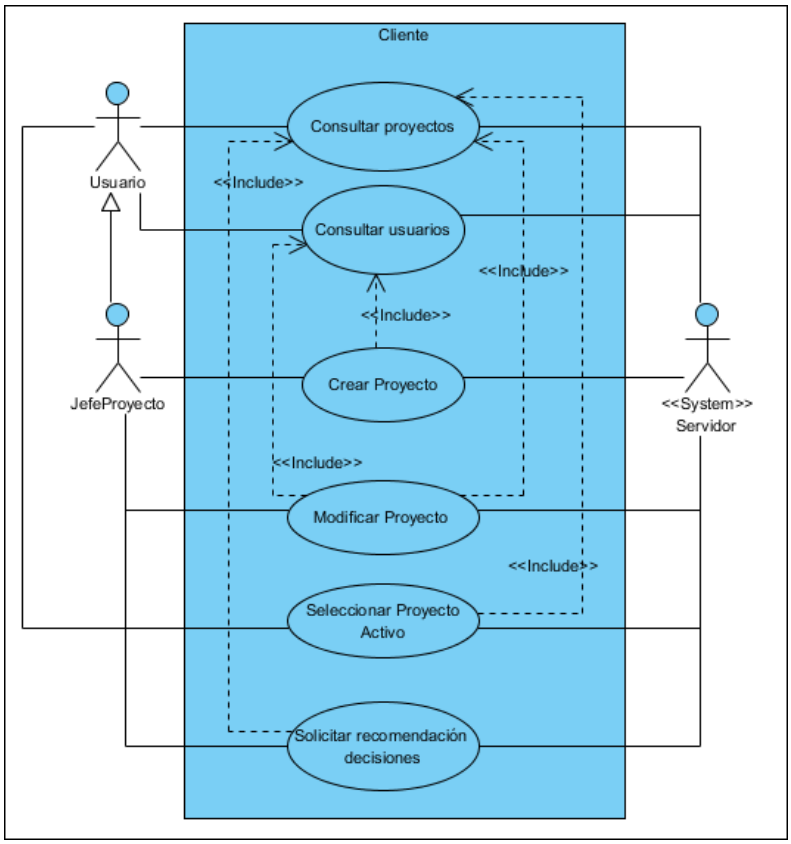

Figura 1.15: Diagrama de casos de uso - Cliente - Gestión Proyectos

1.2.1.2.6 Generación de estadísticas

En la Figura 1.17 se muestra el diagrama de casos de uso para el cliente.

Esta funcionalidad es exclusiva del subsistema cliente, ya que, utilizando los datos que proporciona el servidor, las gráficas se generan y muestran en el cliente, debido a que los tipos de gráficas son independientes del servidor y dependen de la tecnología utilizada para representarlas gráficamente en el cliente.

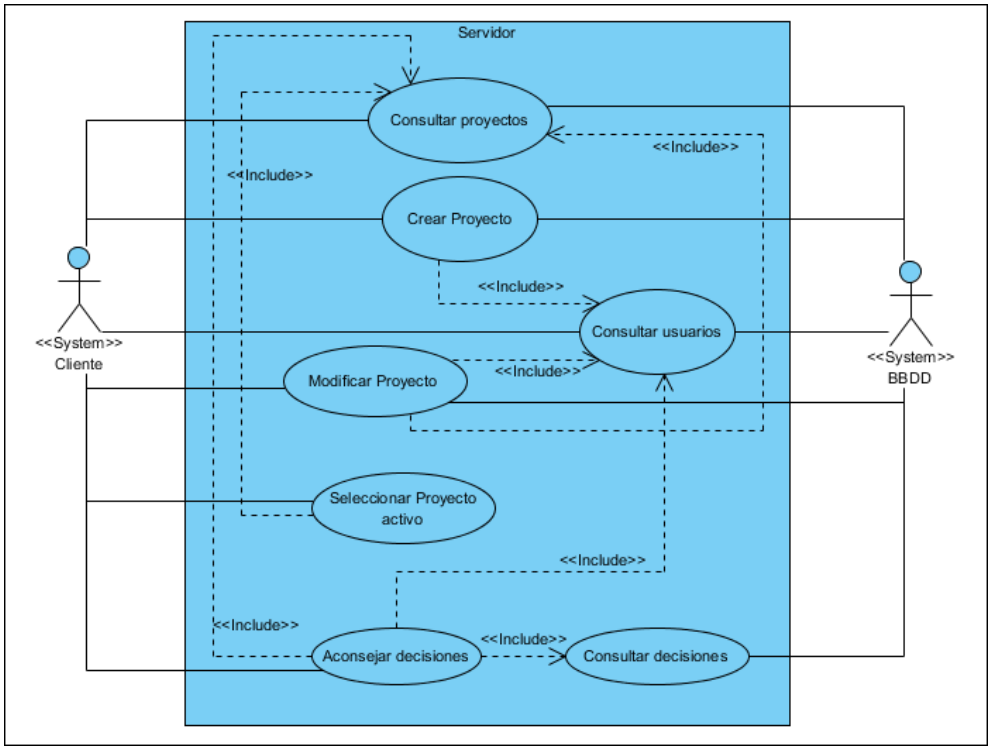

Figura 1.16: Diagrama de casos de uso - Servidor - Gestión Proyectos

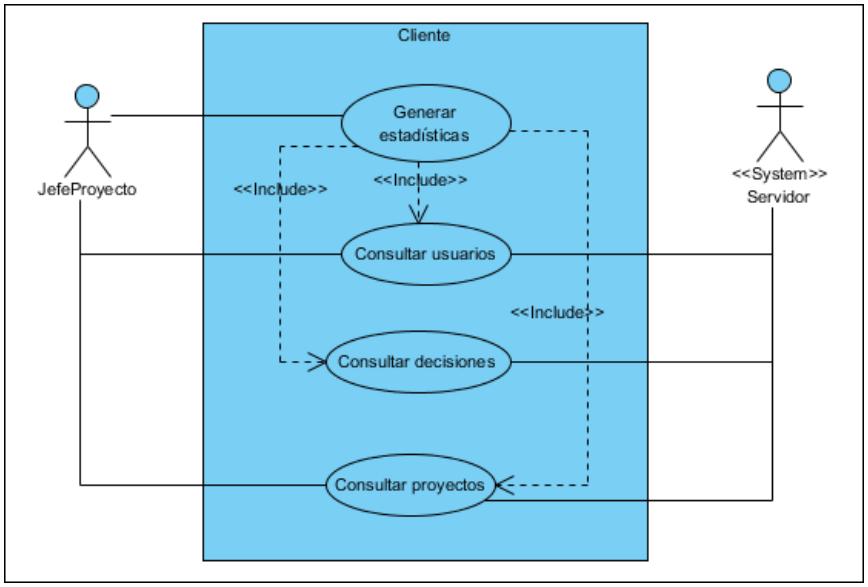

Figura 1.17: Diagrama de casos de uso - Cliente - Generación Estadísticas

1.2.1.2.7 Generación de informes

En la Figura 1.18 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 1.19 se muestra el diagrama de casos de uso para el servidor.

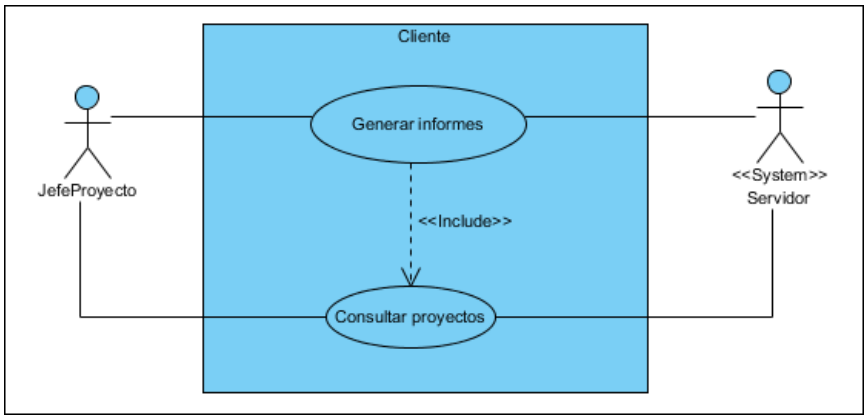

Figura 1.18: Diagrama de casos de uso - Cliente - Generación Informes

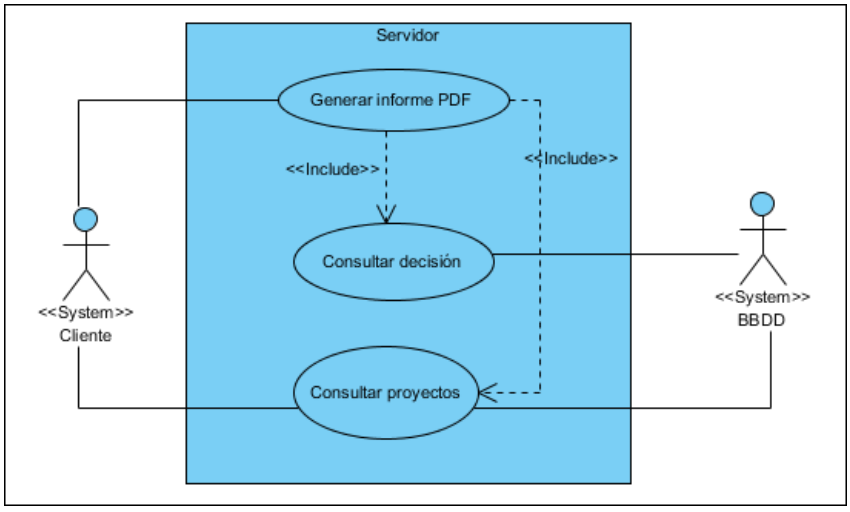

Figura 1.19: Diagrama de casos de uso - Servidor - Generación Informes

1.2.1.2.8 Gestión de idiomas

En la Figura 1.20 se muestra el diagrama de casos de uso para el cliente.

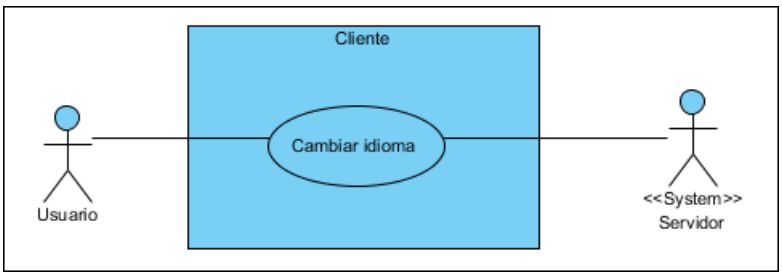

Figura 1.20: Diagrama de casos de uso - Cliente - Gestión Idiomas

1.2.1.2.9 Exportar información

En la Figura 1.21 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 1.22 se muestra el diagrama de casos de uso para el servidor.

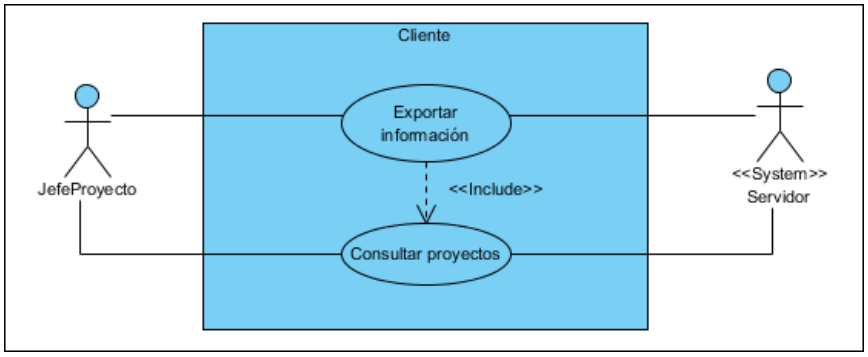

Figura 1.21: Diagrama de casos de uso - Cliente - Exportar información

1.2.1.3 Plan de iteraciones

Una vez identificados nuevos requisitos, se reorganiza el plan de proyecto con nuevas iteraciones, para poder tener en cuenta el desarrollo de los nuevos grupos funcionales identificados. Por tanto, a partir de las iteraciones planificadas en la Tabla 1.7, se obtiene el plan de iteraciones definitivo mostrado en la Tabla **??**.

Los nuevos requisitos identificados en esta iteración sólo afectan, con respecto al plan de proyecto inicial, a las iteraciones de la fase de Construcción. Por tanto, gracias al carácter iterativo del PUD, se han podido identificar e incluir nuevos requisitos al desarrollo del sistema, sin verse afectadas ninguna de las iteraciones anteriores y habiéndose detectado estos requisitos aún en fases tempranas del ciclo de vida del desarrollo, por lo que no se ponía en riesgo la viabilidad del proyecto debido a estos nuevos cambios.

Fase	Iteración	Objetivos
Inicio	Preliminar	Realizar un estudio del sistema a desarrollar, capturando e identificando sus requisitos funcionales.
Elaboración	1	Validar el estudio del sistema e identificar posibles nuevos requisitos, modelando los requisitos del sistema con mayor detalle y realizando la definición de su arquitectura.
	2	Analizar e identificar los objetos de dominio que forman parte del sistema, a partir de los casos de uso detallados, y diseñar e implementar la comunicación entre sistemas en la arquitectura cliente-servidor definida.
Construcción	3	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F1**.
	4	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F3**.
	5	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F2** y **F4**
	6	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F5**.
	7	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F6** y **F7**.
	8	Análisis, diseño, implementación y pruebas de los casos de uso del grupo funcional **F8** y **F9**.
Transición	9	Obtener la versión entregable del sistema, así como su documentación y manuales de usuario.

Tabla 1.7: Primera versión del plan de iteraciones

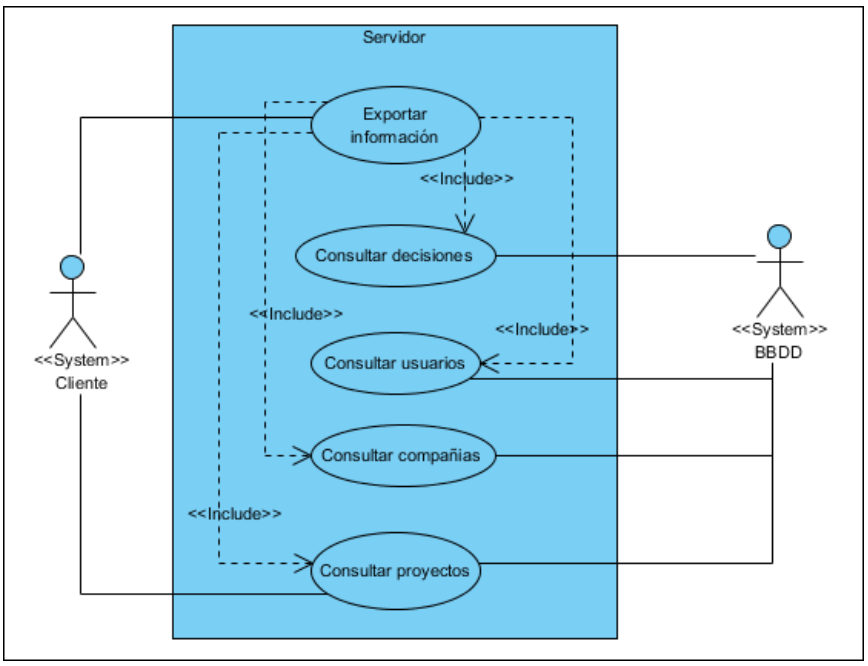

Figura 1.22: Diagrama de casos de uso - Servidor - Exportar información

A partir de este punto, se siguen las iteraciones planificadas en este plan de iteraciones definitivo.

1.2.1.4 Arquitectura del sistema

Al final de esta primera iteración de la fase de elaboración se define la arquitectura del sistema.

1.2.1.4.1 Arquitectura cliente-servidor

Como se comentó en el apartado 1.1.1.2, el sistema a desarrollar debe ser distribuido, para poder ser utilizado desde localizaciones diferentes, por lo que se ha decidido utilizar una arquitectura cliente-servidor. De esta forma, el sistema completo se divide en dos subsistemas:

- **Subsistema Cliente**: este subsistema es el utilizado por los miembros de los equipos de desarrollo deslocalizados. Este subsistema es el que se encarga de mostrar la interfaz gráfica de usuario, recoger las acciones que el usuario desea hacer y enviar dicha acción al servidor, esperando su respuesta para actualizar la interfaz gráfica en consecuencia.
- **Subsistema Servidor**: este subsistema se encarga de recibir las peticiones del subsistema cliente, procesarlas, almacenar y recuperar información de su base de datos y enviar

la respuesta al cliente. También se encarga de que todas las acciones queden registradas.

Así, cada uno de los subsistemas se puede encontrar en máquinas diferentes, pues, siguiendo la arquitectura cliente-servidor, se comunican a través de **RMI** (ver sección **??**). Además, la base de datos utilizada por el servidor también se pueden encontrar en una máquina diferente a la máquina donde se encuentre el servidor.

Por tanto, algunas ventajas de utilizar este enfoque distribuido siguiendo la arquitectura cliente-servidor son:

- **Centralización del control**: toda la lógica de dominio y control está centralizada en el servidor, por lo que el sistema cliente es totalmente independiente de la implementación del servidor. Del mismo modo, el sistema cliente es independiente de como se realiza la gestión del conocimiento en el servidor. Además, los accesos, recursos y la integridad de los datos son controlados por el servidor de forma que un cliente defectuoso o no autorizado no pueda dañar el sistema.
- **Escalabilidad**: se pueden añadir nuevos tipos de sistemas clientes para que se comuniquen con el servidor.
- **Fácil mantenimiento**: al estar los sistemas distribuidos en diferentes máquinas, es posible reemplazar, reparar o actualizar el servidor, mientras que sus clientes no se verán afectados por ese cambio.

En la Figura 1.23 se muestra una vista de los subsistemas que integran el sistema global y como se comunican dichos sistemas a través de interfaces, facilitando la distribución de cada uno de los sistemas en diferentes máquinas.

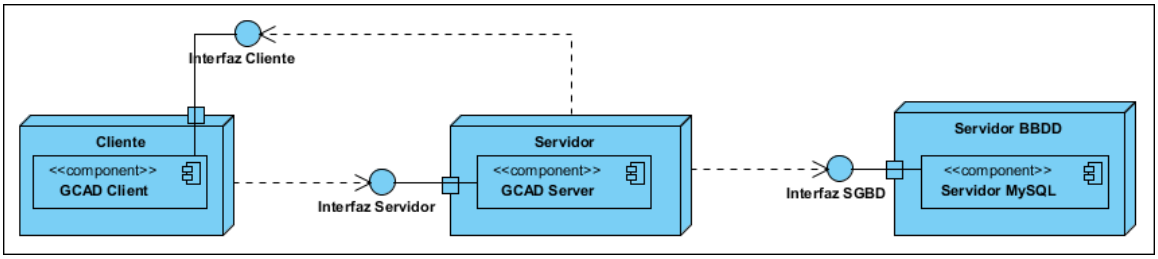

Figura 1.23: Arquitectura cliente-servidor

1.2.1.4.2 Arquitectura multicapa

En cuanto a la arquitectura de implementación, ambos sistemas, cliente y servidor, serán desarrollados siguiendo una arquitectura multicapa, de modo que se pueda aislar la capa de presentación de la de la lógica de dominio y ésta de la capa de persistencia (ver Figura 1.24).

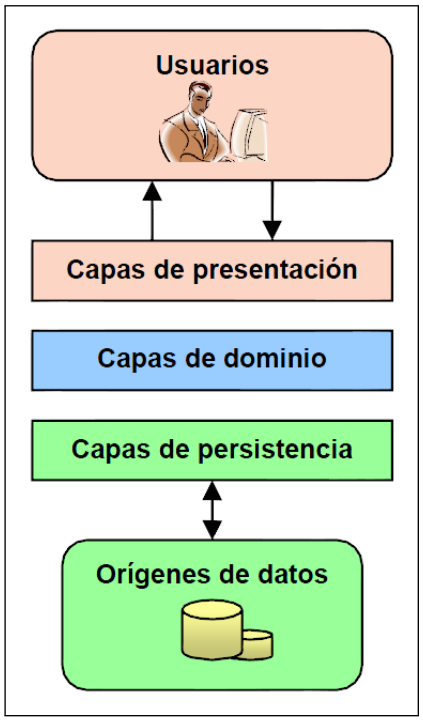

Figura 1.24: Arquitectura multicapa

De este modo, las capas con las que cuenta cada subsistema son las siguientes:

- Cliente: comunicaciones, dominio y presentación.
- Servidor: comunicaciones, dominio, persistencia y presentación.

Más concretamente, estas capas se traducirán a paquetes de implementación, donde cada uno agrupara los siguientes elementos:

- **Comunicaciones**: contiene las clases e interfaces necesarias para la comunicación de los subsistemas a través de RMI.
- **Dominio**: contiene todos los objetos del dominio de la aplicación.

- **Persistencia**: contiene las clases encargadas de gestionar la persistencia de los objetos de dominio.

- **Presentación**: contiene todas las vistas de la interfaz gráfica de usuario, organizadas en subpaquetes.

En la Figura 1.25 puede observarse esta arquitectura multicapa y las relaciones entre dichas capas.

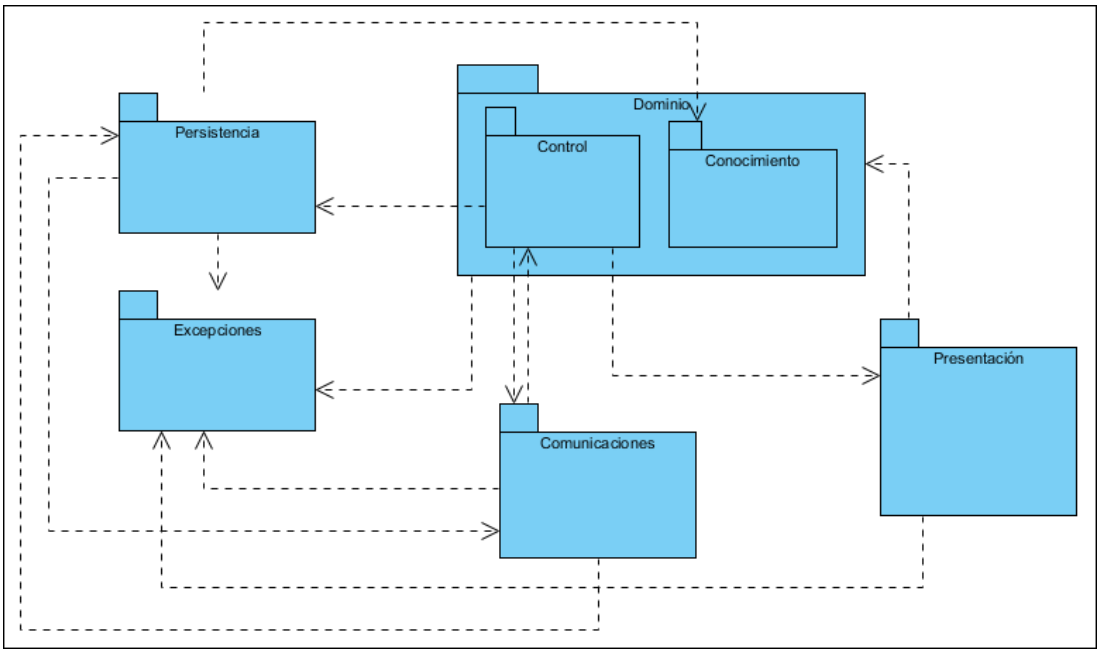

Figura 1.25: Arquitectura multicapa

Utilizando este enfoque multicapa, se sigue el principio de mínimo acoplamiento y máxima cohesión, desacoplando los elementos de una capa de los de otra. De este modo, se facilita el mantenimiento y extensibilidad del sistema, pues cuando se realice el cambio en algún elemento de una capa, el resto de capas no se verán afectadas.

1.2.1.4.3 Patrones utilizados

En cuanto a los patrones de diseño, se han utilizado los siguientes para el desarrollo del sistema:

- **Singleton**: patrón que asegura que para una determinada clase solo exista una única

instancia y provee un punto de acceso a ella. Se utiliza para implementar el patrón Agente y para otras clases que deben ser únicas en el sistema.

- **Agente**: patrón utilizado para gestionar acceso y manipulación a ficheros.
- **DAO (Data Access Object)**: patrón que permite separar los objetos de dominio de su persistencia, haciendo independiente cómo se almacenan. Se ha utilizado este patrón para gestionar la persistencia de los objetos de dominio.
- **Observador**: patrón que permite que los cambios sobre un objeto se notifiquen directamente al conjunto de objetos dependientes (observados). Se utiliza para gestionar varias conexiones de bases de datos y para notificar a los clientes cambios que se produzcan en el sistema.
- **Fachada**: patrón que proporciona una interfaz de alto nivel que delega posteriormente en una o varias clases. Se ha utilizado para implementar la interfaz de operaciones del servidor, que el subsistema cliente puede utilizar.
- **Proxy**: patrón que proporciona el acceso a otro objeto, normalmente remoto. Se ha utilizado para comunicar ambos subsistemas por RMI.
- **Iterator**: patrón que proporciona un mecanismo para acceder y recorrer objetos secuencialmente, sin exponer su representación. Se ha utilizado para recorrer enumeraciones y colecciones.

1.2.2 Iteración 2

Al finalizar la iteración anterior, se realizó otra reunión de seguimiento para revisar y validar todos estos artefactos obtenidos.

En dicha reunión, ya no se identificaron más requisitos y se validó el plan de iteraciones definitivo y el resto de artefactos obtenidos, por lo que en sucesivas iteraciones y fases se comenzó con el desarrollo de los casos de uso identificados, utilizando como entrada el modelo de casos de uso y de análisis obtenido en la iteración anterior.

Cabe destacar que, como se explicó en el apartado 1.2.1.4, al utilizar una arquitectura cliente-servidor, el subsistema del servidor es el encargado de toda la lógica y control de

dominio y persistencia, por lo que es donde más hincapié se hará en los diagramas de diseño, mientras que el cliente se encarga de proporcionar la interfaz gráfica del sistema, validar los datos que introduce el usuario y enviar y recibir las peticiones del servidor.

Señalar que la implementación de ambos subsistemas se irá haciendo en paralelo, es decir, cada caso de uso se desarrollará para el servidor y para el cliente, cerrando así la implementación completa de ese caso de uso.

Las tareas a realizar en esta iteración son:

- Análisis e identificación de los objetos de dominio, creando un modelo de alto nivel.
- Modelado y diseño de la base de datos.
- Diseño e implementación de la arquitectura cliente-servidor, centrándose en la comunicación entre subsistemas.
- Diseño e implementación de pruebas relativas a la comunicación entre ambos subsistemas.

1.2.2.1 Diagrama de clases de dominio

Se realiza un diagrama de clases de diseño de alto nivel, reflejando las clases de los objetos de dominio y sus relaciones. Este diagrama se irá detallando y refinando durante las iteraciones de la fase de construcción.

Así, según la especificación de requisitos de la sección 1.1.1.1, se han modelado las siguientes clases de conocimiento, junto con sus relaciones:

- **User**: una clase abstracta que representa los usuarios que pueden acceder y hacer uso del sistema. De ella heredan dos clases, que son *Employee* y *ChiefProject*, representando los diferentes roles de usuarios que existen en el sistema. Estas clases concretas implementan el método abstracto *getRole()*, devolviendo el rol correspondiente a cada clase. Es una clase persistente.
- **Company**: clase que representa una sede de una compañía donde trabajan los diferentes usuarios del sistema. Es una clase persistente.

- **Address**: representa la dirección física de una sede de una compañía, representando su dirección, ciudad, país, etc.. Es una clase persistente.

- **Project**: es la clase que representa un proyecto software donde trabajan los usuarios del sistema. Es una clase persistente.

- **Session**: es la clase encargada de almacenar toda la información sobre las sesiones que se inicien en el sistema.

- **Knowledge**: es una clase abstracta que representa el conocimiento del sistema. En este caso, son las decisiones tomadas en cada proyecto por los diferentes usuarios que participan en los proyectos. De esta clase heredan las clases concretas *Topic*, *Proposal* y *Answer*, formando la jerarquía de decisiones que se detallará en el apartado 1.3.2.1.2. Es una clase persistente.

- **TopicWrapper**: es una clase que representa el conjunto de temas (*topics*) presentes en un proyecto.

- **File**: clase que representa un fichero que puede adjuntarse a una decisión. Es una clase persistente.

- **Notification**: clase que representa una notificación o alerta que se crea en el sistema cuando se provoca un cambio en las decisiones de un proyecto. Es una clase persistente.

- **PDFElement**: esta clase representa los elementos que van a componer un documento PDF. De ella heredan los diferentes elementos usados en un documento PDF, como son *PDFTable*, *PDFText* y *PDFTitle*. Gracias a esta herencia, se podrá usar el polimorfismo a la hora de componer las secciones del documento PDF.

- **PDFSection**: representa una sección de un documento PDF, compuesta de diferentes elementos.

- **PDFConfiguration**: clase que representa la información de un documento PDF, compuesto de un conjunto de secciones.

- **Operation**: clase que representa las acciones que se pueden realizar en el sistema.

- **LogEntry**: por la especificación de requisitos, se debe mantener un histórico de todas las acciones realizadas en el sistema. Por tanto, esta es la clase que almacena información sobre quién, cuándo y qué acción se ha realizado. Es una clase persistente.

En cuanto a las relaciones entre las clases anteriores, se han modelado las siguientes:

- Un usuario se relaciona con una compañía en la que trabaja, y se relaciona con uno o más proyectos en los que participa. A su vez, en cada proyecto trabajan varios usuarios.
- Una compañía se asocia con una dirección.
- Una sesión se relaciona con un usuario, que es el que se identifica e inicia sesión en el sistema.
- Cada clase que representa una decisión (*Knowledge*) se asocia con un usuario, que es el autor que la crea.
- Cada tema (*Topic*) se asocia con 0 o más propuestas (*Proposal*) y con un proyecto, que es donde se crea.
- Cada propuesta se asocia con 0 o más respuestas (*Answer*).
- El conjunto de temas (*TopicWrapper)* se asocia con 0 o más temas.
- Cada decisión puede tener 0 o más ficheros adjuntos y un fichero puede estar adjunto en 1 o más decisiones.
- Una notificación se asocia con una decisión, un proyecto y uno o más usuarios (todos los que trabajan en ese proyecto).
- Una tabla del documento PDF (*PDFTable*) se asocia con un proyecto, para conocer y generar toda la información de ese proyecto.
- Una sección del documento PDF (*PDFSection*) se compone de uno o más elementos del documento PDF (*PDFElement)*, para componer la sección.
- Un documento PDF (*PDFConfiguration*) se compone de una o más secciones del documento (*PDFSection*).

- La clase *LogEntry* utiliza la clase usuario para consultar el nombre de usuario que ha realizado una acción en el sistema.

En la Figura 1.26 se puede observar este diagrama de clases de dominio.

1.2.2.2 Diseño de la base de datos

A partir de las clases de dominio persistentes y sus relaciones, se modela y diseña la base de datos para que los objetos de dominio puedan ser persistentes. Para realizar esto, se han tenido en cuenta las siguientes consideraciones a la hora de diseñar la base de datos:

- Para representar la jerarquía de herencia de las clases *User*, *Employee* y *ChiefProject*, se ha utilizado el patrón de persistencia **1 árbol de herencia, 1 tabla**, por lo que sólo se creará la tabla *Users*, que agrupará los atributos de todas las entidades anteriores. Sin embargo, es necesario añadir un nuevo atributo a la tabla *Users* para indicar el rol del usuario, correspondiente a cada una de las subclases. La razón de utilizar dicho patrón es agrupar en única tabla toda la jerarquía de herencia que existe entre esas clases en el modelo de dominio, pues ninguna de las clases *Employee* ni *ChiefProject* añaden nuevos atributos a la clase *User*, por lo que no se van a obtener atributos (columnas) nulas en la tabla resultante.

- En la tabla *LogEntries*, obtenida al transformar la clase *LogEntry* en una tabla, la columna ”usuario” puede ser vacía, pues hay acciones que no están asociadas a ningún usuario del sistema, como, por ejemplo, iniciar o detener el servidor.

- Debido a las asociaciones n:m (o *muchos a muchos*) entre las clases de *User* y *Project*, entre las de *Notification* y *User* y entre las de *Knowledge* y *File*, es necesario modelar tablas adicionales que permitan almacenar dichas asociaciones, utilizando claves ajenas a las tablas que representan las clases que participan en esas asociaciones.

- En la tabla *NotificationsUsers*, que modela la relacion *muchos a muchos* entre usuarios y notificaciones, se ha creado un *trigger* para borrar automáticamente una notificación cuando todos los usuarios a los que iba dirigida ya la han borrado. Esto se comentará más en detalle en el apartado 1.3.3.2.1.

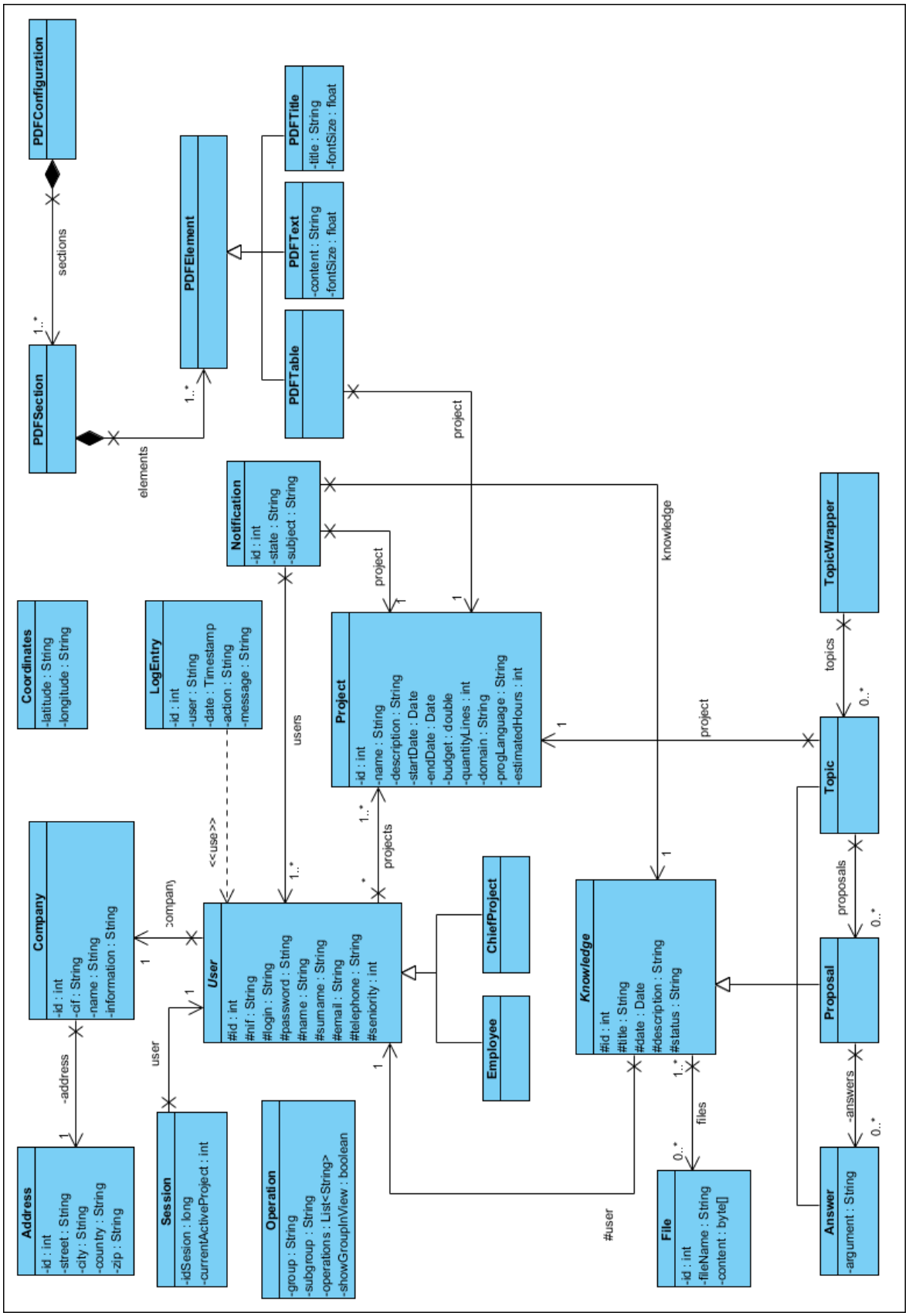

Figura 1.26: Diagrama de clases de dominio

Teniendo en cuenta dichas consideraciones, se obtiene el modelo EER (Entidad-Interrelación Extendido) de la base de datos, mostrado en la Figura 1.27.

Para terminar, cabe destacar que este modelo se ha creado utilizando la herramienta **MySQL Workbench** (ver sección **??**), la cuál permite generar automáticamente el código SQL necesario para crear las tablas y relaciones de la base de datos a partir de ese modelo EER.

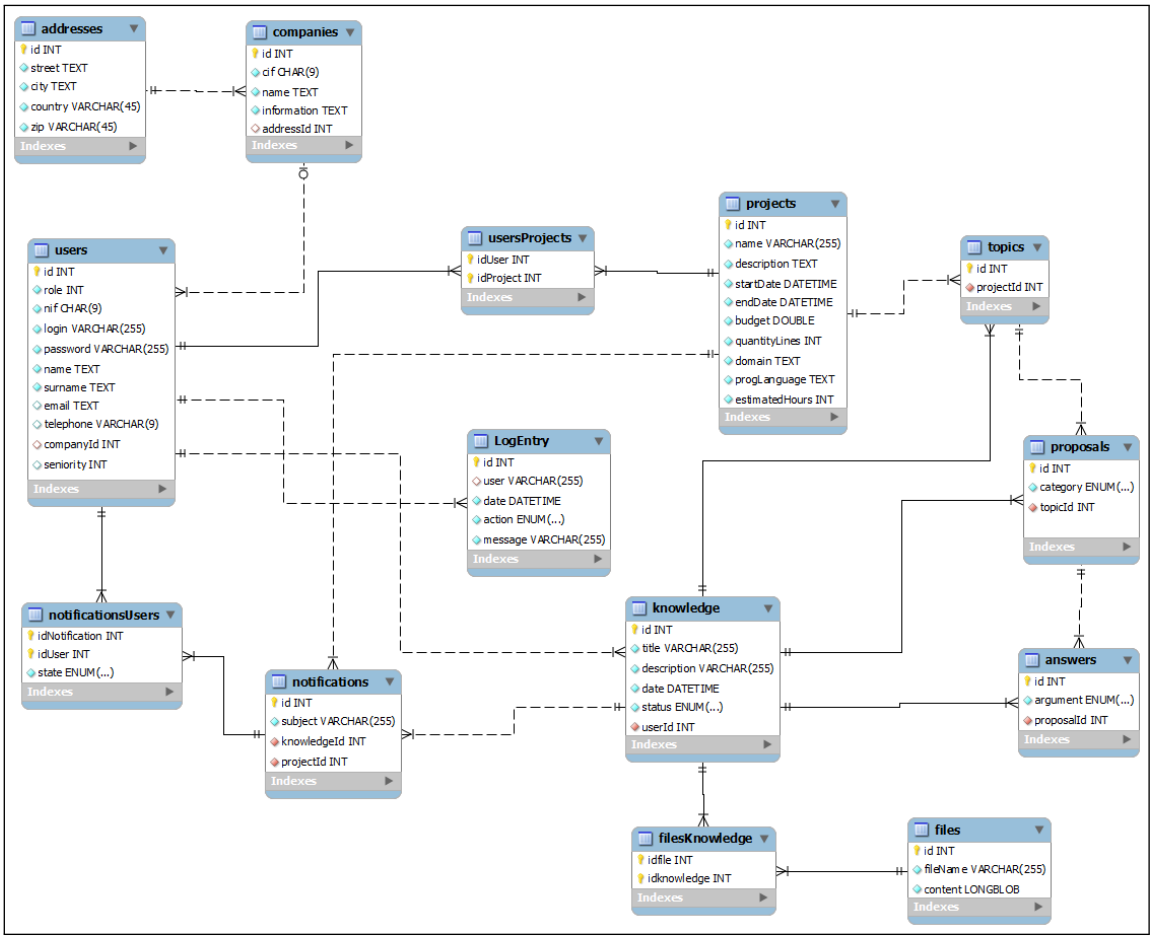

Figura 1.27: Diagrama EER de la base de datos

1.2.2.3 Diseño e implementación de la arquitectura cliente-servidor

Antes de comenzar con la implementación de los grupos funcionales del sistema, es necesario diseñar e implementar la comunicación entre los subsistemas cliente y servidor, siguiendo la arquitectura cliente-servidor definida anteriormente, así como la comunicación entre el servidor y su base de datos (ver Figura 1.23).

Por tanto, a continuación se comentan los aspectos reseñables del diseño e implementación de la comunicación entre sistemas.

1.2.2.3.1 Comunicación entre cliente y servidor

El primer paso es hacer que las clases de dominio mostradas en la Figura 1.26 sean serializables, para poder ser exportadas y enviarse de un sistema a otro. Para ello, dichas clases implementan la interfaz *Serializable* de Java.

A continuación, se diseña e implementa la capa de comunicaciones de ambos sistemas, haciendo uso del patrón **Proxy**, responsable de establecer conexión con las clases remotas exportadas por cada uno de los subsistemas. Además, para conocer los métodos que pueden invocarse utilizando RMI, se crean interfaces para cada uno de los subsistemas, implementando la interfaz *Remote* provista por RMI.

De este modo, el proxy del servidor implementa la interfaz remota del servidor, la cual contiene todas las operaciones que el cliente puede solicitar al servidor, por lo que es un patrón **Fachada**. Así, el cliente puede invocar al proxy del servidor como si éste fuera local y estuviera en la misma máquina. De un modo análogo, se implementa el proxy y la interfaz del cliente, para que el servidor pueda enviar y notificar los resultados al cliente.

En la Figura 1.28 se muestra el diagrama de clases para esta capa de comunicaciones. Para mejorar la legibilidad del diagrama, no se reflejan todos los métodos de estas clases y fachadas, mostrándose en el Apéndice **??** el código fuente completo de estas fachadas, que contienen todas las operaciones que el cliente puede invocar al servidor, y viceversa.

Conviene destacar que, con el fin de que el sistema funcione correctamente si alguno de los subsistemas pertenece a varias redes, al exportar los objetos remotos se recorren todas las interfaces de red para buscar una IP según el siguiente orden:

1. Si el ordenador pertenece a una red pública, se usa una IP pública.

2. Si el ordenador no pertenece a una red pública pero sí a una privada, se utiliza una IP privada.

3. Si el ordenador no está conectado a ninguna red, se emplea la IP *localhost* (127.0.0.1).

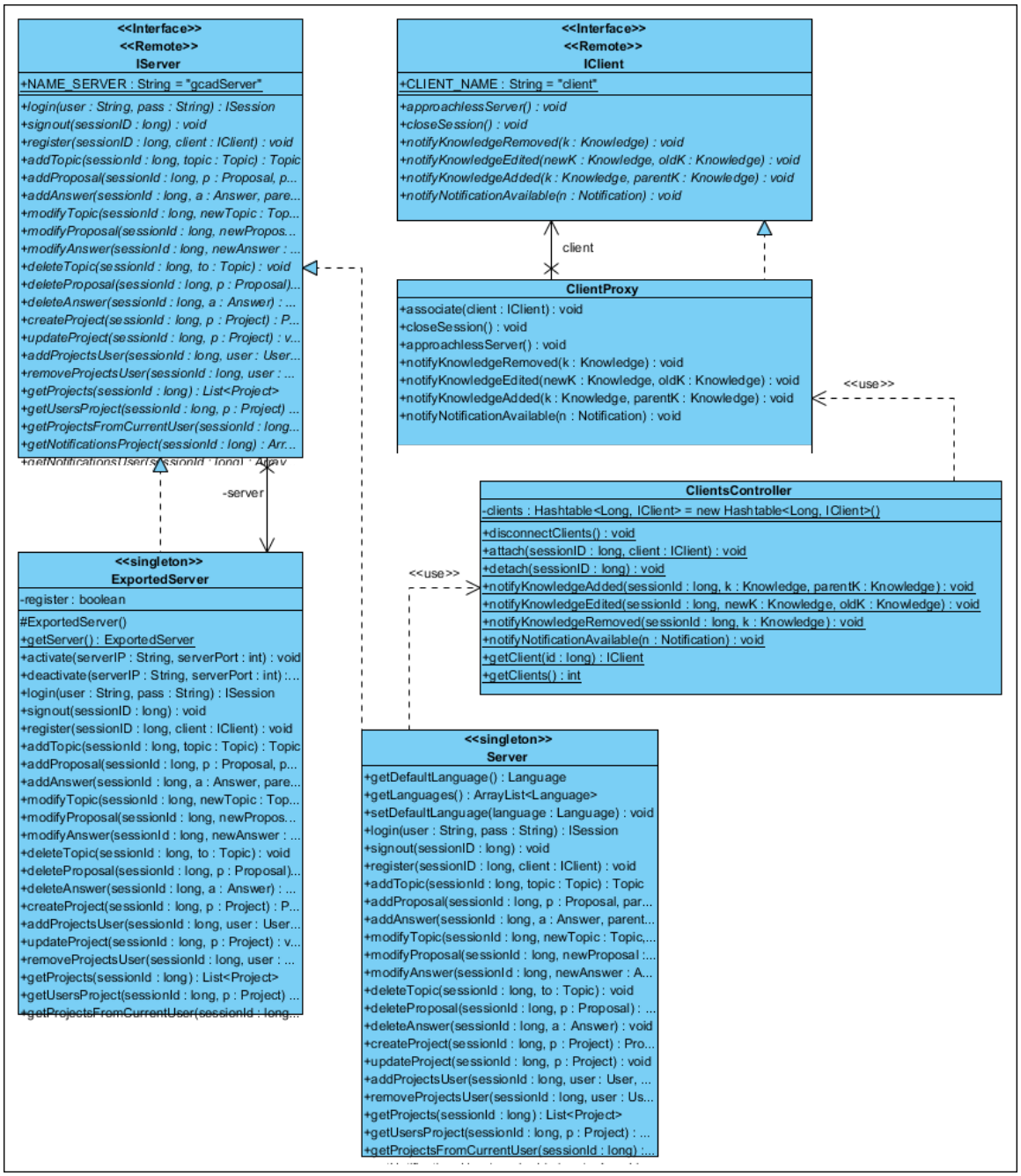

Figura 1.28: Diagrama de clases - Capa de comunicación cliente-servidor

Además, para que la comunicación con los objetos remotos se establezca correctamente, no sólo es necesario indicar la IP al exportar los objetos, sino que también hace falta modificar la propiedad *java.rmi.server.hostname* de la máquina virtual de Java, que representa la IP del servidor RMI que contiene los objetos.

En el fragmento de código 1.1 se muestra como se exporta el objeto que representa al servidor y como se modifica la propiedad *java.rmi.server.hostname* de la máquina virtual de Java, para que el servidor sea accesible y sus métodos puedan ser invocados por el cliente.

```
public class ExportedServer extends UnicastRemoteObject implements IServer {
   ....

   public void activate(String serverIP, int serverPort) throws MalformedURLException,
       RemoteException {
      // If the server is already exports, don't throw exception
      try {
         if(!register) {
            LocateRegistry.createRegistry(serverPort);
            register = true;
         }
         exportObject(this, serverPort);
        } catch(ExportException ex) {
         if(!ex.getMessage().toLowerCase().equals("object already exported")) {
            throw ex;
         }
        }
        try {
            Naming.bind("rmi://" + serverIP + ":" + String.valueOf(serverPort) + "/" +
                NAME_SERVER , this);
        } catch(AlreadyBoundException ex) {
            Naming.rebind("rmi://" + serverIP + ":" + String.valueOf(serverPort) + "/" +
                NAME_SERVER, this);
        }
    }

   ....

public class ServerController {
   public void startServer(ServerConfiguration configuration) throws RemoteException,
       MalformedURLException, SQLException {
      serverIP = CommunicationsUtilities.getHostIP();

      // Indicate to RMI that it have to use the given IP as IP of this host in remote
          communications.
      // This instruction is necessary because if the computer belongs to more than one
```

```
        network, RMI may take a private IP as the host IP
    // and incoming communications won't work
    System.setProperty("java.rmi.server.hostname", serverIP);

    ....
```

Listado 1.1: Proceso para exportar un objeto utilizando RMI

Como se puede observar en el fragmento anterior, para exportar el objeto del servidor se utiliza el método *bind* de la clase *Naming* de RMI, indicando la IP, el puerto y el nombre del objeto exportado. De este modo, en el cliente se utiliza el método *lookup* de la clase *Naming* para localizar ese objeto exportado, utilizando la IP, el puerto y el nombre con el que se exportó. En el fragmento de código 1.2 se muestra un ejemplo de cómo realizar esta acción.

```
public class ProxyServer implements IServer {
   ....

   public void connectServer(String ip, int port) throws MalformedURLException,
       RemoteException, NotBoundException {
      String url;

      url = "rmi://" + ip + ":" + String.valueOf(port) + "/" + NAME_SERVER;
      server = (IServer)Naming.lookup(url);
   }
```

Listado 1.2: Proceso para localizar un objeto remoto utilizando RMI

Para terminar, cabe destacar que a efectos de simplificar la conexión y desconexión del subsistema servidor, se ha creado una pequeña interfaz gráfica de usuario, la cuál permite configurar los parámetros del servidor y su base de datos y poner a la escucha o detener el servidor, de manera sencilla. Sin embargo, el subsistema servidor es totalmente independiente de esta interfaz gráfica y podría gestionarse de otro modo, como, por ejemplo, a través de una línea de comandos, pudiéndose automatizar su ejecución y estar siempre disponible en la máquina donde el servidor se distribuya.

1.2.2.3.2 Comunicación entre servidor y base de datos

En el paquete de comunicaciones también se ha utilizado el patrón **Observador** para crear un gestor de conexiones de bases de datos. De este modo, se consigue extensibilidad en las

comunicaciones con bases de datos, pues haciendo uso de este observador, se podrían añadir más de una base de datos y este observador sería el encargado de enviar las peticiones a todas ellas.

Para ello, se ha creado una interfaz que agrupa las operaciones típicas de una base de datos (*CRUD - Create, Read, Update, Delete*), implementada por las diferentes conexiones a bases de datos que pudieran existir. También se ha creado un gestor de conexiones de bases de datos (el observador), que es el encargado de enviar las peticiones a cada una de esas conexiones, utilizando su interfaz. En la Figura 1.29 se puede observar el diagrama de clases que representa este observador.

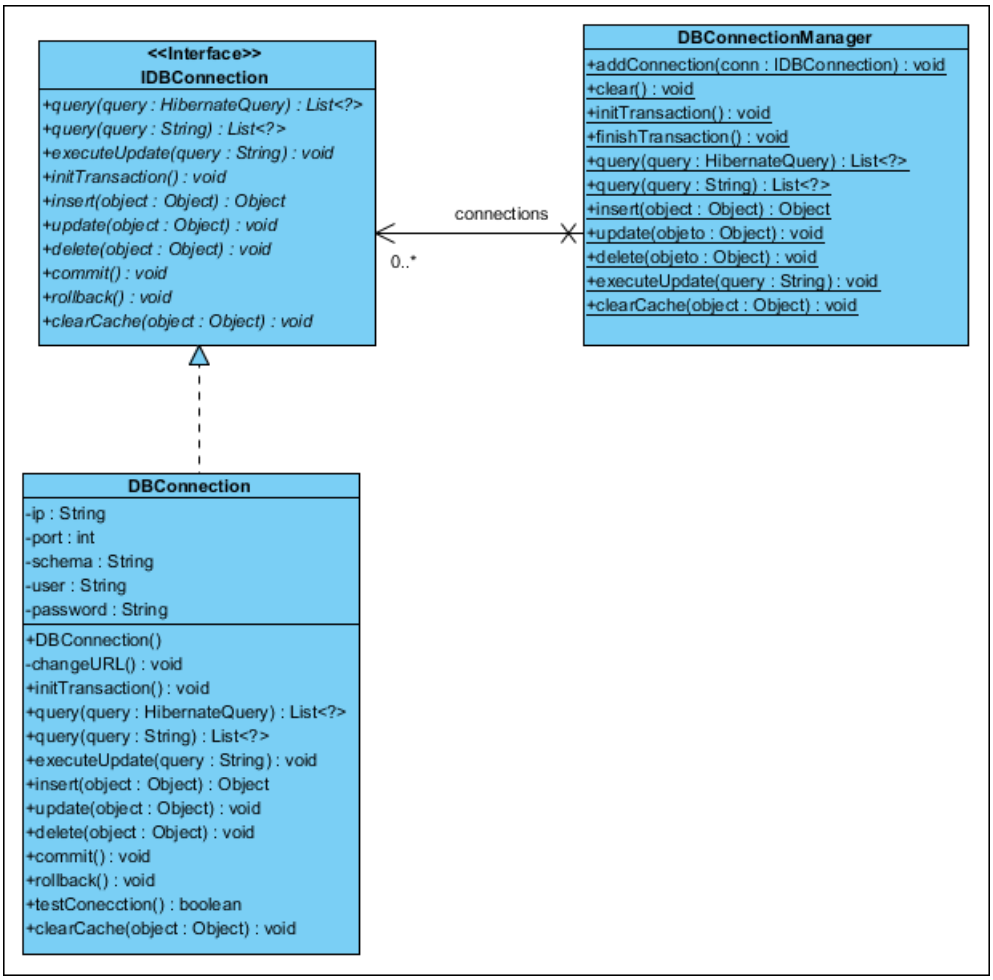

Figura 1.29: Diagrama de clases - Capa de comunicación para bases de datos

En el caso del sistema a desarrollar, sólo existe una base de datos, cuyas operaciones *CRUD* y conexión con la base de datos MySQL es gestionada por **Hibernate**. Cabe señalar algunos problemas que se detectaron al trabajar con RMI e Hibernate:

- Hay que clonar las referencias que devuelve Hibernate, para que sean serializables por RMI.

- Al actualizar un objeto, hay que buscarlo primero en la base de datos, pues la referencia que llega por RMI es diferente a la que utiliza Hibernate.

- Hay que limpiar las cachés que mantiene Hibernate tras hacer una consulta a la base de datos para evitar problemas de referencias.

De un modo similar al anterior, se ha utilizado también el patrón **Observador** para gestionar el *log*, ya que el servidor debe registrar todas las operaciones realizadas por los usuarios. Para ello, se ha creado un gestor de log (el observador) que utiliza interfaces para poder enviar las entradas del log que hay que registrar tanto a la base de datos como a la interfaz gráfica del servidor. De este modo, utilizando este patrón y las interfaces, las entradas de log se crean automáticamente tanto en la base de datos como en la interfaz gráfica, siendo independiente de su implementación. En la Figura 1.30 se puede observar el diagrama de clases que representa este observador.

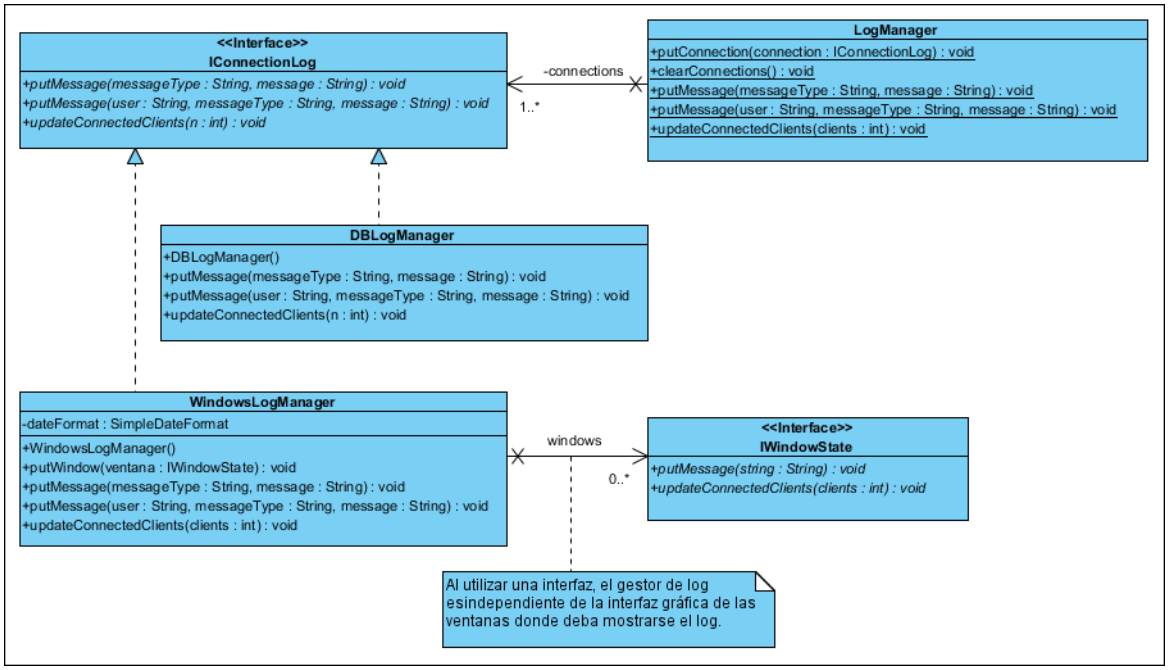

Figura 1.30: Diagrama de clases - Capa de comunicación para gestionar el log

1.2.2.4 Pruebas

1.3 Fase de construcción

En esta fase intervienen principalmente las etapas de Diseño, Implementación y Pruebas con mayor influencia, aunque también se realizan tareas de Análisis para cada iteración, como se muestra en la Figura 1.31. Esta fase, formada por 6 iteraciones, es la fase de más duración del desarrollo del producto software.

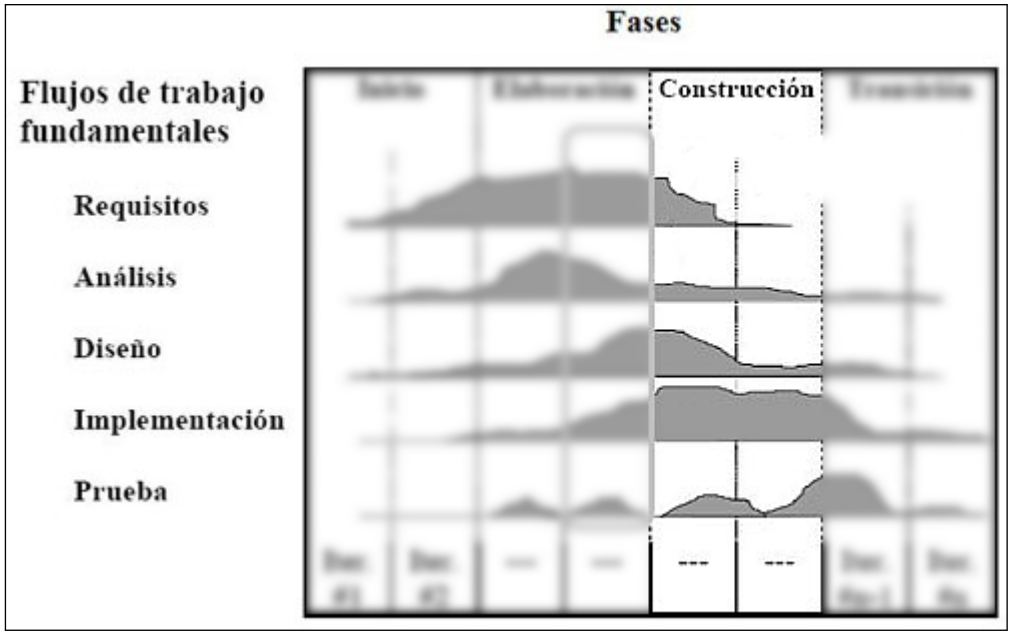

Figura 1.31: Fase de construcción en el PUD

A lo largo de las iteraciones de esta fase, se realiza el análisis, diseño, implementación y pruebas de los grupos funcionales planificados para cada iteración en el plan de iteraciones (ver Tabla **??**), obteniendo incrementos en la funcionalidad del sistema hasta llegar a tener el producto totalmente implementado y probado. Por tanto, se incluirán solamente diagramas de análisis, secuencia de diseño y de clases de diseño para aquellos aspectos más destacables del sistema.

1.3.1 Iteración 3

Siguiendo los casos del grupo funcional *F1: Acceso al sistema* (ver Figura 1.12), se abordan las siguientes tareas en esta primera iteración de la fase de Construcción:

- Análisis de los casos de uso.
- Diseño de la funcionalidad relativa de acceso al sistema.
- Implementación de dicha funcionalidad.
- Diseño e implementación de pruebas relativas a la funcionalidad de acceso al sistema.

1.3.1.1 Grupo funcional F1: *Acceso al sistema*

Una vez se ha definido e implementado la arquitectura del sistema, se han modelado los objetos de dominio y se ha diseñado la base de datos a utilizar en el sistema, se pasa a desarrollar los casos de uso que componen la funcionalidad de *Acceso al sistema*, mostrados en la Figura 1.8.

1.3.1.1.1 Análisis de casos de uso

Los casos de uso englobados en este grupo funcional se especifican de manera más formal, describiendo los escenarios de funcionamiento de cada uno de ellos. Además, para cada caso de uso, se crea el diagrama de clases de análisis, donde se representan los objetos de dominio involucrados en el caso de uso, así como las clases encargadas de su control.

Existen tres tipos de clases en un diagrama de clases de análisis:

1. **Interfaz o *Boundary***: representa clases utilizadas para la comunicación entre el actor y el sistema.
2. **Control**: representa clases y objetos de control. Es la que controla y coordina el flujo (o escenario) de funcionamiento del caso de uso.
3. **Entidad o *Entity***: son clases que representan objetos de información persistentes del sistema.

Señalar que, para cada caso de uso, se describe su funcionamiento y sus flujos desde un punto de vista global al sistema, sin entrar en detalle en cada subsistema, ya que lo que interesa es describir a alto nivel su funcionamiento. Posteriormente, con los diagramas de

secuencia y de clases de diseño, se irá profundizando más en detalle en el flujo que se sigue en cada subsistema.

En este caso, y del mismo modo para el resto de grupos funcionales del sistema, se incluyen las especificaciones y diagramas de análisis de los principales casos de uso, para evitar excederse en la longitud del capítulo.

Login

En la Tabla 1.8 se describe el caso de uso *Login*.

En la Figura 1.32 se muestra el diagrama de clases de análisis para el subsistema cliente. La Figura 1.33 refleja el diagrama de clases de análisis para el subsistema servidor.

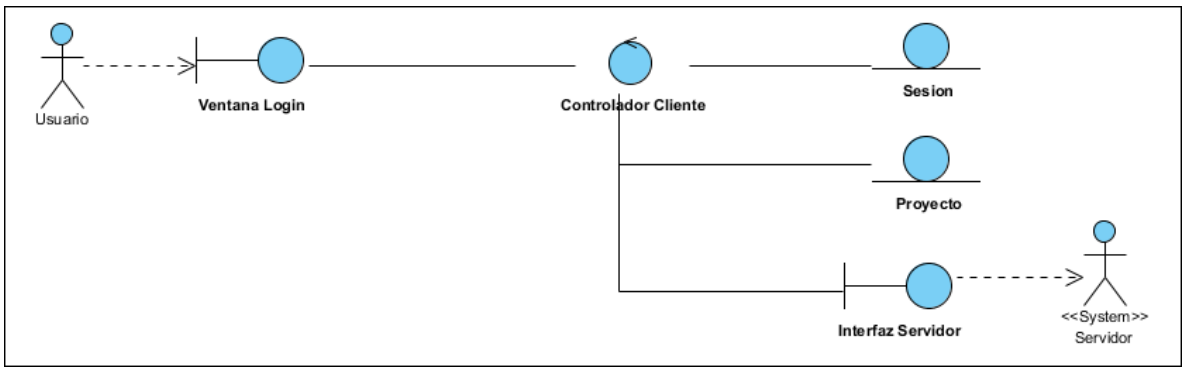

Figura 1.32: Diagrama de clases de análisis - Cliente - Login

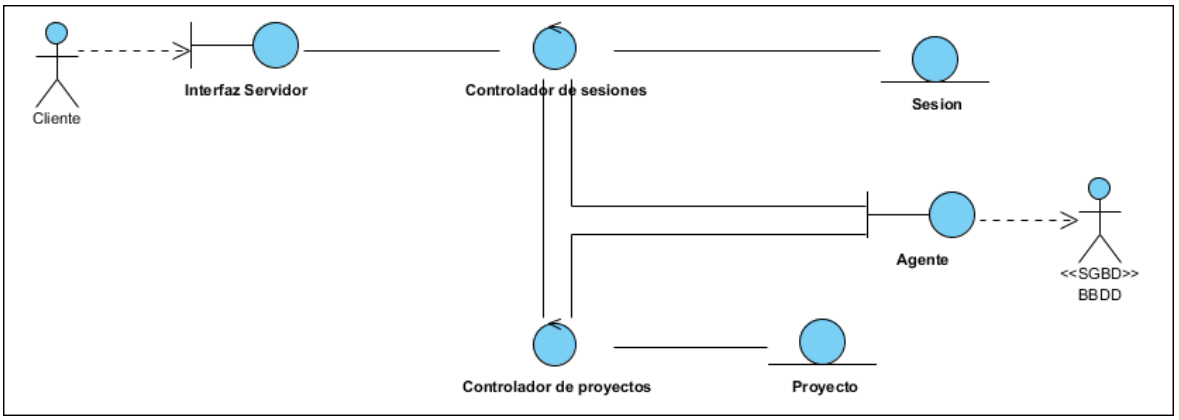

Figura 1.33: Diagrama de clases de análisis - Servidor - Login

Nombre: Iniciar sesión (Login)
Descripción: Funcionalidad para que un usuario pueda acceder al sistema
Precondiciones: Disponer de nombre de usuario y contraseña y estar dado de alta en al menos un proyecto
Post-condiciones: El usuario accede al sistema
Flujo principal: 1. El usuario introduce usuario y contraseña. 2. Se valida el usuario y contraseña. 3. Se crea una sesión para el usuario. 4. Se muestran los proyectos en los que el usuario está dado de alta. 5. El usuario elige uno de los proyectos. 6. El usuario accede al sistema para trabajar sobre ese proyecto.
Flujo alternativo 1: error de Login: 1. El usuario introduce usuario y contraseña. 2. Se valida el usuario y contraseña. 3. El usuario no es válido. Se informa del error y se vuelve al comienzo.
Flujo alternativo 2: usuario inexistente: 1. El usuario introduce usuario y contraseña. 2. Se valida el usuario y contraseña. 3. El usuario no existe. Se informa del error y se vuelve al comienzo.
Flujo alternativo 3: usuario ya logueado: 1. El usuario introduce usuario y contraseña. 2. Se valida el usuario y contraseña. 3. Ya existe una sesión para ese usuario. Se cierra dicha sesión y se crea una nueva. 4. Se muestran los proyectos en los que el usuario está dado de alta. 5. El usuario elige uno de los proyectos. 6. El usuario accede al sistema para trabajar sobre ese proyecto.
Flujo alternativo 4: no existen proyectos: 1. El usuario introduce usuario y contraseña. 2. Se valida el usuario y contraseña. 3. Se crea una sesión para el usuario. 4. Se consultan los proyectos en los que el usuario está dado de alta. 5. El usuario no tiene ningún proyecto. Se informa del error y se vuelve al comienzo.

Tabla 1.8: Especificación del caso de uso *Login*

1.3.1.1.2 Diseño e implementación de acceso al sistema

Una vez realizado el análisis de los casos de uso, identificados los objetos de dominio que en ellos intervienen y definidos sus escenarios, se modela el funcionamiento de dichos casos de uso que componen este grupo funcional a través de diagramas de secuencia de diseño, tanto para el subsistema cliente como para el servidor. De este modo, en la Figura 1.34 se muestra el diagrama de secuencia para el caso de uso *Login* en el cliente, mientras que la Figura 1.35 refleja el diagrama de secuencia para el servidor.

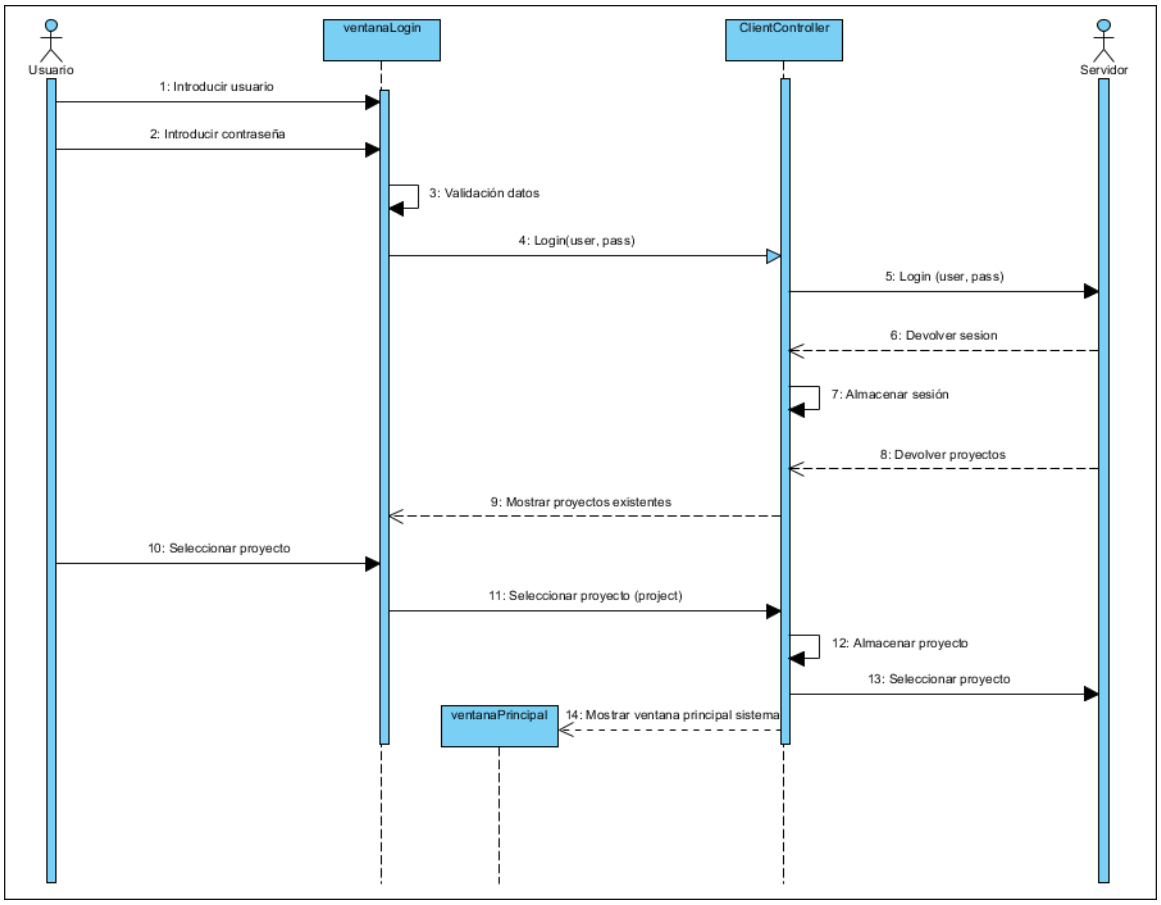

Figura 1.34: Diagrama de secuencia - Cliente - Login

Una vez modelados los diagramas de secuencia, se procede a la implementación de los casos de uso en ambos subsistemas, con algunos aspectos reseñables.

Servidor

La clase *Usuario* es abstracta ya que no se van a instanciar objetos directamente de esa clase, sino de una de sus subclases, que representan los diferentes usuarios que pueden existir en el

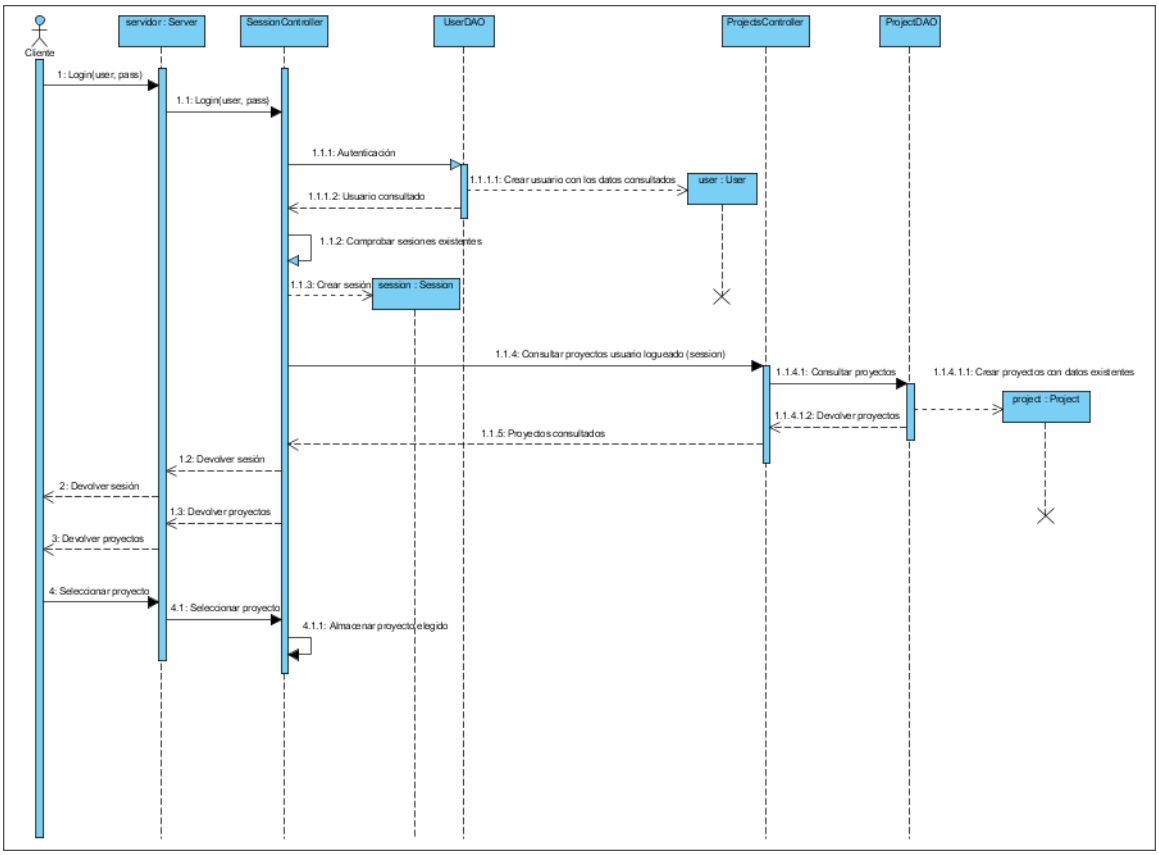

Figura 1.35: Diagrama de secuencia - Servidor - Login

sistema, según la especificación de requisitos (ver sección 1.1.1.1), como se observa en el diagrama de clases de la Figura 1.26.

Además, esta solución propuesta facilita la extensibilidad futura del sistema, ya que el rol de cada usuario se define mediante una enumeración, de tal modo que cada subclase de la clase Usuario redefinirá el método abstracto *getRol()*, devolviendo el valor correspondiente a su rol en la enumeración. Por tanto, para añadir un nuevo rol de usuario, basta con añadir una nueva clase que herede de la superclase y un nuevo rol a la enumeración.

Por otra parte, para aumentar la seguridad del sistema, se decidió encriptar la contraseña de los usuarios, utilizando para ello el algoritmo *SHA1*. Algunas de las razones para utilizar una encriptación por código y no delegar esta responsabilidad en el sistema gestor de base de datos, son las siguientes:

1. Si se quiere cambiar la encriptación a una más segura, no haría falta más que cambiar el método que encripta la contraseña.
2. Puede que otros SGBD que no sean MySQL no tengan encriptación incorporada.
3. El número de encriptaciones que incorpora un SGBD es limitado.

Tras acceder al sistema, la interfaz gráfica de usuario del subsistema cliente debe adaptarse a las operaciones del usuario logueado, según su rol. Para ello, el gestor de sesiones utiliza archivos XML donde están definidos los perfiles existentes en el sistema y las operaciones que puede realizar este perfil. Así, las operaciones se han dividido en categorías, de tal modo que los elementos de los menús de la aplicación, submenús, *toolbars*, etc, se generarán de manera automática en el cliente tras acceder al sistema y conocer las operaciones que puede realizar un cierto rol.

De este modo, además de permitir una interfaz de usuario totalmente flexible y adaptable, se facilita la extensibilidad, ya que, además de los cambios mencionados anteriormente, solamente habría que añadir el nuevo perfil y sus operaciones a los ficheros XML alojados en el servidor. Para gestionar estos archivos XML, se ha utilizado el patrón **Agente** para encapsular en un clase el acceso a los ficheros XML y todas las operaciones a realizar con ellos, como consultas al archivo XML, gestión de XPath, etc, utilizando JDOM y Jaxen (ver sección **??**).

En la Figura 1.36 se muestra el diagrama de clases para el gestor de sesiones.

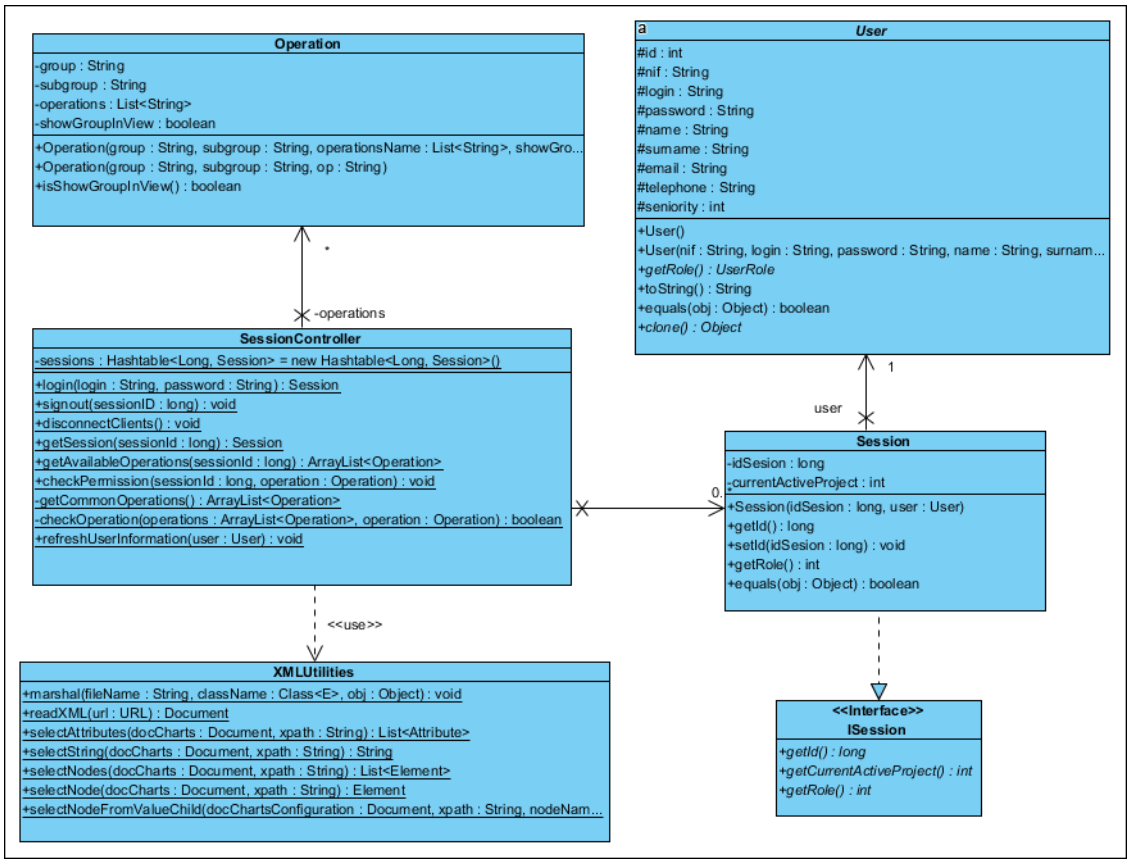

Figura 1.36: Diagrama de clases - Gestor de sesiones

Cliente

Cabe destacar que la interfaz gráfica de usuario del subsistema cliente se ha diseñado e implementado siguiendo una estructura de composición de clases, de tal modo que la ventana (o *frame)* principal se compone de paneles, y éstos, a su vez, de otros paneles y elementos gráficos. De este modo se consigue una interfaz gráfica ampliamente extensible y adaptable.

Gracias a esta estructura, según el rol del usuario que accede al sistema, los menús y diferentes elementos de la interfaz gráfica son fácilmente adaptables a las operaciones que ese usuario pueda realizar en el sistema. Además, para tener en cuenta el multi-idioma, la interfaz se adaptará al idioma elegido y todos sus menús, etiquetas, texto, etc, se mostrarán en el idioma preferido. Este aspecto de internacionalización se detallará en una iteración posterior.

Para terminar, en lo que respecta a esta funcionalidad de acceso al sistema en el cliente, se ha diseñado e implementado una ventana para que el usuario pueda introducir sus datos, validarlos y enviarlos al servidor, accediendo al sistema si todo es correcto. En este aspecto, cabe destacar el uso del método *invokeLater* de la librería Swing (ver sección **??**) que permite manipular la interfaz gráfica mientras se realiza una tarea de larga duración en un hilo separado. Por tanto, la acción de validar los datos para acceder al sistema y enviarlos al servidor, se realiza en un hilo separado, mientras que en la interfaz gráfica se muestra un panel con un *spinner* de carga (ver ejemplo de *spinner* en la Figura 1.37) animado, para proporcionar al usuario un *feedback* visual y saber que la operación se está realizando y que debe esperar.

Figura 1.37: Ejemplo de *spinner* de carga

En el fragmento de código 1.3 se muestra como se ha utilizado el método *invokeLater* para delegar en un nuevo hilo el *login* en el sistema, mostrando el panel con el *spinner* animado.

```
this.glassPane = new InfiniteProgressPanel(ApplicationInternationalization.getString("
    glassLogin"));
 getMainFrame().setGlassPane(glassPane);

....
```

```
@Action
public void loginAction() {
   ....

   // Invoke a new thread in order to show the panel with the loading
   // spinner
   SwingUtilities.invokeLater(new Runnable() {
      public void run() {
         glassPane.setColorB(241);
         glassPane.start();
         Thread performer = new Thread(new Runnable() {
            public void run() {
               perform(txtUserName.getText(), new String(txtPass.getPassword()), ip,
                   port);
            }
         }, "Performer");
         performer.start();
      }
   });
}

private void perform(String user, String pass, String ip, String port) {
   // Login
   ClientController.getInstance().initClient(ip, port, user, pass);
   glassPane.stop();
   getMainFrame().setEnabled(true);
   getMainFrame().requestFocus();

   ....
}
```

Listado 1.3: Fragmento de código para acceder al sistema en el cliente

1.3.2 Iteración 4

Siguiendo los casos de uso del grupo funcional **F3:** *Visualización información* (ver Figura 1.12), se abordan las siguientes tareas:

- Análisis de los casos de uso.
- Diseño de la funcionalidad relativa a la visualización de información en el sistema, como son las decisiones y su información asociada.
- Implementación de dicha funcionalidad.

- Diseño e implementación de pruebas relativas a la visualización de información.

1.3.2.1 Grupo funcional F3: *Visualización información*

1.3.2.1.1 Análisis de casos de uso

Este grupo funcional constituye una de las funcionalidades principales del sistema, ya que se encarga de mostrar y representar todas las decisiones realizadas en un proyecto, así como su información asociada (datos del usuario que realizó la decisión, compañía a la que pertenece, etc). Por tanto, esta iteración comienza con el análisis de los casos de uso que componen dicha funcionalidad, definiendo sus escenarios y diagramas de análisis.

Visualizar decisiones

En la Tabla 1.9 se describe el caso de uso *Visualizar decisiones*.

Nombre: Visualizar decisiones
Descripción: Funcionalidad para visualizar las decisiones (y su información relacionada) de un proyecto.
Precondiciones: Que el usuario haya accedido al sistema y tenga permisos para realizar la operación.
Post-condiciones: Se consultan las decisiones de un proyecto y se visualizan.
Flujo principal: 1. El usuario inicia la acción para consultar las decisiones de un proyecto. 2. El sistema consulta las decisiones del proyecto. 3. El sistema consulta la información asociada a las decisiones. 4. Se muestran las decisiones encontradas. 5. Se muestra información asociada a las decisiones.
Flujo alternativo 1: no existen decisiones: 1. El usuario inicia la acción para consultar las decisiones de un proyecto. 2. El sistema consulta las decisiones del proyecto. 3. Se muestra mensaje de que no hay ninguna decisión.

Tabla 1.9: Especificación del caso de uso *Visualizar decisiones*

La Figura 1.38 representa el diagrama de clases de análisis para el subsistema cliente. En la Figura 1.39 se muestra el diagrama de clases de análisis para el subsistema servidor.

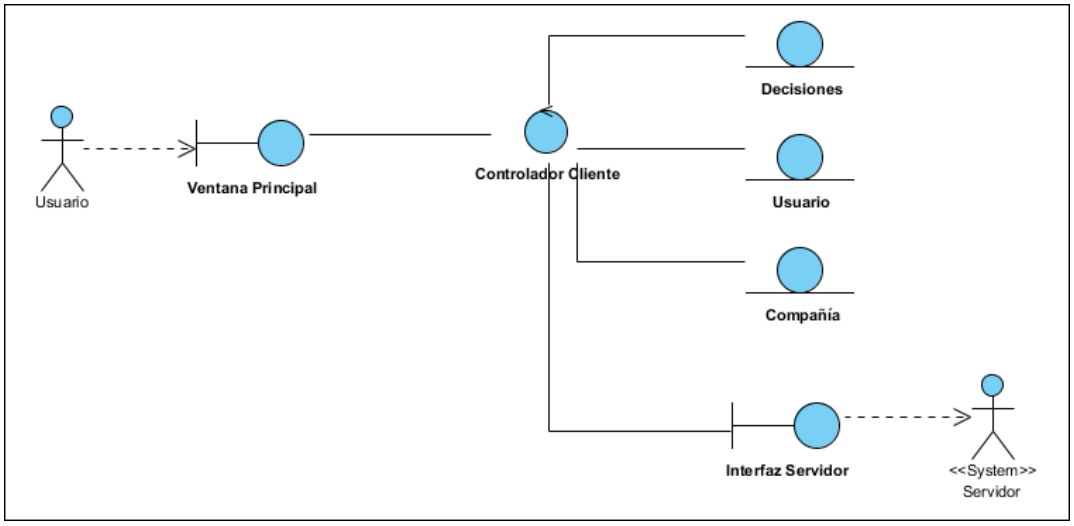

Figura 1.38: Diagrama de clases de análisis - Cliente - Visualizar Decisiones

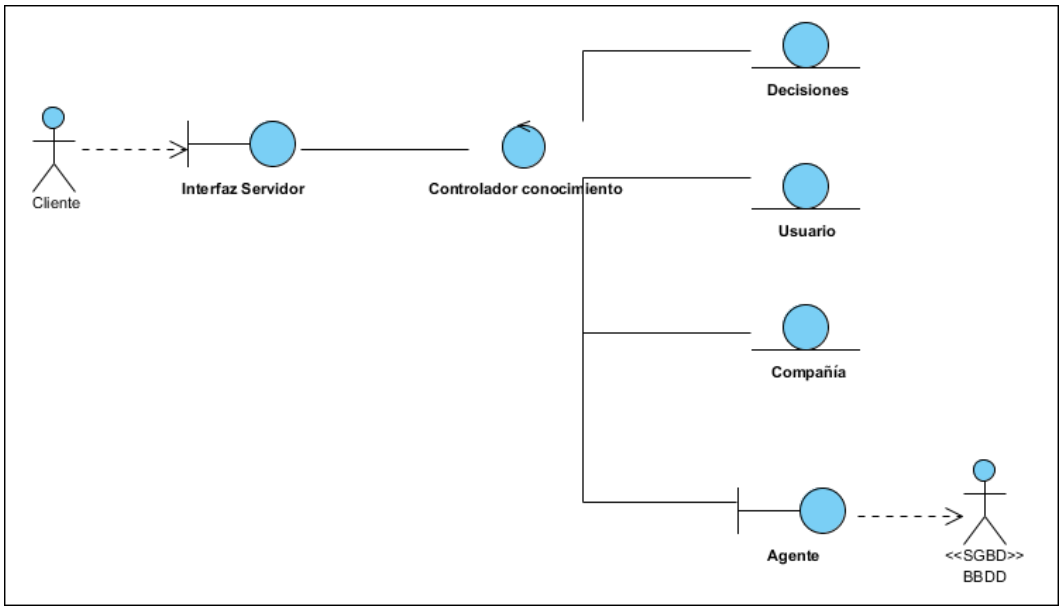

Figura 1.39: Diagrama de clases de análisis - Servidor - Visualizar Decisiones

1.3.2.1.2 Diseño e implementación

En primer lugar, se modela el funcionamiento de los casos de uso que componen este grupo funcional a través de diagramas de secuencia de diseño. De este modo, en la Figura **??** se muestra el diagrama de secuencia para el caso de uso *Visualizar decisiones* en el cliente,

mientras que la Figura 1.35 refleja el diagrama de secuencia para el servidor. El resto de diagramas de secuencia de este grupo funcional se modelan de un modo muy similar a los diagramas mostrados.

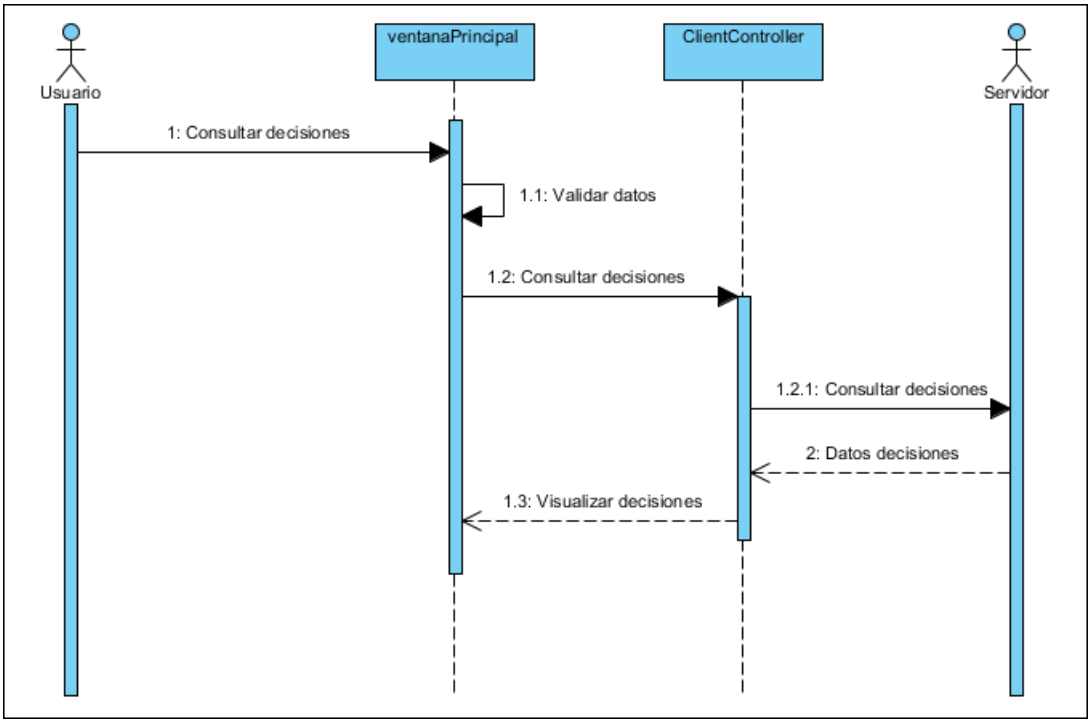

Figura 1.40: Diagrama de secuencia - Cliente - Consultar decisiones

Posteriormente, siguiendo los diagramas de secuencia, se realizan los diagramas de clases de diseño y se procede a la implementación de los casos de uso, para obtener las clases Java que dan soporte a estas funcionalidades. A continuación se destacan algunos aspectos tenidos en cuenta al diseñar e implementar los casos de uso que componen este grupo funcional del sistema.

Servidor

Cabe destacar que para mantener la seguridad en el sistema, siempre se comprobará en el servidor la sesión del usuario antes de realizar cualquier operación, tanto para comprobar que esa sesión existe como para validar si se tiene permiso para ejecutar dicha acción en el sistema. Por ello, los gestores (o controladores) de ésta y del resto de funcionalidades del sistema tendrán una relación con el gestor de sesiones, descrito en la segunda iteración de la fase anterior (ver Figura 1.36).

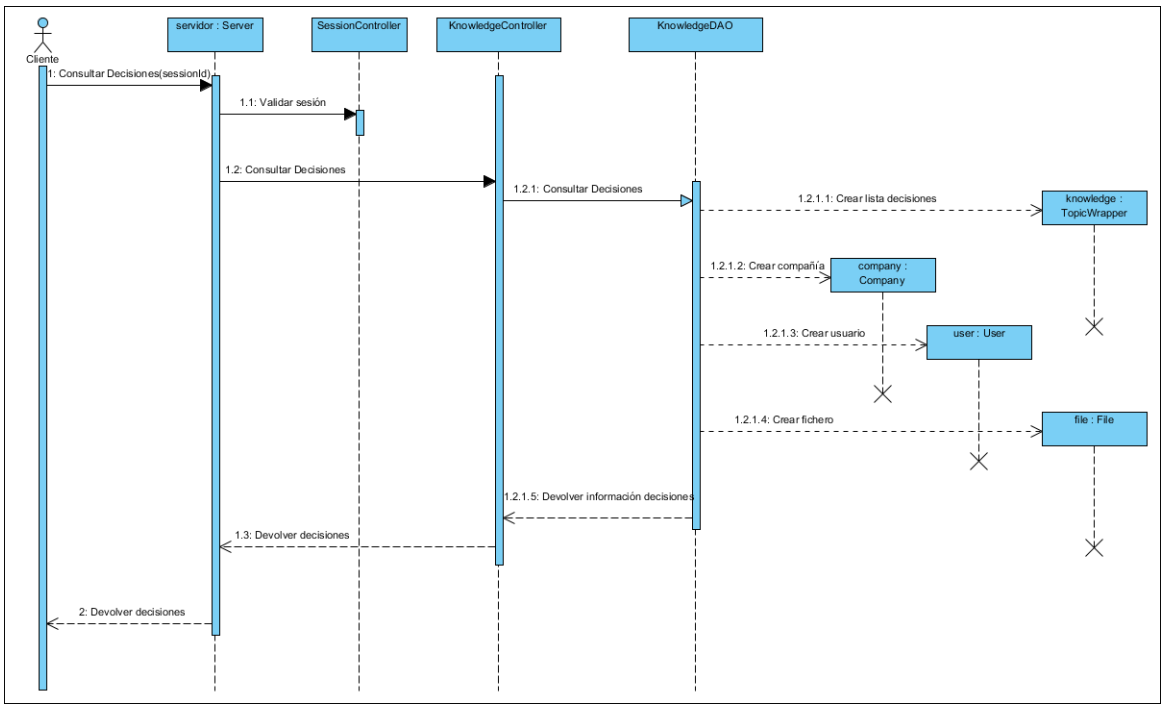

Figura 1.41: Diagrama de secuencia - Servidor - Consultar decisiones

Como se puede apreciar en el diagrama de clases de la Figura 1.42, las decisiones de un proyecto siguen una jerarquía de herencia, existiendo la clase abstracta *Knowledge* de las que heredan otras tres clases: *Topic*, *Proposal* y *Answer*. Se ha decidido diseñar esta herencia para facilitar la futura extensibilidad del sistema, ya que se podría incluir un nuevo tipo de conocimiento creando una nueva subclase. Además, para mejorar la organización de las decisiones, estas tres subclases están asociadas de una manera jerárquica, del siguiente modo:

- **Topic** (o Tema): esta clase, que representa un Tema, está compuesta por un conjunto de decisiones del tipo *Proposal* dentro de un proyecto.
- **Proposal** (o Propuesta): esta clase, que representa una Propuesta, esta compuesta por un conjunto de decisiones del tipo *Answer*, y están agrupadas bajo un *Topic* común.
- **Answer** (o Respuesta): esta clase, que representa una Respuesta están agrupadas bajo una *Proposal* común y ya no pueden tener más hijos.

De este modo, como se muestra en la Figura 1.42, para cada proyecto, existirán una serie de *Topics*, donde cada *Topic* estará compuesto por una serie de *Proposals* y éstas, a su vez, estarán compuestas por un conjunto de *Answers*. Por tanto, se sigue una estructura de

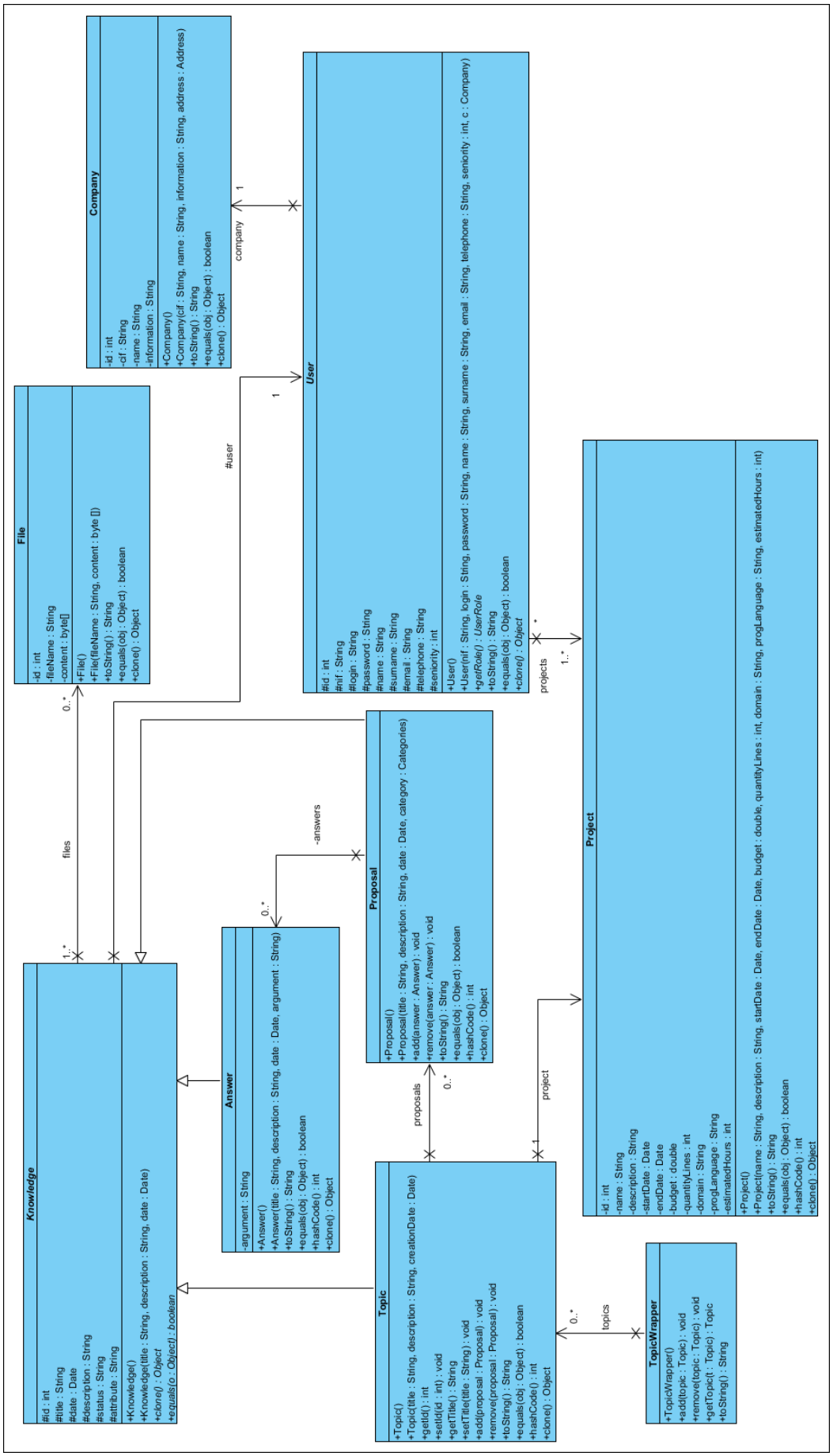

Figura 1.42: Diagrama de clases - Jerarquía de decisiones

composición y jerarquía de conocimiento. Un ejemplo de esta estructura jerárquica puede encontrarse en los foros de debate presentes en la Web (ver Figura 1.43).

Figura 1.43: Ejemplo de jerarquía de decisiones en foros de debates

Además, existe la clase *TopicsWrapper*, que engloba todos los *Topic* de un proyecto, facilitando su recuperación y necesario además para exportar todo el conocimiento a un fichero XML, como se detallará en el posterior apartado 1.3.6.1.

Esta jerarquía de decisiones, junto con toda su información y clases relacionadas, se considera como uno de los puntos de más importancia del presente PFC, ya que muchas del resto de funcionalidad del sistema giran en torno a las decisiones tomadas en los proyectos y a su información relacionada y utilizan las relaciones y jerarquía diseñadas en este punto para cumplir con su objetivo.

Siguiendo esta estructura descrita, en lo que respecta al caso de uso *Visualizar Decisiones*, se recupera y visualiza esta composición jerárquica de decisiones y toda su información asociada, como es el usuario, la compañía y los posibles ficheros adjuntos (ver diagrama de la Figura 1.42). Sin embargo, al utilizar el framework de persistencia *Hibernate*, no hace falta hacer varias consultas para recuperar toda esta información, como se muestra en el diagrama de secuencia de la Figura 1.41, ya que al existir asociaciones entre los objetos, Hibernate se encargará de recuperar todos ellos al consultar las decisiones existentes en un proyecto.

En lo que concierne al caso de uso *Visualizar Compañía*, que sirve para consultar y mostrar los detalles de una compañía, cabe destacar como aspecto de implementación que se ha utilizado un servicio Web para obtener la posición geográfica de la dirección de la

compañía, para poder mostrarla posteriormente en un mapa, en el subsistema cliente. Para ello, como se vió en el marco tecnológico de la sección **??**, se ha utilizado el servicio web **Yahoo! PlaceFinder** para obtener las coordenadas geográficas a partir de una dirección.

En un primer momento se pensó en utilizar el API de geolocalización de Google para realizar esta tarea. Sin embargo, atendiendo a las condiciones de uso de Google [**?**], esta API debe usarse en conjunto con el API de *Google Maps*, que a su vez sólo se puede utilizar en aplicaciones Web. Por tanto, se descartó esta opción para no violar las condiciones de uso de Google y se buscó una alternativa, encontrando el servicio web de Yahoo!, cuyos términos de uso permitían utilizarlo en el sistema [**?**].

Para invocar el servicio *Yahoo! PlaceFinder*, se necesita una URL del tipo:

```
http://where.yahooapis.com/geocode?[parameters]&appid=APPID
```

donde *APPID* es un código proporcionado para desarrolladores para poder utilizar su API, y *parameters* es la dirección a buscar.

Un ejemplo de URL sería:

```
http://where.yahooapis.com/geocode?location=701+First+Sunnyvale&APPID
```

Al invocar el servicio web, éste devuelve una respuesta en formato XML indicando una serie de parámetros, como se muestra en el Listado 1.4. Entre ellos, interesa el código de error y las coordenadas geográficas de la dirección, formadas por longitud y latitud.

```
<?xml version="1.0" encoding="UTF-8"?>
<ResultSet version="1.0">
   <Error>0</Error>
   <ErrorMessage>No error</ErrorMessage>
   <Locale>us_US</Locale>
   <Quality>87</Quality>
   <Found>1</Found>
   <Result>
      <quality>87</quality>
      <latitude>37.416275</latitude>
      <longitude>-122.025092</longitude>
      <offsetlat>37.416397</offsetlat>
      <offsetlon>-122.025055</offsetlon>
      <radius>500</radius>
      <name>
```

```
      </name>
      <line1>701 1st Ave</line1>
      <line2>Sunnyvale, CA  94089-1019</line2>
      <line3>
      </line3>
      <line4>United States</line4>
      <house>701</house>
      <street>1st Ave</street>
      <xstreet>
      </xstreet>
      <unittype>
      </unittype>
      <unit>
      </unit>
      <postal>94089-1019</postal>
      <neighborhood>
      </neighborhood>
      <city>Sunnyvale</city>
      <county>Santa Clara County</county>
      <state>California</state>
      <country>United States</country>
      <countrycode>US</countrycode>
      <statecode>CA</statecode>
      <countycode>
      </countycode>
      <uzip>94089</uzip>
      <hash>DDAD1896CC0CDC41</hash>
      <woeid>12797150</woeid>
      <woetype>11</woetype>
   </Result>
</ResultSet>
```

Listado 1.4: Respuesta del servicio Web Yahoo! PlaceFinder

En el Listado 1.5 se muestra un fragmento de código utilizado para invocar el servicio web desde Java y obtener su respuesta XML, que será tratada con JDOM para extraer los datos necesarios (las coordenadas) para poder mostrar en un mapa la posición exacta de la compañía.

```
/**
 * Class used to obtain geographic coordinates from an address. To do so, uses the Web
     Service "Yahoo! PlaceFinder"
 */
public class GeoCoder {

   // Yahoo! application ID
```

```
   private static final String APPID = "NzaqCw38";
   // Yahoo! Web Service base url
   private static final String BASE_URL =  "http://where.yahooapis.com/geocode";

   public static Coordinates getGeoCoordinates(Address address) throws
       NonExistentAddressException, WSResponseException, IOException, JDOMException {
      ....

      String request = BASE_URL + "?q="+ad.toString()+"&appid="+APPID;
      url = new URL(request);
      // Connect and get response stream from Web Service
      InputStream in = url.openStream();
      // The response is an XML
      SAXBuilder builder = new SAXBuilder();
      // Uses JDOM and XPath in order to parser Xml
      Document doc;
      doc = builder.build(in);
      // Get values from XML using XPath
      String status = ((Element) XPath.selectSingleNode(doc, "/ResultSet/Error")).
      getContent(0).getValue();
      if (!status.equals("0"))
         throw new WSResponseException();

      String found = ((Element) XPath.selectSingleNode(doc, "/ResultSet/Found")).getContent
          (0).getValue();
      if (found.equals("0"))
         throw new NonExistentAddressException(AppInternationalization.
               getString("AddressNotFound_Exception"));

      String latitude = ((Element) XPath.selectSingleNode(doc, "/ResultSet/Result/latitude"
          )).getContent(0).getValue();
      String longitude = ((Element) XPath.selectSingleNode(doc, "/ResultSet/Result/
          longitude")).getContent(0).getValue();
      in.close();
      coor = new Coordinates(latitude, longitude);
      return coor;
   }
}
```

Listado 1.5: Invocación del servicio Web Yahoo! PlaceFinder

Cliente

En el subsistema cliente, cuando se consultan las decisiones, éstas se muestran de manera jerárquica (en árbol) y en forma de grafo, utilizando **JUNG** (ver sección **??**). Se ha decidido utilizar ambos tipos de visualización para ofrecer una mayor flexibilidad e información al

usuario, ya que gracias a la representación jerárquica, se puede apreciar toda la estructura de decisiones de un simple vistazo, mientras que con el grafo, se pueden observar cómo están asociadas dichas decisiones, mostrándose de manera mas detallada, siguiendo el modo de representación de *Rationale* llamado **Dialogue Map**, mencionado en la sección **??**. Además de mostrar las decisiones, también se muestra toda su información asociada, utilizando paneles expandibles, para flexibilizar y personalizar la información que en cada momento el usuario quiere mostrar.

Un tipo de información que se muestra para cada decisión son los archivos adjuntos que tiene, que pueden ser descargados por el usuario, guardando una copia local que envía el sistema servidor al hacer la petición.

Para terminar, en lo que se refiere al caso de uso de *Visualizar compañía*, se ha decidido mostrar en un mapa la posición geográfica (además de otros datos) de la compañía consultada, para poder conocer su información exacta y real. Para ello se han utilizado los mapas proporcionados por **OpenStreetMaps** (ver sección **??**), ya que, como se comentó anteriormente, debido a los términos de uso de Google, los mapas proporcionados por *Google Maps* sólo pueden utilizarse en aplicaciones Web, por lo que se decidió optar por esta alternativa.

Para mostrar los mapas proporcionados por *OpenStreetMaps*, se ha utilizado un contenedor gráfico llamado **JXMapKit**, proporcionado por la librería *Swingx*, el cuál puede ser configurado para proporcionarle un proveedor de mapas para visualizar. Además, una vez calculadas las coordenadas geográficas en el sistema servidor, este elemento permite añadir un marcador, llamado *WayPointer* para marcar exactamente esas coordenadas, correspondientes a la posición real de la compañía. En el fragmento de código 1.6 se muestra esta implementación y uso de *JXMapKit*.

```
JXMapKit jXMapKit = new JXMapKit();
// Configure maps provider
jXMapKit.setDefaultProvider(org.jdesktop.swingx.JXMapKit.DefaultProviders.OpenStreetMaps);
jXMapKit.setAutoscrolls(true);
jXMapKit.setZoomButtonsVisible(false);

......

// Get the coordinates from server and set the wayPointer in the map
Coordinates coor;
coor = GeoCoder.getGeoCoordinates(company.getAddress());
```

```
double latitude = Double.parseDouble(coor.getLatitude());
double longitude = Double.parseDouble(coor.getLongitude());
position = new GeoPosition(latitude, longitude);

jXMapKit.setAddressLocation(position);
Set<Waypoint> waypoints = new HashSet<Waypoint>();
waypoints.add(new Waypoint(latitude, longitude));

WaypointPainter<?> painter = new WaypointPainter();
painter.setWaypoints(waypoints);

jXMapKit.getMainMap().setOverlayPainter(painter);
jXMapKit.getMainMap().setZoom(2);
jXMapKit.setAddressLocationShown(true);
```

Listado 1.6: Fragmento de código para mostrar mapas geoposicionados

1.3.2.1.3 Pruebas

1.3.3 Iteración 5

Siguiendo los casos de uso del grupo funcional **F2:** *Gestión de decisiones* (ver Figura 1.10) y del grupo funcional **F4:** *Gestión de notificaciones* (ver Figura 1.14), se abordan las siguientes tareas en esta iteración:

- Análisis de los casos de uso.
- Diseño de la funcionalidad relativa a la gestión de decisiones y alertas (o notificaciones) del sistema.
- Implementación de dichas funcionalidades.
- Diseño e implementación de pruebas relativas a la gestión de decisiones y alertas.

Es esta iteración se diseñan e implementan ambos grupos funcionales porque éstos están estrechamente relacionados, ya que en la gestión de decisiones se incluye la creación de notificaciones automáticas.

1.3.3.1 Grupo funcional F2: *Gestión de decisiones*

1.3.3.1.1 Análisis de casos de uso

Como en iteraciones anteriores, se comienza analizando los casos de uso que componen este grupo funcional, considerado uno de los más importantes, ya que permitirá la gestión de decisiones del sistema.

Modificar decisión

Este es un caso de uso abstracto del que heredan otros tres casos específicos, según el tipo de decisión que se desee modificar (ver Figura 1.10). Se ha utilizado la herencia porque los tres casos de uso comparten características en común, como es la introducción de los datos del título y la descripción de la decisión a modificar por parte del usuario.

En la Tabla 1.10 se describe el caso de uso *Modificar Decisión - Modificar Propuesta*.

En la Figura 1.44 se muestra el diagrama de clases de análisis para el subsistema cliente. En la Figura 1.45 se muestra el diagrama de clases de análisis para el subsistema servidor.

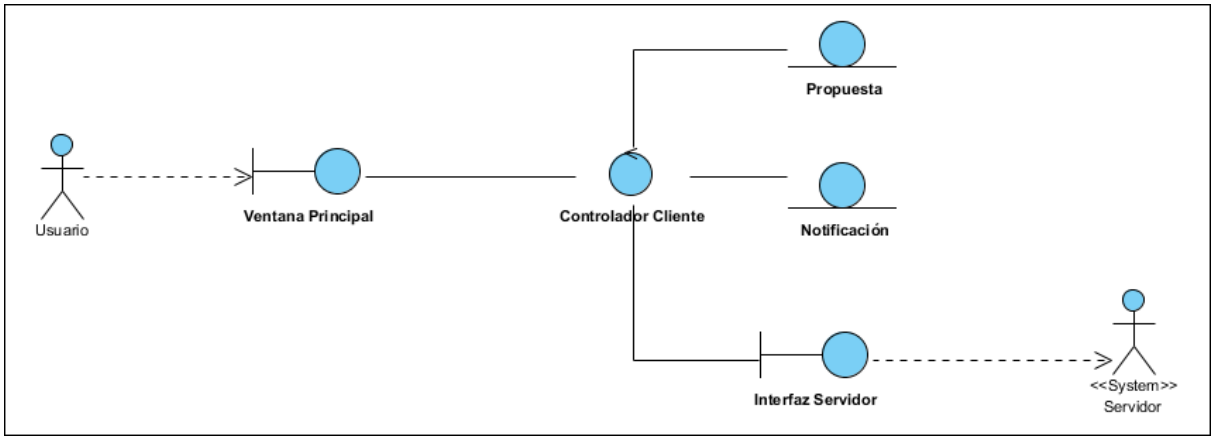

Figura 1.44: Diagrama de clases de análisis - Cliente - Modificar Propuesta

Los otros casos de uso, *Crear decisión* y *Eliminar decisión*, se especifican de una manera muy similar a la mostrada para este caso de uso.

Nombre: Modificar Decisión - Modificar Propuesta (*Proposal*)
Descripción: Funcionalidad para modificar una nueva Propuesta en un proyecto.
Precondiciones: Que el usuario haya accedido al sistema, tenga permisos para realizar la operación y sea autor de la Propuesta.
Post-condiciones: Se modifica la Propuesta.
Flujo principal: 1. El usuario selecciona una Propuesta de un Tema. 2. El usuario introduce el nuevo título y descripción de la Propuesta. 3. El usuario introduce otros datos específicos para este tipo de decisión. 4. El sistema valida los datos introducidos. 5. Se modifica la Propuesta para el Tema seleccionado. 6. El sistema crea una notificación para los usuarios dados de alta en el proyecto. 7. Se refrescan y actualizan las decisiones, para reflejar visualmente el cambio.
Flujo alternativo 1: datos incompletos: 1. El usuario selecciona una Propuesta de un Tema. 2. El usuario introduce el nuevo título y descripción de la Propuesta. 3. El usuario introduce otros datos específicos para este tipo de decisión. 4. El sistema valida los datos introducidos. 5. Los datos son incompletos. Se muestra un mensaje y se vuelven a solicitar los datos de la Propuesta.
Flujo alternativo 2: Propuesta ya existente: 1. El usuario selecciona una Propuesta de un Tema. 2. El usuario introduce el nuevo título y descripción de la Propuesta. 3. El usuario introduce otros datos específicos para este tipo de decisión. 4. El sistema valida los datos introducidos. 5. Ya existe otra Propuesta con ese título en ese tema. Se muestra un mensaje y se vuelven a solicitar los datos.

Tabla 1.10: Especificación del caso de uso *Modificar decisión - Modificar Propuesta*

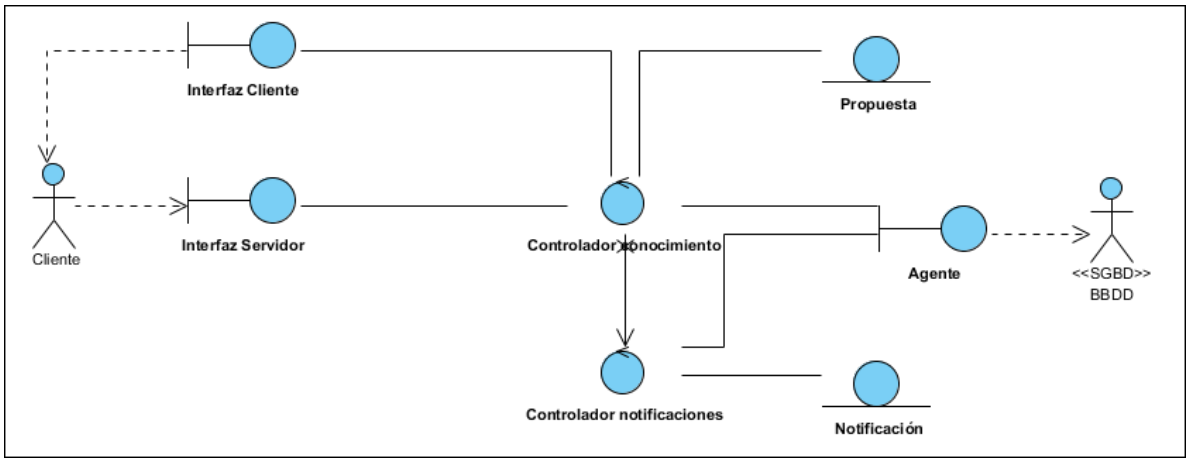

Figura 1.45: Diagrama de clases de análisis - Servidor - Modificar Propuesta

Aceptar o rechazar decisión

Este caso de uso se ha modelado como una extensión al caso de uso anterior, *Modificar Decisión* (ver Figura 1.10). Esto es así porque, aunque también es una modificación de una determinada decisión, existen algunas diferencias en el comportamiento con respecto al caso de uso anterior, siguiendo un comportamiento específico. Dichas diferencias consisten en que este caso de uso (sólo disponible para el rol de jefe de proyecto) es independiente del tipo de decisión, y el jefe de proyecto no tiene porqué ser necesariamente el autor de la decisión.

En la Tabla 1.11 se describe el caso de uso *Aceptar o rechazar decisión*.

En este caso, el diagrama de clases de análisis en ambos subsistemas son prácticamente iguales a los diagramas del caso de uso anterior.

1.3.3.1.2 Diseño e implementación

En las Figuras 1.46 se muestra los diagramas de secuencia para el caso de uso *Crear decisión - Crear Propuesta* en el subsistema cliente, mientras que la Figura 1.47 refleja el diagrama de secuencia de ese caso de uso para el servidor. Señalar que en el caso del subsistema cliente, aunque sólo se muestre un diagrama, se han creado tres diagramas de secuencia para representar los diferentes actores (o roles de usuarios del sistema) que participan en el caso de uso, según el tipo de decisión. En el servidor, sin embargo, no es necesario separarlos, ya que la secuencia de acciones a realizar es la misma, independientemente del tipo de decisión creada.

Nombre: Aceptar o rechazar decisión
Descripción: Funcionalidad para poder cambiar el estado de una decisión, aceptándola o rechazándola, reflejando este cambio visualmente.
Precondiciones: Que el usuario haya accedido al sistema y tenga permisos para realizar la operación.
Post-condiciones: Se cambia el estado de una decisión, a aceptada o rechazada.
Flujo principal: 1. El jefe de proyecto selecciona una decisión de las visualizadas. 2. El jefe de proyecto selecciona un estado (Aceptada/Rechazada) para dicha decisión. 3. El sistema valida los datos. 4. Se modifica la decisión. 5. El sistema crea una notificación para los usuarios dados de alta en el proyecto. 6. Se refrescan y actualizan las decisiones, para reflejar visualmente el cambio.

Tabla 1.11: Especificación del caso de uso *Aceptar o rechazar decisión*

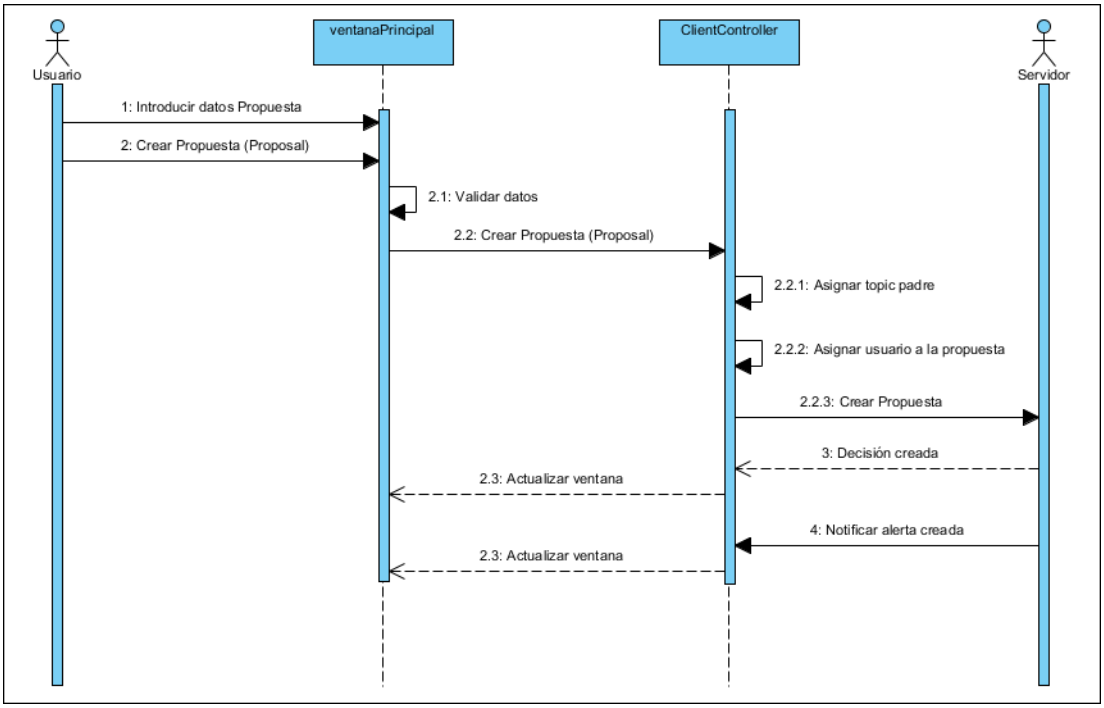

Figura 1.46: Diagrama de secuencia - Cliente - Crear decisión (*Topic*)

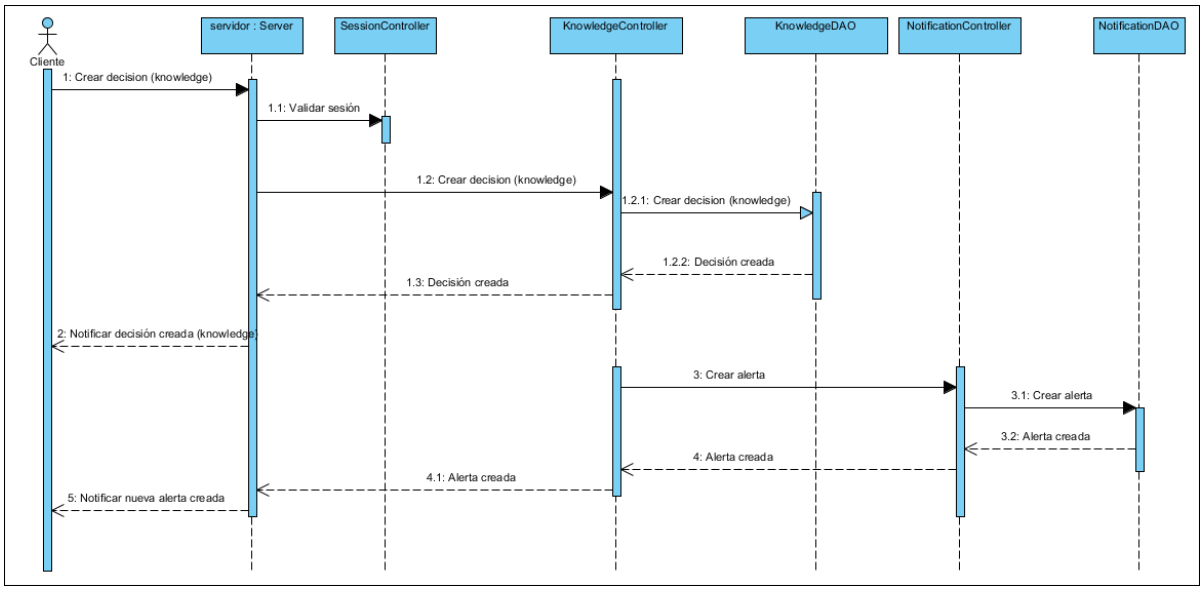

Figura 1.47: Diagrama de secuencia - Servidor - Crear decisión

Del mismo modo y de manera muy similar a estos diagramas de secuencia anteriores, se modela el funcionamiento del resto de casos de uso englobados en este grupo funcional, pasando al diseño e implementación de dicha funcionalidad, con algunos aspectos a destacar en ella.

Servidor

En el diseño e implementación de estos casos de uso, cabe destacar la creación de una alerta o notificación de manera automática por parte del servidor al realizar cualquier acción sobre las decisiones. Dicha alerta se crea para el proyecto al cuál pertenece esa decisión, y para todos los usuarios que en dicho proyecto participan. De este modo, se notifica a los empleados que trabajan en ese proyecto qué nuevo conocimiento está disponible, indicando en la alerta el tipo de decisión afectada, su autor, fecha y otros detalles. Esto facilita la comunicación asíncrona, ya que cuando un usuario vuelva a iniciar sesión, podrá comprobar sus nuevas alertas y adquirir conciencia de los cambios producidos.

En la Figura 1.48 se muestra el diagrama de clases para la funcionalidad de gestión de decisiones, mostrando las asociaciones entre clases y entre los controladores de decisiones y de notificaciones. Como en el resto de casos, las operaciones de bases de datos se delegan en el gestor de bases de datos y éste, a su vez, delega en el framework **Hibernate**.

Este es otro de los puntos principales del sistema desarrollado, ya que, gracias a esta funcionalidad, se permite gestionar todas las decisiones de los proyectos, se permiten añadir archivos adjuntos y se lleva a cabo el sistema de alertas.

También cabe destacar otra decisión de diseño que se ha tenido en cuenta para implementar otro de los requisitos del sistema, que es la notificación de información de manera síncrona. Para ello, cuando un cliente crea, modifica o elimina una decisión y envía la petición al servidor, éste, además de crear la alerta, notifica a los clientes conectados al servidor que se ha producido un cambio, para que éstos puedan actualizar su vista de la interfaz gráfica y puedan reflejar los cambios sobre esa decisión en tiempo real. Para ello, el servidor lanza un hilo por cada uno de los clientes registrados en el sistema y les envía la información necesaria. Se utilizan hilos para no bloquear el servidor mientras manda actualizaciones a los clientes y pueda seguir atendiendo otras peticiones.

La clase controlador encargada de realizar esta tarea constituye además un patrón **Observador**, el cuál se utiliza para registrar los clientes autenticados en el sistema y notificar y actualizar su estado. En el diagrama de la Figura 1.49 se muestra este patrón y las clases que lo forman.

En el fragmento de código 1.7 se muestra parte de la implementación de la clase controlador de los clientes, y en el fragmento 1.8 como se ha implementado la gestión de hilos para notificar a los clientes. La clase encargada de esto último además representa un patrón **proxy**, ya que los clientes son remotos.

```
public class ClientsController {

   private static Hashtable<Long, IClient> clients = new Hashtable<Long, IClient>();

   public static void attach(long sessionID, IClient client) {
      clients.put(sessionID, client);
   }

   public static void detach(long sessionID) {
      clients.remove(sessionID);
   }

    public static void notifyKnowledgeAdded(long sessionId, Knowledge k, Knowledge parentK)
         throws RemoteException {
      // Notify the clients (except the client that launched the operation) about the
          operation, in order to refresh their view
```

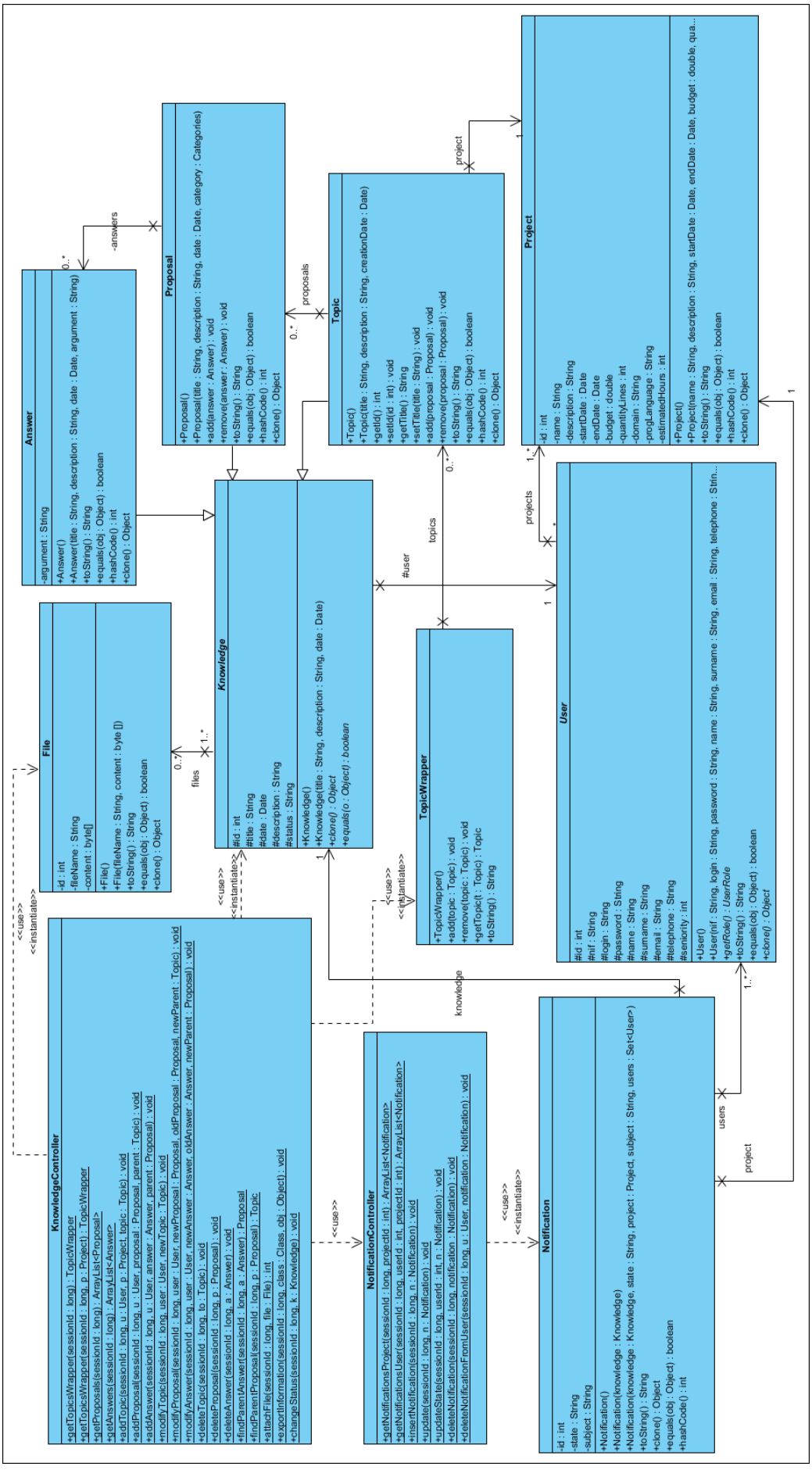

Figura 1.48: Diagrama de clases - Gestión de decisiones

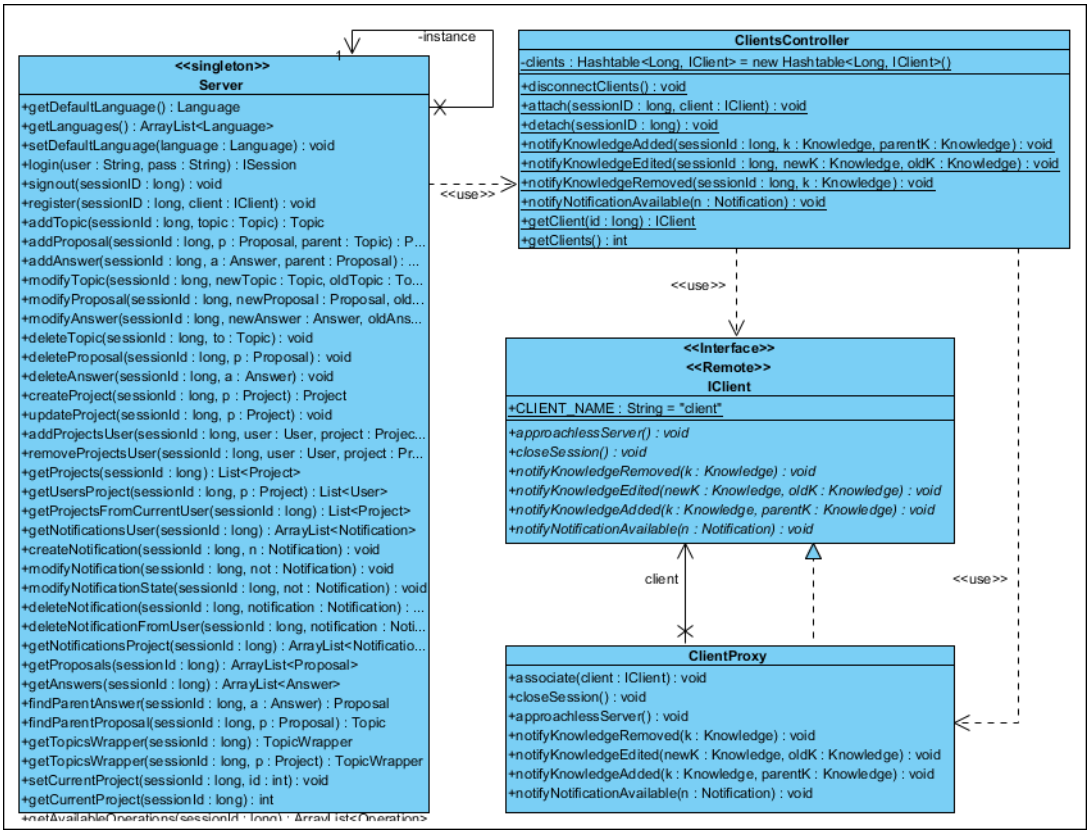

Figura 1.49: Diagrama de clases - Observador para actualizar clientes conectados

```
      for(Long id : clients.keySet())
         if (id != sessionId)
            clients.get(id).notifyKnowledgeAdded(k, parentK);
   }

   ....
}
```

Listado 1.7: Fragmento de código del controlador de clientes

```
public class ClientProxy implements IClient {

    ....

   @Override
   public void notifyKnowledgeAdded(Knowledge k, Knowledge parentK) throws RemoteException
       {
      Thread thread;

      // Launch the operation on another thread for not stopping the server
      thread = new Thread(new notifyKnowledgeAddedThread(client, k, parentK));
      thread.start();
   }

   ....

   private class notifyKnowledgeAddedThread implements Runnable {

      private IClient client;
      private Knowledge k;
      private Knowledge parentK;

      public notifyKnowledgeAddedThread(IClient client, Knowledge k, Knowledge parentK) {
         this.client = client;
         this.k = k;
         this.parentK = parentK;
      }

      public void run() {
         try {
            client.notifyKnowledgeAdded(k, parentK);
         } catch(Exception e) {
         }
      }
   }

   ....
```

```
}
```

Listado 1.8: Soporte multi-hilo para actualizar el estado de clientes

Cliente

En el subsistema cliente cabe destacar la utilización del API de Java **Reflection** para poder configurar el diálogo con el fin de gestionar las decisiones (creación y modificación) en tiempo de ejecución. Así, cuando el usuario selecciona una decisión a crear o modificar (*Tema*, *Propuesta* o *Respuesta*), la interfaz se adaptará a ese tipo de decisión, mostrando los elementos oportunos. Por tanto, se utiliza la reflexión de Java para instanciar el panel gráfico correspondiente y visualizarlo cuando el usuario haya elegido una acción, en tiempo de ejecución.

En el fragmento de código 1.9 se presenta un ejemplo de uso de la reflexión para crear un componente visual conocido en tiempo de ejecución, según el valor de la variable *subgroup*.

```
Constructor c = Class.forName("presentation.panelsManageKnowledge.JPManage"+subgroup).
    getConstructor(new Class [] {JFMain.class, JDialog.class, Object.class, String.class});
component = (Component) c.newInstance(new Object [] {parentFrame, dialog, data,
    operationToDo});
```

Listado 1.9: Fragmento de código utilizando *reflection*

Por otra parte, en el cliente, cuando el servidor le notifica que se ha producido un cambio en las decisiones, producido por otro cliente, se refrescan las decisiones en la vista de visualización de decisiones (comentada en la iteración anterior), para poder reflejar este cambio de manera síncrona, refrescando el grafo y árbol de decisiones, así como la información asociada a cada una de ellas.

Para terminar, cabe destacar que las operaciones de gestión de decisiones (creación, modificación, etc) pueden bien realizarse desde un menú o bien desde los nodos del grafo, gracias a un menú contextual que aparece al interactuar con el ratón. Se ha decidido diseñarlo de esta manera para permitir a los usuarios una flexibilidad y libertad a la hora de realizar las acciones, pudiendo seleccionar aquella que le sea más cómoda. En otras funcionalidades, la interfaz también proporciona estas opciones de realizar las acciones desde diferentes puntos.

1.3.3.2 Grupo funcional F4: *Gestión de notificaciones*

1.3.3.2.1 Diseño e implementación

Como en iteraciones anteriores y de modo similar, se modelan los diagramas de secuencia para los casos de uso de este grupo funcional y se comienza con su diseño e implementación.

Servidor

Como se ha detallado en el apartado 1.3.3.1.2, se crean alertas (o notificaciones) de manera automática para todos los usuarios del proyecto que ha sufrido cambios en sus decisiones, como se muestra en el diagrama de la Figura 1.48. Por tanto, la misma alerta debe crearse para todos los usuarios de ese proyecto, pero, para evitar que la información de esa alerta esté repetida, la base de datos se ha diseñado de tal modo que la alerta sólo se crea una vez y se hace referencia a ella para todos los usuarios, utilizando la tabla *notificationsUsers* con claves ajenas, como se muestra en el diseño de base de datos de la Figura 1.27.

De este modo, cada usuario podrá editar y eliminar su propia alerta, sólo eliminando la alerta original cuando ningún usuario tenga ya referencias a ella, es decir, cuando todos los usuarios del proyecto hayan borrado esa alerta. Esta tarea se ha delegado al SGBD de MySQL, mediante la creación de un *trigger*, mostrado en el fragmento de código 1.10. Dicho trigger será el encargado de borrar la alerta original cuando ya no existan referencias a ella por parte de ningún usuario.

```
CREATE TRIGGER DeleteEntity
AFTER DELETE ON notificationsUsers
FOR EACH ROW
BEGIN
DECLARE
  cont INT;
  SELECT COUNT(*) INTO cont FROM notificationsUsers WHERE idNotification = OLD.
      idNotification;
  IF (cont = 0) THEN
    DELETE FROM notifications WHERE id = OLD.idNotification;
  END IF;
END$$
```

Listado 1.10: Trigger de base de datos para gestionar la eliminación de alertas

Cliente

Siguiendo los diagramas de secuencia para este subsistema, se diseña e implementa la interfaz gráfica de usuario para dar soporte a cada caso de uso, enviando peticiones al servidor y obteniendo los datos que éste devuelve, mostrándolos en la interfaz. En este caso, se ha diseñado e implementado una vista para poder mostrar esas alertas, de manera similar a una vista de correo electrónico, mostrando las alertas leídas y no leídas, la información de dichas alertas, su autor, etc.

1.3.3.2.2 Pruebas

1.3.4 Iteración 6

Siguiendo los casos de uso del grupo funcional **F5:** *Gestión de proyectos* (ver Figura 1.16), se abordan las siguientes tareas en esta iteración:

- Análisis de los casos de uso.
- Diseño de la funcionalidad relativa a la gestión de proyectos.
- Implementación de dicha funcionalidad.
- Diseño e implementación de pruebas relativas a la gestión de proyectos.

1.3.4.1 Grupo Funcional F5: *Gestión de proyectos*

1.3.4.1.1 Análisis de casos de uso

Esta iteración comienza con el análisis de los casos de uso que componen este grupo funcional, resaltando el caso de uso *Aconsejar decisiones*, una de las funcionalidades destacadas del sistema.

1.3.4.1.2 Diseño e implementación

Utilizando diagramas de secuencia de diseño, se modela el funcionamiento de los casos de uso que componen este grupo funcional. Para el caso de uso *Crear Proyecto*, su diagrama de

secuencia para el subsistema cliente se muestra en la Figura 1.50, mientras que en la Figura 1.51 se muestra el diagrama de secuencia para el servidor. En este caso, para crear un proyecto, es necesario también conocer los empleados de la compañía, para poder asignarle aquellos que deben trabajar en ese proyecto.

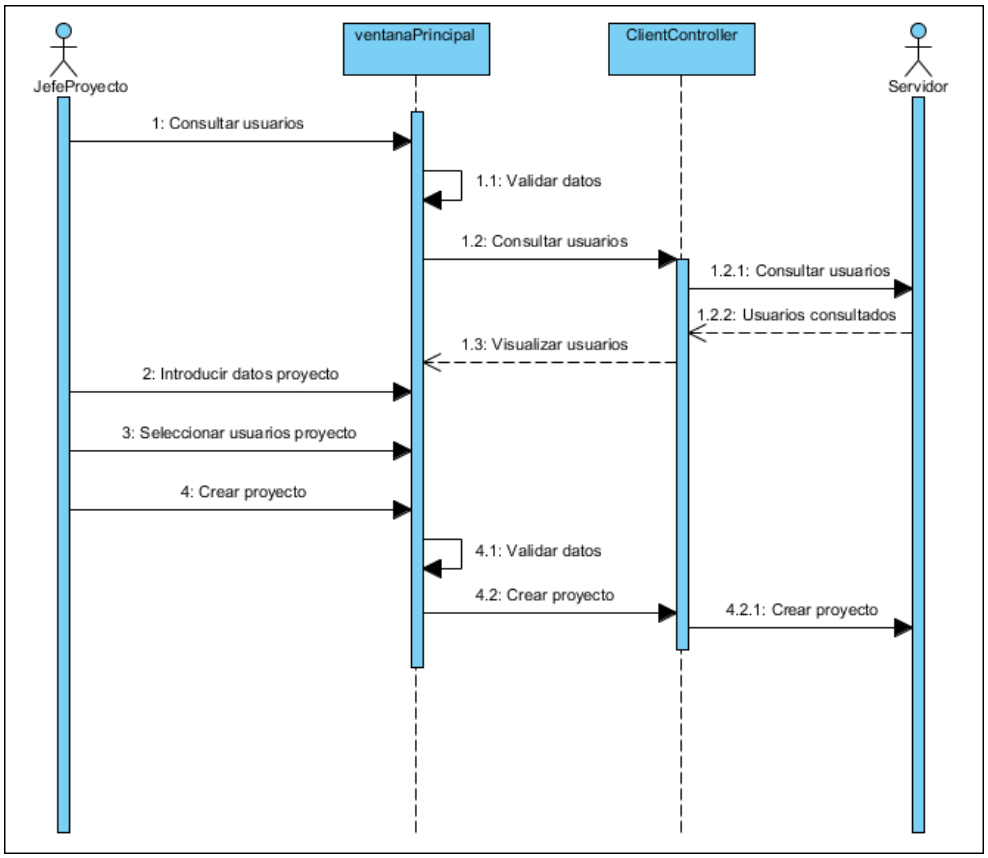

Figura 1.50: Diagrama de secuencia - Cliente - Crear proyecto

Como se ha mencionado anteriormente, es importante destacar el caso de uso *Aconsejar Decisiones*, ya que éste es uno de los requisitos más importantes que el sistema debe cunplir. Así, en la Figura 1.52 se muestra el diagrama de secuencia para este caso de uso en el cliente, mientras que en la Figura 1.53 se observa el diagrama de secuencia para el servidor.

Hay que señalar que esta funcionalidad incluye el comportamiento de los casos de uso *Consultar decisiones* y *Consultar proyectos*, por lo que dichos comportamientos se muestran de manera simplificada en estos diagramas de secuencia.

En los siguientes apartados se detallan aspectos de diseño e implementación tenidos en cuenta en este grupo funcional.

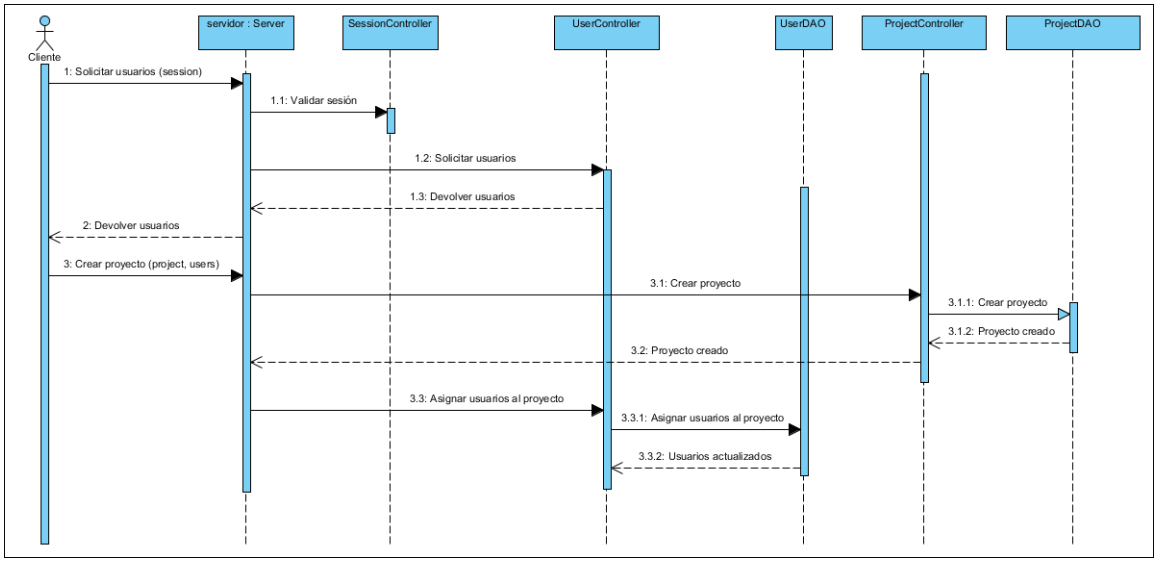

Figura 1.51: Diagrama de secuencia - Servidor - Crear proyecto

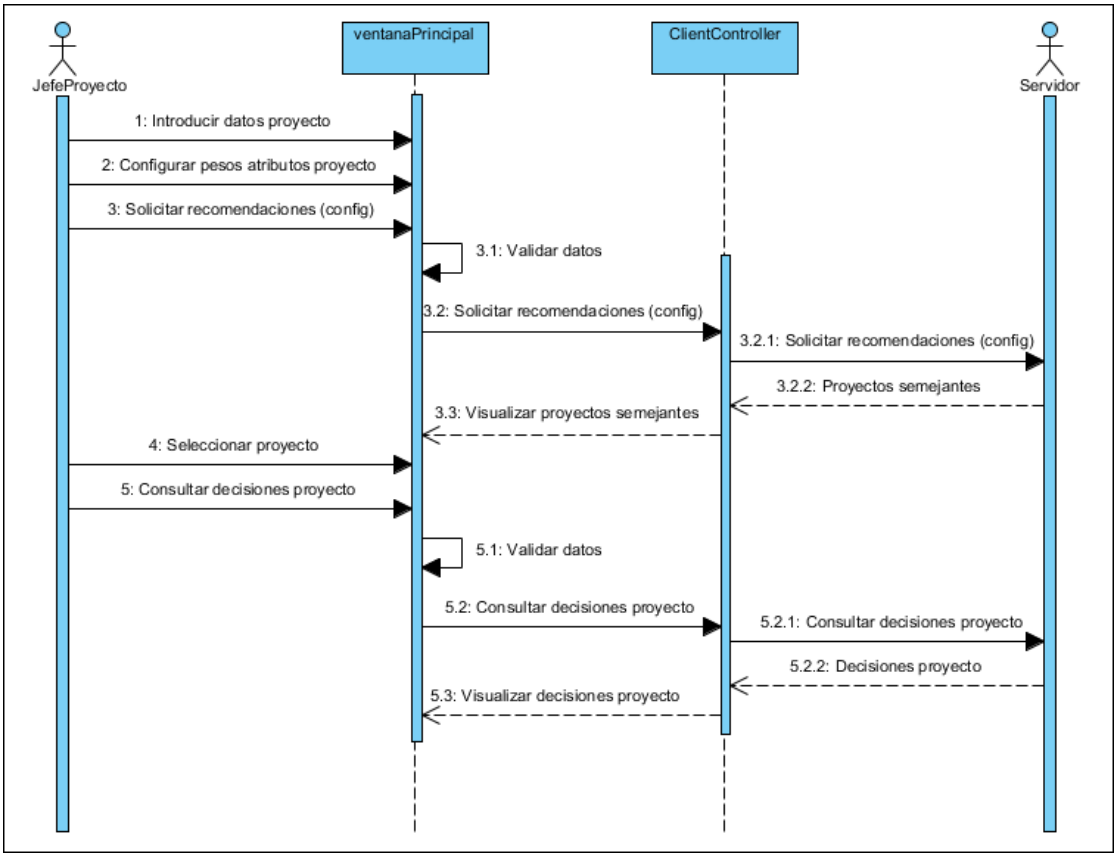

Figura 1.52: Diagrama de secuencia - Cliente - Aconsejar decisiones

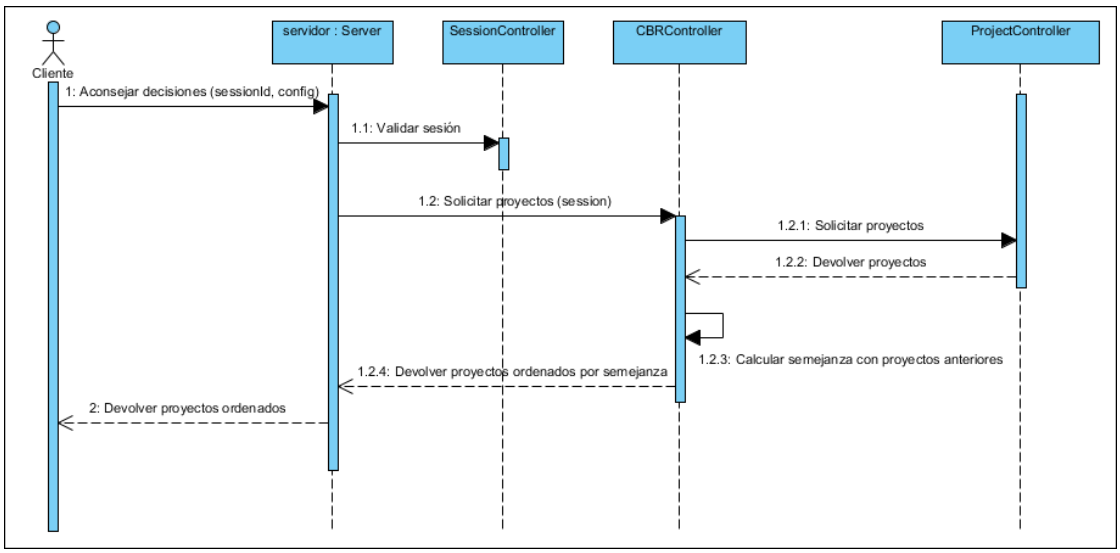

Figura 1.53: Diagrama de secuencia - Servidor - Aconsejar decisiones

Servidor

Como se comentó en el modelo de negocio en el apartado **??**, cada usuario trabaja en 1 o más proyectos, y en cada proyecto participan varios usuarios, por lo que, como se refleja en los diagramas de secuencia anteriores, para crear y modificar un proyecto hay que asignar los usuarios que en dicho proyecto trabajan. En el diagrama de la Figura 1.54 se observan las clases que participan en la gestión de proyectos y las asociaciones que existen entre los usuarios y los proyectos, y sus clases controladoras correspondientes.

Sin embargo, en este grupo funcional, lo que cabe destacar es el diseño e implementación del caso de uso de *Aconsejar decisiones*. Este caso de uso representa el requisito de recuperar proyectos similares a uno dado, para poder mostrar las decisiones que en él se tomaron, cuáles fueron aceptadas o rechazadas, etc, para poder tenerlas en cuenta en un nuevo proyecto. De este modo, a partir de un nuevo proyecto, y basándose en proyectos anteriores ya terminados, se recuperan aquellos que sean más similares al proyecto dado y se muestran sus decisiones. Para ello, se utiliza el **Razonamiento Basado en Casos**, o CBR, detallado en la sección **??**.

Cada caso del CBR representará un proyecto, definido por un conjunto de atributos, que serán los utilizados para poder comparar proyectos y calcular la semejanza entre casos.

En lo que respecta al CBR en el subsistema servidor, éste recibe la configuración definida en el cliente, el tipo de algoritmo a utilizar, el proyecto a evaluar y el número de proyectos (o casos) que se desean recuperar y visualizar. Por otro lado, se consultan todos los proyectos

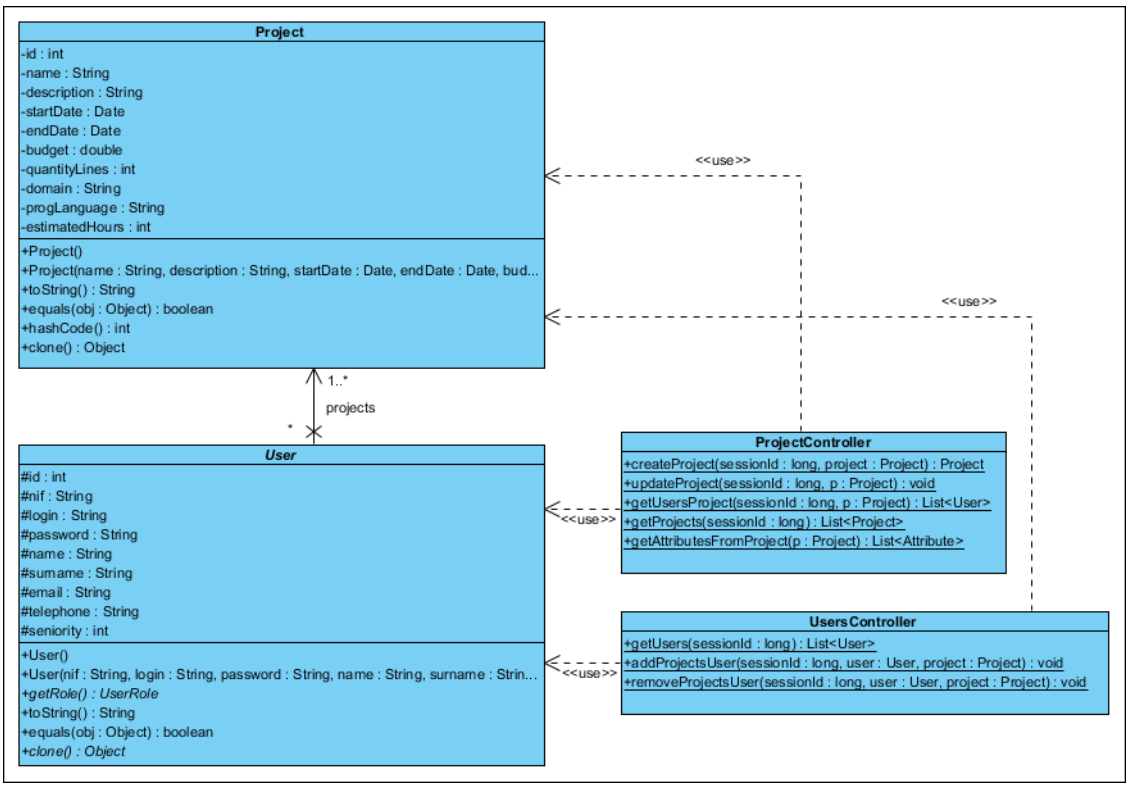

Figura 1.54: Diagrama de secuencia - Servidor - Controlador proyectos

del sistema y se almacenan en una lista aquellos que ya han finalizado.

A continuación, se explican cada uno de estos elementos involucrados en esta funcionalidad de CBR.

- Número de proyectos a recuperar, definidos por una variable *k*. Es el número de proyectos similares que se recuperan en la fase de *Recuperación* del CBR. Pueden recuperarse todos los proyectos similares, o sólo algunos, según el valor de *k*.
- El proyecto a evaluar es el nuevo proyecto, o caso, del que se desean obtener sus decisiones, basándose en proyectos ya pasados y similares a éste.
- La lista de proyectos ya terminados, que componen lo que se llama la *base de casos* en CBR. Son todos los proyectos pasados con los que se calculará la semejanza o similitud con el caso nuevo a evaluar.
- El algoritmo a utilizar para calcular la semejanza entre el nuevo caso y cada uno de los casos pasados.
- La configuración necesaria para utilizar en los algoritmos de CBR.

Cabe destacar este último elemento, que es la configuración utilizada en los algoritmos del CBR. Dicha configuración contiene el conjunto de pesos que los atributos van a tomar al calcular el valor final de la semejanza entre casos. Además contiene, para cada atributo, una función que indica como valorar y comparar ese atributo con los atributos de los diferentes casos, llamada *función de semejanza local*. Dicha función establece como comparar los atributos entre dos casos, devolviendo el valor de semejanza entre esos atributos. Existen tres tipos de esta función:

- **Diferencia**: esta función devuelve la diferencia entre dos valores numéricos o fechas. Si los valores son de tipo cadena, devuelve 1 si las cadenas son iguales, o 0 en caso contrario.
- **Igualdad**: función que devuelve 1 si los valores son iguales, y en caso contrario.
- **Intervalo**: función utilizada para evaluar si la diferencia de dos valores numéricos se encuentran en un determinado margen, devolviendo la desviación con respecto a ese margen.

En lo que respecta a los algoritmos utilizados para calcular el valor de la semejanza final, o *global*, entre casos, se han diseñado e implementado diferentes algoritmos, basándose en la literatura estudiada y comentada en la sección **??**. Dichos algoritmos son:

- **Nearest neighbor**: este algoritmo, conocido también como *NN Method*, calcula el valor de semejanza global entre dos casos realizando la media aritmética de los valores de los atributos. Así, se calcula el sumatorio del producto del valor de cada atributo (calculado por la función de semejanza local) por su peso, y se divide por la suma del peso total de todos los atributos. En la ecuación 1.1 se muestra esta función de semejanza global utilizada en este algoritmo, siendo *c1* y *c2* los casos a evaluar; w_i el peso de cada atributo, y $sem(att1_i, att2_i)$ el valor de semejanza calculado por la función de semejanza local entre los atributos de ambos casos.
- **Euclidean Distance**; este algoritmo calcula el valor de semejanza global entre dos casos realizando la distancia euclídea de los valores de los atributos. Así, se calcula el sumatorio del producto del valor de cada atributo (calculado por la función de semejanza local) elevado al cuadrado, por su peso. En la ecuación 1.2 se muestra esta función

de semejanza global utilizada en este algoritmo, siendo *c1* y *c2* los casos a evaluar; w_i el peso de cada atributo, y $sem(att1_i, att2_i)$ el valor de semejanza calculado por la función de semejanza local entre los atributos de ambos casos.

$$sem(c1, c2) = \sum_{i=1}^{n}(w_i * sem(att1_i, att2_i))/\sum_{i=1}^{n} w_i \tag{1.1}$$

$$sem(c1, c2) = \sum_{i=1}^{n}(w_i * sem(att1_i - att2_i)^2) \tag{1.2}$$

Una vez calculada la semejanza global entre el caso a evaluar (el nuevo proyecto) y cada uno de los proyectos pasados, obteniendo una lista de proyectos con su valor de semejanza, dicha lista se ordena de mayor a menor y se toman los *k* primeros, en caso de haber establecido el valor de *k*. Con ello, termina esta fase del CBR, que es la fase de **Recuperación**, de la que se encarga el sistema servidor. Acto seguido, esta lista de proyectos, ordenados de mayor a menor semejanza con el nuevo caso a evaluar, se devuelve al cliente.

En la Figura 1.55 se muestra el diagrama de clases de diseño de la funcionalidad del CBR. Como se puede observar, se han diseñado interfaces para implementar las funciones de semejanza local y global. De este modo, se podrían extender estas funciones, añadiendo nuevos métodos de comparación, simplemente implementando estas interfaces, consiguiendo que el controlador de CBR sea extensible a nuevas funciones de semejanza.

Para terminar, en el fragmento de código 1.11 se muestra la implementación del algoritmo *NN*, haciendo uso del diseño y elementos comentados anteriormente. Un aspecto a señalar es el uso del API de relexión de Java, utilizado para acceder a los atributos de un proyecto en tiempo de ejecución.

```
/**
* Class used to apply the Nearest Neighbor algorithm
*/
public class NNMethod {

   /* Apply the algorithm over the cases.
    *  Return the similarity cases */
   public static List<Project> evaluateSimilarity(Project caseToEval, List<Project> cases,
       ConfigCBR config, int k) throws Exception
   {
```

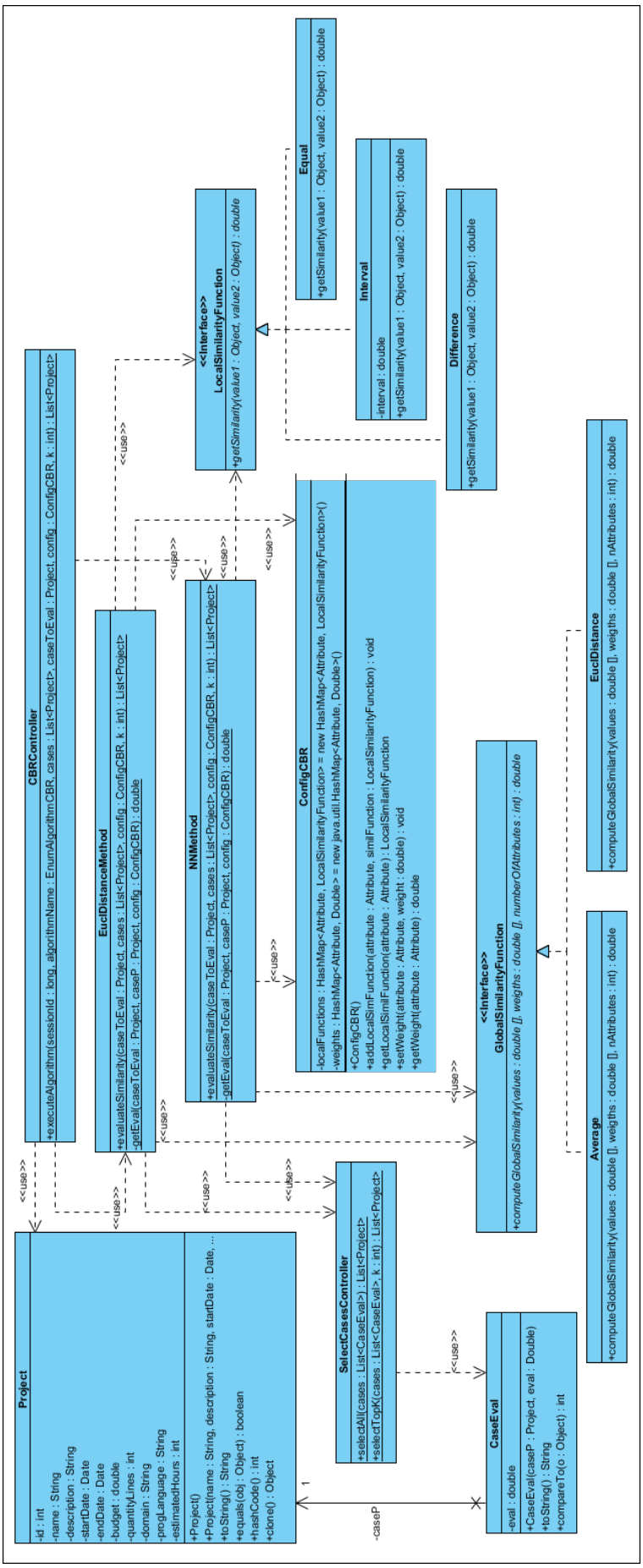

Figura 1.55: Diagrama de clases - Servidor - Razonamiento Basado en Casos

```
    List<Project> similCases = new ArrayList<Project>();
    List<CaseEval> result = new ArrayList<CaseEval>();
    for(Project caseP: cases)
    {
       result.add(new CaseEval(caseP, getEval(caseToEval, caseP, config)));
    }
    // Sort the result
    Collections.sort(result);
    if (k == 0)
       similCases = SelectCasesController.selectAll(result);
    else
       similCases = SelectCasesController.selectTopK(result, k);

    return similCases;
}

 /*
  * Get the evaluation for each attribute of the cases to compare
  */
 private static double getEval(Project caseToEval, Project caseP, ConfigCBR config)
     throws Exception {
   LocalSimilarityFunction lsf = null;
   GlobalSimilarityFunction gsf = null;

   // Take attributes from each case
   List<Attribute> attributesCaseToEval = ProjectController.getAttributesFromProject(
       caseToEval);
   List<Attribute> attributesCase = ProjectController.getAttributesFromProject(caseP);
   // Global similarity function used in NN Method
   gsf = new Average();

   // Evaluation for each attribute (ignore id and serialVersionUID)
   double[] values = new double[attributesCaseToEval.size() - 2];
   // Weights for each attribute (ignore id and serialVersionUID)
   double[] weights = new double[attributesCaseToEval.size() - 2];

   int nAttributes = 0;
   for(int i=2; i<attributesCaseToEval.size(); i++)
   {
      Attribute attCase1 = attributesCaseToEval.get(i);
      Attribute attCase2 = attributesCase.get(i);

      // Evaluation of the attributes using local similarity function
      if ((lsf = config.getLocalSimilFunction(attCase1)) != null) {
         // Using reflection in order to get the attribute value
         Field attField1 = Project.class.getDeclaredField(attCase1.getName());
         Field attField2 = Project.class.getDeclaredField(attCase2.getName());
         attField1.setAccessible(true);
         attField2.setAccessible(true);
```

```
            // Calculate similarity using the local similarity function
            values[i - 2] = lsf.getSimilarity(attField1.get(caseToEval), attField2.get(
                caseP));
            weights[i - 2] = config.getWeight(attCase1);
            nAttributes++;
        }
      }

      // Return the similarity between both cases, applying the global function (average,
          in this algorithm)
      return gsf.computeGlobalSimilarity(values, weights, nAttributes);
    }
}
```

Listado 1.11: Fragmento de código para el algoritmo *NN* del CBR

Cliente

Respecto al subsistema cliente, tiene especial interés el diseño e implementación del caso de uso *Solicitar recomendaciones*. A continuación se comentan algunos detalles tenidos en cuenta a la hora del diseño y la implementación, ya que en el Apéndice **??** se muestra paso a paso el proceso que hay que seguir para poder crear la configuración necesaria para ejecutar el CBR.

En primer, al igual que en el subsistema servidor, se ha utilizado el API de reflexión de Java para poder mostrar en la interfaz gráfica los atributos de un proyecto y poder introducir sus valores, creando el proyecto que se desea comparar y evaluar. A continuación, utilizando también reflexión, se ha diseñado otra ventana donde se pueden configurar los pesos de cada atributo y seleccionar el tipo de función de semejanza local a utilizar para cada uno de ellos. Para introducir los pesos, como deben ser valores numéricos entre 0 y 1, se ha utilizado un *slider*, que permite seleccionar de manera sencilla valores en el rango indicado.

De este modo, con la configuración creada, se envían los datos al servidor y se obtiene su respuesta, mostrando todos los datos de los proyectos semejantes obtenidos. Además, para cada uno de esos proyectos, se consultan sus decisiones e información asociada a éstas y se muestran también en la interfaz. Con ello, se completa la fase de **Reutilización** del CBR.

Así, el jefe de proyecto podría modificar el proyecto inicial (el usado para obtener recomendaciones) con algunos de los datos y decisiones de los proyectos más semejantes, si lo

considera oportuno, cerrando así las fases de *Revisión* y **Almacenamiento** del CBR, ya que ese proyecto actualizado quedaría almacenado en la base de datos.

Para terminar, señalar que gracias a la reflexión de Java, la interfaz gráfica es independiente de los atributos definidos en un proyecto, adaptándose a nuevos atributos que se pudieran añadir a la clase *Proyecto*, sin tener que revisar la implementación de la interfaz. Además, gracias a la estructura de composición de elementos diseñada para la interfaz de usuario, muchos de los paneles y elementos se pueden reutilizar para diferentes casos de uso, como por ejemplo, el panel para introducir los datos de un proyecto, el árbol donde se muestran las decisiones, etc. Esto, como se comentó anteriormente, facilita la reutilización, flexibilidad y extensibilidad del subsistema cliente.

1.3.4.1.3 Pruebas

1.3.5 Iteración 7

Siguiendo los casos de uso del grupo funcional **F6:** *Generación de informes* (ver Figura 1.19) y de grupo funcional **F7:** *Generación de estadísticas* (ver Figura 1.17), se abordan las siguientes tareas en esta iteración:

- Análisis de los casos de uso.
- Diseño de la funcionalidad relativa a la generación de informes y gráficos estadísticos.
- Implementación de dichas funcionalidades.
- Diseño e implementación de pruebas relativas a la generación de informes y gráficos estadísticos.

1.3.5.1 Grupo funcional F6: *Generación de informes*

1.3.5.1.1 Análisis de casos de uso

Como en iteraciones anteriores, se comienza analizando los casos de uso que componen este grupo funcional, realizando las especificaciones de dichos casos de uso y generando sus diagramas de clases de análisis.

1.3.5.1.2 Diseño e implementación

Una vez analizados los casos de uso, se comienza con el diseño, utilizando dicho análisis como base. De este modo, en la Figura 1.56 se muestra el diagrama de secuencia de diseño para el escenario del caso de uso de *Generación de informes PDf* en el cliente, y en la Figura 1.57 se muestra su comportamiento en el servidor.

Hay que señalar que en esta funcionalidad se incluye el comportamiento del caso de uso *Consultar decisiones*, por lo que dicho comportamiento se muestra de manera simplificada en estos diagramas de secuencia.

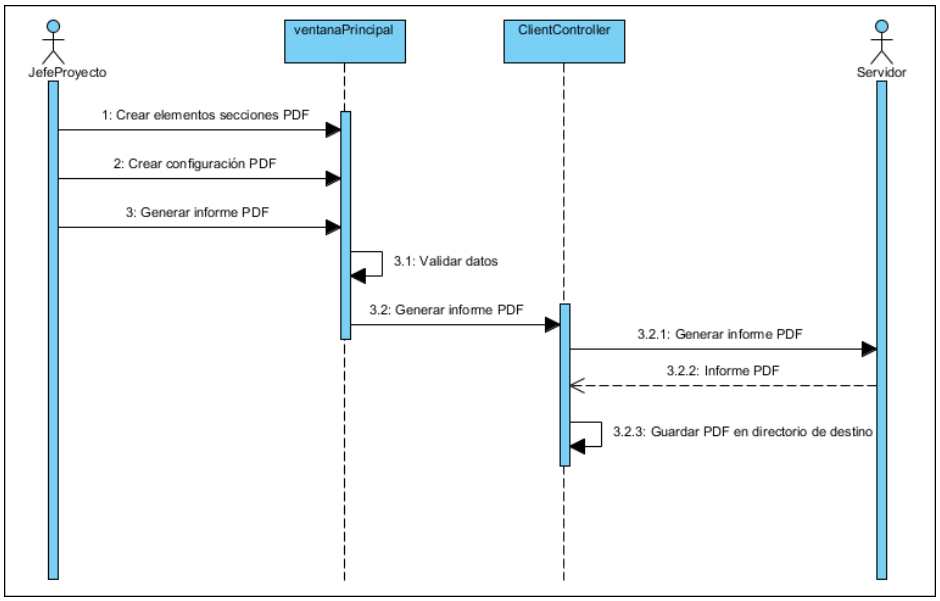

Figura 1.56: Diagrama de secuencia - Cliente - Generar informe

Servidor

Siguiendo los diagramas de secuencia, se realiza el diseño de los casos de uso utilizando un diagrama de clases de diseño y se procede a la implementación de dichos casos de uso, para obtener las clases Java que dan soporte a estas funcionalidades. En la Figura 1.58 se muestra el diagrama de clases de diseño para la funcionalidad de generación de informes en PDF.

Se ha utilizado una jerarquía de herencia para diseñar los elementos que forman parte de las secciones que componen un documento PDF, para poder utilizar la capacidad de polimorfismo de las clases de Java y poder componer una sección del PDF de diferentes elementos, con una superclase de la que heredan. De este modo, una sección se puede componer de:

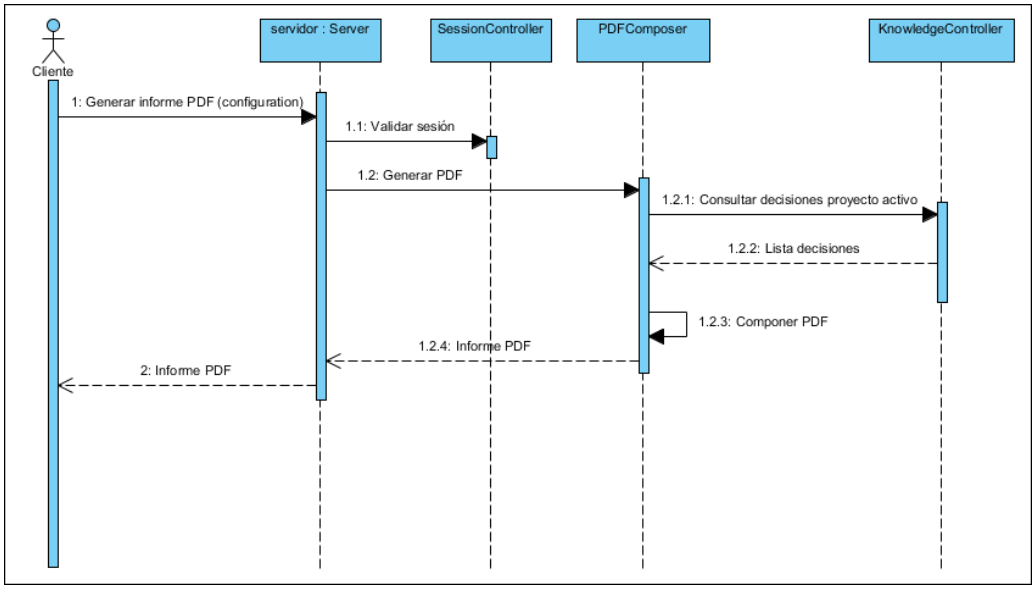

Figura 1.57: Diagrama de secuencia - Cliente - Generar informe

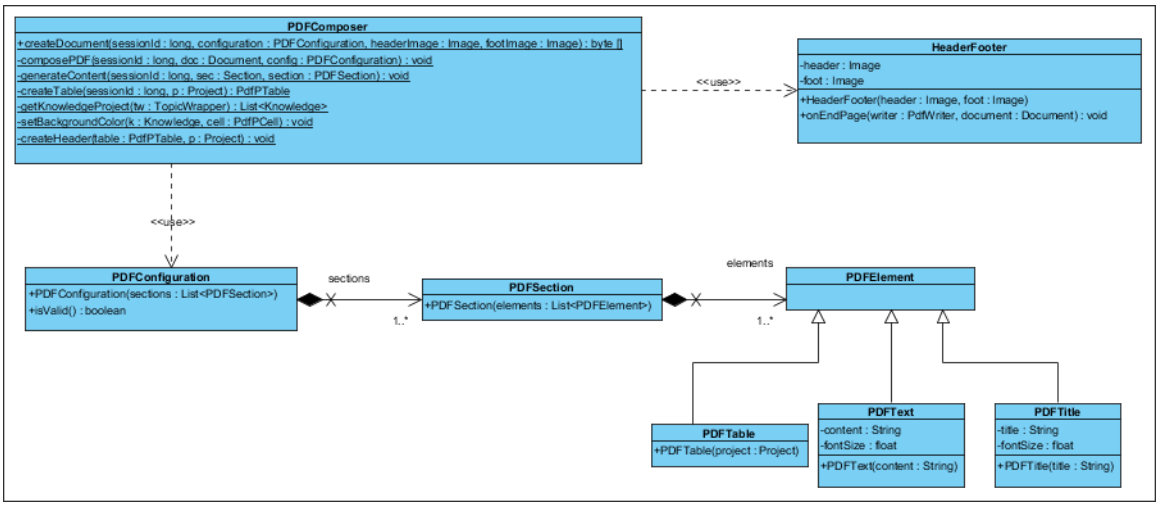

Figura 1.58: Diagrama de clases - Generación de documentos PDF - Servidor

- Un título de sección, que tiene un tipo de fuente determinada.
- Texto, que compone el texto de la sección, también con un tipo de fuente determinada.
- Una tabla, donde se van a mostrar el conjunto de decisiones y toda su información asociada del proyecto seleccionado.

Como se observa en la Figura 1.58, cabe señalar el uso de la clase *HeaderFooter* para colocar una imagen en el encabezado y pie de página de cada página del documento PDF. Esto es útil, por ejemplo, para colocar el logo de una compañía en cada página del informe.

En cuanto a aspectos de implementación, se utiliza la librería **iText** (ver sección **??**) para crear el documento PDF y para ir componiendo sus secciones con los diferentes elementos que las forman, según hayan sido configuradas en el cliente. En el listado de código 1.12 se muestra un fragmento de código de cómo crear el documento y como se insertan en él los elementos.

```
/**
 * Class used to generate a PDF document from the user-entered configuration
 */
public class PDFComposer {

   public static byte[] createDocument(long sessionId, PDFConfiguration configuration,
        Image headerImage, Image footImage) throws NumberFormatException, RemoteException,
        SQLException, NonPermissionRoleException, NotLoggedException, Exception {
      .....

      Document doc = new Document(PageSize.A4, 20, 20, marginTop, marginBottom);

      ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        PdfWriter pdfWriter = PdfWriter.getInstance(doc, buffer);

        // Event used to add header image and foot image
        HeaderFooter event = new HeaderFooter(headerImage, footImage);
      pdfWriter.setPageEvent(event);

      doc.open();
      composePDF(sessionId, doc, configuration);
      doc.close();

      return buffer.toByteArray();
   }
```

```
   private static void composePDF (long sessionId, Document doc, PDFConfiguration config)
        throws NumberFormatException, RemoteException, SQLException,
        NonPermissionRoleException, NotLoggedException, Exception {
      int count = 1;
      // Create the chapter
      Chapter ch = new Chapter(count);
      ch.setNumberDepth(0);
      // Create the different sections
      for (PDFSection section : config.getSections()) {
         Section s = ch.addSection(4f, "");
         generateContent(sessionId, s, section);
         doc.add(ch);
      }
   }

   private static void generateContent(long sessionId, Section sec, PDFSection section)
        throws NumberFormatException, RemoteException, SQLException,
        NonPermissionRoleException, NotLoggedException, Exception {
      for(PDFElement element : section.getElements()){
         if (element instanceof PDFTitle) {
            Font f = FontFactory.getFont(FontFactory.HELVETICA, ((PDFTitle)element).
                getFontSize(), Font.BOLD, new BaseColor(Color.BLACK));
            Paragraph p = new Paragraph(((PDFTitle)element).getTitle().toUpperCase(), f);
            sec.setTitle(p);
         }
         else if (element instanceof PDFText) {
            Font f = FontFactory.getFont(FontFactory.HELVETICA, ((PDFText)element).
                getFontSize(), Font.BOLD, new BaseColor(Color.BLACK));
            Paragraph p = new Paragraph(((PDFText)element).getContent(), f);
            sec.add(p);
         }
         else if (element instanceof PDFTable) {
            PdfPTable table = createTable(sessionId, ((PDFTable)element).getProject());
            sec.add(table);
         }
      }
   }

   ....

}
```

Listado 1.12: Fragmento de código para la generación de documentos PDF

Cliente

Para configurar los elementos que van a formar las secciones del PDF, se han utilizado paneles para representar cada uno de los tres elementos que componen el PDF (tabla, título y texto) y éstos se irán insertando en los paneles que representan las secciones, de tal modo que se pueden ir configurando dichas secciones de una manera visual e intuitiva.

En este aspecto de configuración de las secciones, cabe destacar el uso de la técnica de *Drag & Drop* (*arrastrar y soltar*), que permite la reordenación de los paneles que ya se han insertado en una sección. De este modo, por ejemplo, si en una sección ya se han incorporado los paneles que representan un título, un texto y una tabla, se puede arrastrar y soltar el panel que representa el texto para colocarlo antes que la tabla. Esto es útil para configurar de manera sencilla el orden en que se quiere que aparezcan los diferentes elementos en las secciones que componen el informe PDF.

En la Figura 1.59 se muestra el diagrama de clases para esta funcionalidad en el sistema cliente.

A continuación, se explican las clases involucradas en el proceso de *Drag & Drop*:

- **panelPDFGeneration**: es el panel que representa la vista de la interfaz gráfica para configurar las secciones del PDF con los elementos que en ellas pueden colocarse. Por tanto, se compone de paneles que representan las secciones, y hace uso de la clase *panelPDFElement*, para mostrar los elementos que en dichas secciones pueden insertarse.

- **panelPDFElement**: esta clase representa los elementos que pueden insertarse en las secciones, pero que aún no han sido colocados en el panel que representa una sección.

- **panelPDFDragged**: esta clase representa los elementos del PDF que ya han sido insertados en las secciones, y por lo tanto ya tienen una configuración (texto, fuente, proyecto asignado, etc). Esta clase es la superclase de las clases *panelPDFDraggedTitle*, *panelPDFDraggedText* y *panelPDFDraggedTable*, que representan los tres elementos que se utilizan para componer las secciones del PDF. Esta superclase implementa la interfaz *Transferable*.

- **Transferable**: es una interfaz que deben implementar los objetos sujetos a realizar

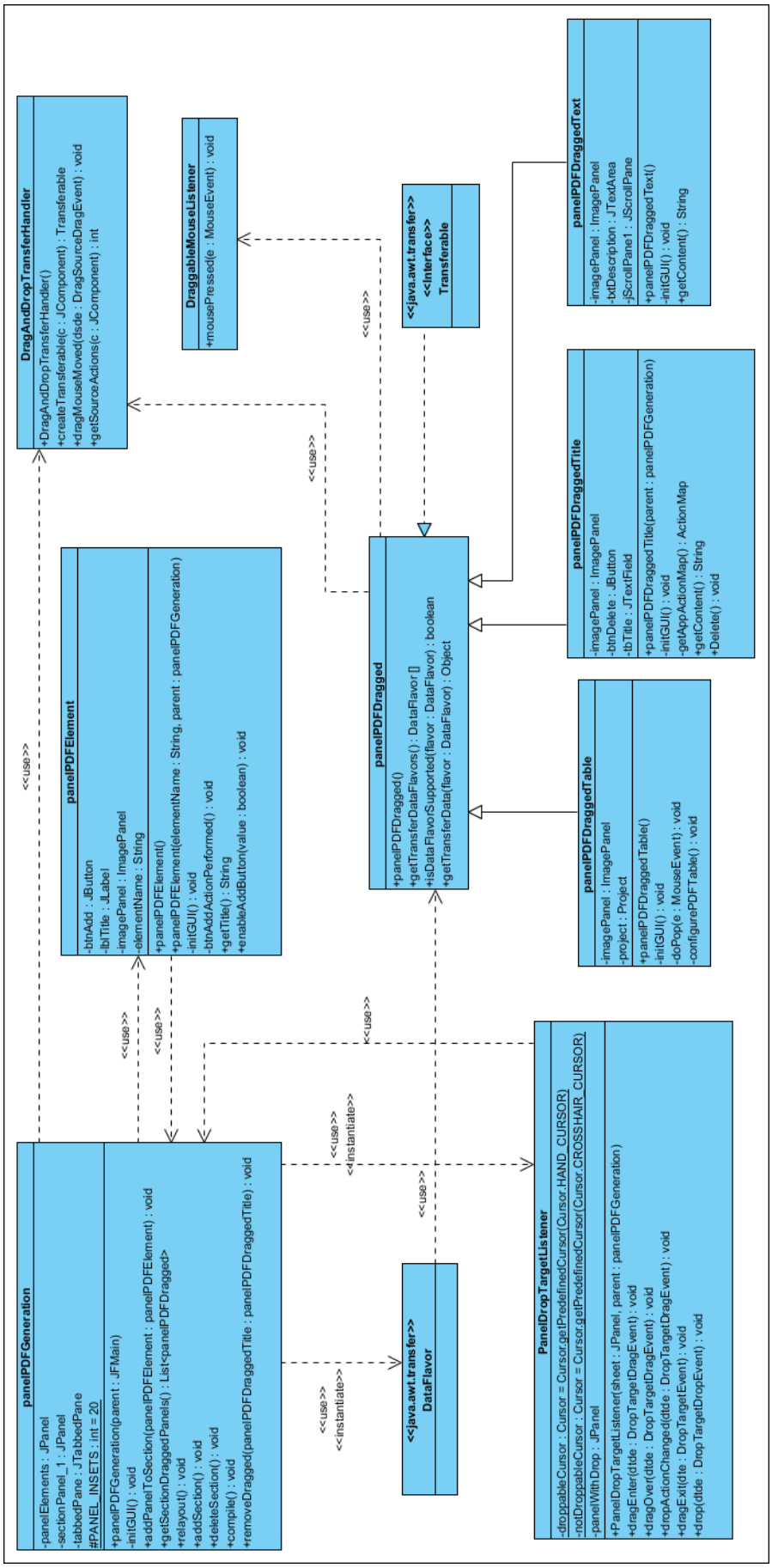

Figura 1.59: Diagrama de clases - Generación de documentos PDF - Cliente

un *Drag*, y que permite conocer cuál es su *DataFlavor* asociado y devolver el objeto correspondiente cuando se realice el evento de *Drop*.

- **DataFlavor**: es una clase de Java, del paquete *java.awt.transfer*, que indica qué datos se transmiten cuando el evento del *Drop* se produce.
- **DraggableMouseListener**: es una clase que cuando detecta el evento del ratón *pressed* inicia el proceso de *Drag & Drop*, creando el objeto que implementa la interfaz *Transferable*, que será el origen del *Drag*.
- **DragandDropTransferHandler**: clase controladora del proceso *Drag & Drop* que devuelve el objeto *Transferable* apropiado y controla el modo del proceso (copia o movimiento).
- **DropTargetListener**: es la clase que contiene la lógica para manejar el evento del *Drop*.

En la Figura 1.60 se muestra visualmente el proceso de *Drag & Drop*, de manera simplificada.

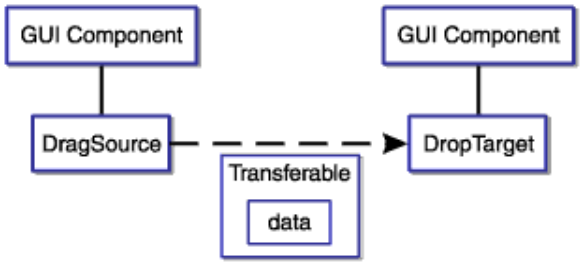

Figura 1.60: Proceso simplificado de *Drag & Drop*

1.3.5.1.3 Pruebas

1.3.5.2 Grupo funcional F7: *Generación de estadísticas*

1.3.5.2.1 Análisis de casos de uso

Este grupo funcional se considera como uno de los principales para permitir al usuario llevar un control acerca de los proyectos software, con las decisiones tomadas en ellos, los usuarios

involucrados, etc. Para facilitar dicho control, se generan gráficos estadísticos de manera automática a partir de los datos existentes en el sistema. Por tanto, esta iteración comienza con el análisis de los casos de uso que componen dicha funcionalidad, definiendo sus escenarios y diagramas de análisis.

Generación gráficos estadísticos

En la Tabla 1.12 se describe el caso de uso *Generación gráficos estadísticos*.

Nombre: Generación gráficos estadísticos
Descripción: Funcionalidad para generar y visualizar un gráfico estadístico.
Precondiciones: Que el usuario haya accedido al sistema y tenga permisos para realizar la operación.
Post-condiciones: Se genera y visualiza un gráfico estadístico
Flujo principal: 1. El jefe de proyecto selecciona un tipo de gráfico a generar. 2. El generar y visualizar un gráfico estadístico selecciona los datos a usar en ese gráfico. 3. Se validan los datos introducidos. 4. El sistema consulta los proyectos y decisiones, según la selección del usuario. 5. Se procesan los datos consultados. 6. Se genera la gráfica con los datos procesados. 7. Se visualiza el gráfico generado.
Flujo alternativo 1: datos incompletos: 1. El jefe de proyecto selecciona un tipo de gráfico a generar. 2. El jefe de proyecto selecciona los datos a usar en ese gráfico. 3. Se validan los datos introducidos. 4. Los datos son incompletos. Se muestra un mensaje y se vuelven a solicitar los datos.

Tabla 1.12: Especificación del caso de uso *Generación gráfico estadístico*

La Figura 1.61 representa el diagrama de clases de análisis para el subsistema cliente.

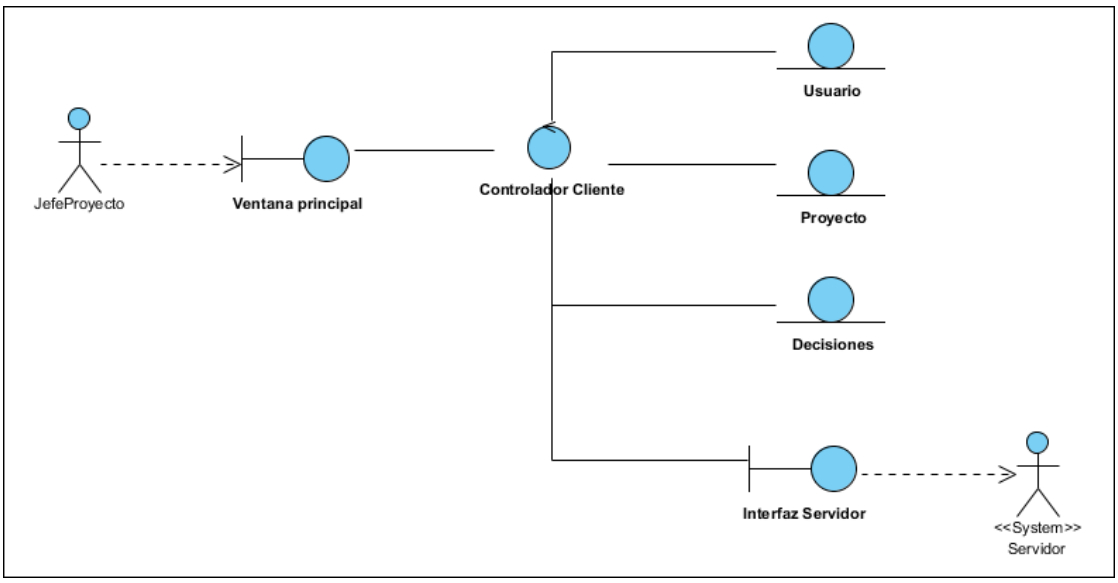

Figura 1.61: Diagrama de clases de análisis - Cliente - Generación gráficos estadísticos

1.3.5.2.2 Diseño e implementación

Como en iteraciones anteriores y de modo similar, se modelan los diagramas de secuencia para los casos de uso de este grupo funcional y se comienza con su diseño e implementación.

La generación de los gráficos estadísticos es una funcionalidad exclusiva del cliente, ya que los gráficos dependen de la tecnología empleada para representarlos, que en este caso es la librería de gráficos **JFreeChart** (ver sección **??**). Por tanto, el cliente, consultando los datos necesarios para crear un gráfico al servidor, es el encargado de generar dichos gráficos y visualizarlos.

En primer lugar, cabe destacar que el tipo de gráficos que pueden generarse y visualizarse están definidos en un fichero XML, por lo que fácilmente se podrán añadir nuevos tipos de gráficos, añadiéndolos en este XML, facilitando su extensibilidad. En este XML se define el tipo de gráfico (de barras, pastel, líneas, etc), un icono, su nombre y descripción.

Para la creación de los gráficos estadísticos, se hace uso del concepto de *dataset*, que representa el conjunto de datos que se van a representar en un gráfico. En este caso, se han utilizado tres tipos de *dataset*, proporcionados por la librería JFreeChart:

- *DefaultPieDataset*: es el conjunto de datos utilizado para crear y visualizar un gráfico de tipo pastel (o *pie*). Contiene los datos que representan cada una de las porciones del gráfico. Hereda de la clase *AbstractDataSet*.

- *DefaultCategoryDataset*: es el conjunto de datos utilizado para crear y visualizar un gráfico de barras. Contiene los datos que corresponden a un valor del eje X en el eje Y. Hereda de la clase *AbstractDataSet*.

- *CategoryDataset*: es el conjunto de datos utilizado para crear y visualizar un gráfico de líneas. Contiene los datos que corresponden a un valor del eje X en el eje Y. Hereda de la clase *AbstractDataSet*.

En la Figura **??** se muestra el diagrama de clases de diseño para esta funcionalidad, donde se puede observar como la clase controladora de esta funcionalidad (*StatisticsGenerator*) hace uso de estos *datasets*. Además, hace uso de una clase que permite leer y extraer información de archivos XML, utilizada para leer y extraer la información de los gráficos definidos en el fichero XML comentado con anterioridad.

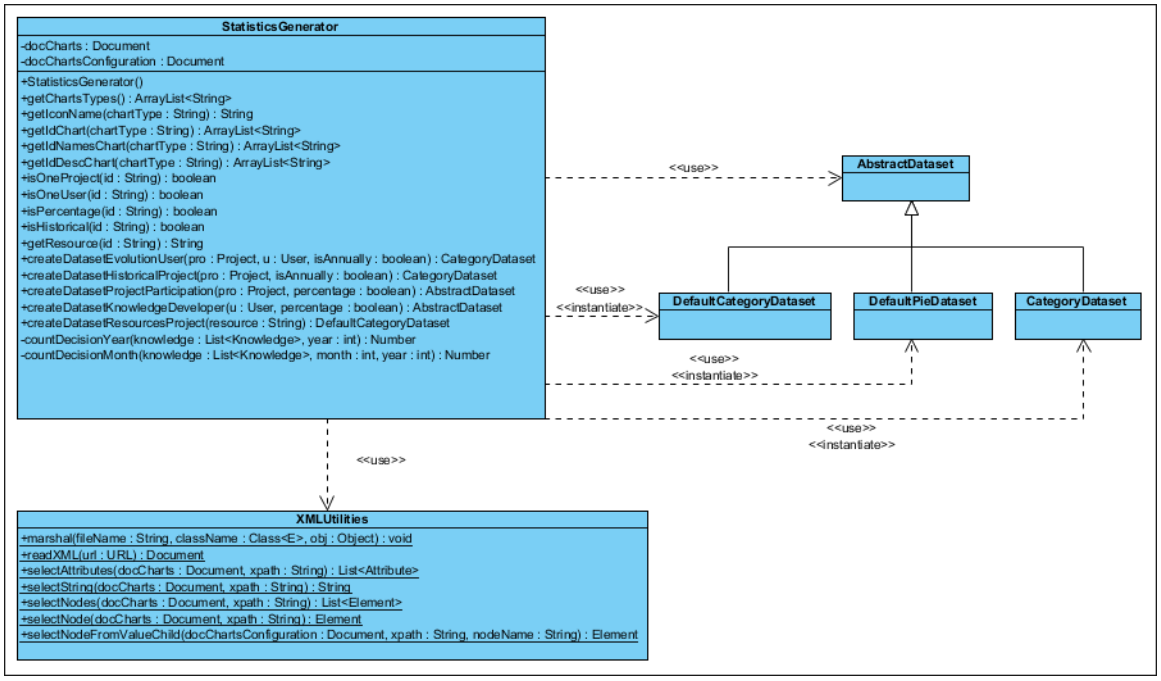

Figura 1.62: Diagrama de clases - Cliente - Generar estadísticas

Por tanto, la clase controladora se encarga de realizar las peticiones al servidor para consultar los datos que se necesitan para poder componer los *datasets* necesarios para generar los gráficos. Los datos consultados dependerán del tipo de gráfico a generar, pudiendo representar la cantidad de decisiones realizadas por un usuario, la cantidad de decisiones realizadas en un proyecto, un histórico de un proyecto, etc. En el fragmento de código 1.13 se

muestra un ejemplo de como generar un *dataset* para un diagrama de barras y de pastel que sirve para representar el número de decisiones que un determinado usuario ha realizado en todos los proyectos en los que participa.

```
// Create dataset for the chart of number of knowledge made on each project for that user
public AbstractDataset createDatasetKnowledgeDeveloper(User u, boolean percentage) throws
    RemoteException, NonPermissionRoleException, NotLoggedException, SQLException,
    Exception {
   AbstractDataset dataset = null;
   if (percentage)
      dataset = new DefaultPieDataset();
   else
      dataset= new DefaultCategoryDataset();

   int totalCount = 0;
   int knowledgeUserCount;
   Hashtable<String, Integer> parcial_counts = new Hashtable<String, Integer>();

   // Get all projects
   List<Project> projects = ClientController.getInstance().getProjects();
   List<User> usersInProject;
   for (Project p: projects) {
      // Get all the users that work in that project
      usersInProject = ClientController.getInstance().getUsersProject(p);
      if (usersInProject.contains(u)){
         // Get knowledge from user
         TopicWrapper tw = ClientController.getInstance().getTopicsWrapper(p);
         List<Knowledge> knowledgeUser = ClientController.getInstance().getKnowledgeUser(tw
             , u);
         knowledgeUserCount = knowledgeUser.size();
         parcial_counts.put(p.getName(), knowledgeUserCount);
         // Use totalCount in order to calculate the percentage in the case of PieDataSet
         totalCount += knowledgeUserCount;
         // Bar chart case
         if (!percentage)
            ((DefaultCategoryDataset)dataset).addValue(knowledgeUserCount, p.getName(), p.
                getName());
      }
   }
   // Pie Chart case
   if (percentage) {
      for (String projectName: parcial_counts.keySet()) {
         double value = ((parcial_counts.get(projectName) * 100.0) / totalCount);
         ((DefaultPieDataset)dataset).setValue(projectName, value);
      }
   }
   return dataset;
```

```
}
```

Listado 1.13: Fragmento de código para la generación de *datasets*

Para terminar, en el fragmento de código 1.14 se muestra como crear y representar el gráfico estadístico cuando ya se ha generado su *dataset*. Para ello, se hace uso de la clase *ChartFactory* de la librería **JFreeChart**. Además, una vez representado el gráfico, esta librería permite interactuar con dicho gráfico, pudiendo cambiar el tamaño de su título, el color, la leyenda, guardar el gráfico como imagen, etc.

```
private JFreeChart generatePieChart(String title, DefaultPieDataset dataset, boolean
    showLegend) {
   final JFreeChart chart = ChartFactory.createPieChart(
         title,
         dataset,
         showLegend, // legend
         true, // tooltips
         false // URLs
      );
   return chart;
}
```

Listado 1.14: Fragmento de código para la generación de gráficos

1.3.5.2.3 Pruebas

1.3.6 Iteración 8

Siguiendo los casos del grupo funcional **F8:** *Exportar información* (ver Figura 1.22) y del grupo funcional **F9:** *Gestión de idiomas* (ver Figura **??**), se abordan las siguientes tareas en esta iteración:

- Análisis de los casos de uso.
- Diseño de la funcionalidad relativa a la gestión de idiomas y a exportar la información de las decisiones.
- Implementación de dichas funcionalidades.

- Diseño e implementación de pruebas relativas a la gestión de idiomas y a exportar la información de las decisiones.

1.3.6.1 Grupo funcional F8: *Exportar conocimiento*

1.3.6.1.1 Análisis de casos de uso

En esta etapa se aborda el análisis de los casos de uso de esta funcionalidad, mostrados en la Figura 1.22. A partir del diagrama de casos de uso, se realiza su especificación y descripción de escenarios y se generan sus diagramas de clases de análisis.

Exportar información

En la Tabla 1.13 se describe el caso de uso *Exportar información*.

La Figura 1.63 representa el diagrama de clases de análisis para el subsistema cliente. La Figura 1.64 refleja el diagrama de clases de análisis para el subsistema servidor.

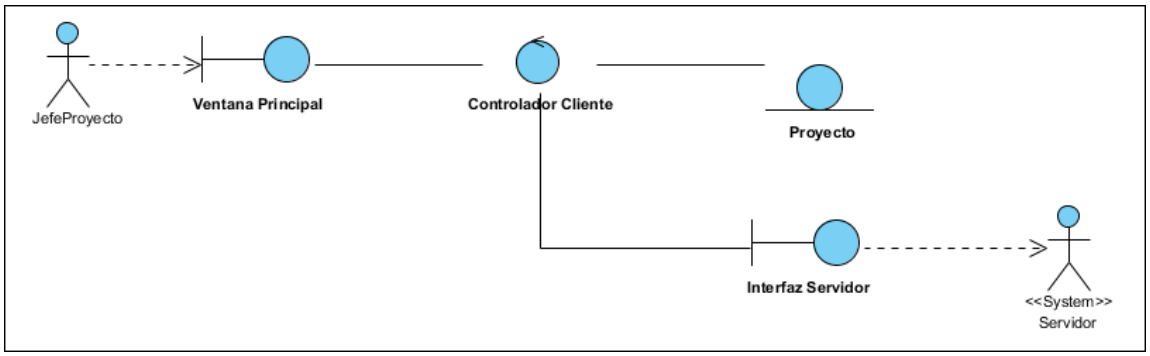

Figura 1.63: Diagrama de clases de análisis - Cliente - Exportar información

1.3.6.1.2 Diseño e implementación

Como en iteraciones anteriores y de modo similar, se modelan los diagramas de secuencia para los casos de uso de este grupo funcional y se comienza con su diseño e implementación.

Nombre: Exportar información
Descripción: Funcionalidad para exportar la información de proyectos y sus decisiones a un fichero XML.
Precondiciones: Que el usuario haya accedido al sistema y tenga permisos para realizar la operación.
Post-condiciones: Se genera y almacena un fichero XML con la información de proyectos y sus decisiones.
Flujo principal: 1. El jefe de proyecto inicia la acción de exportar información. 2. Se consulta en el sistema los proyectos disponibles. 3. El jefe de proyecto selecciona un proyecto para exportar la información de ese proyecto. 4. Se validan los datos introducidos. 5. El sistema consulta el proyecto seleccionado, sus decisiones y la información asociada a éstas. 6. Los datos consultados se serializan y se guardan en un fichero XML.
Flujo alternativo 1: no existen proyectos: 1. El jefe de proyecto inicia la acción de exportar información. 2. Se consulta en el sistema los proyectos disponibles. 3. No existen proyectos. Se muestra mensaje informando de esta situación.

Tabla 1.13: Especificación del caso de uso *Exportar información*

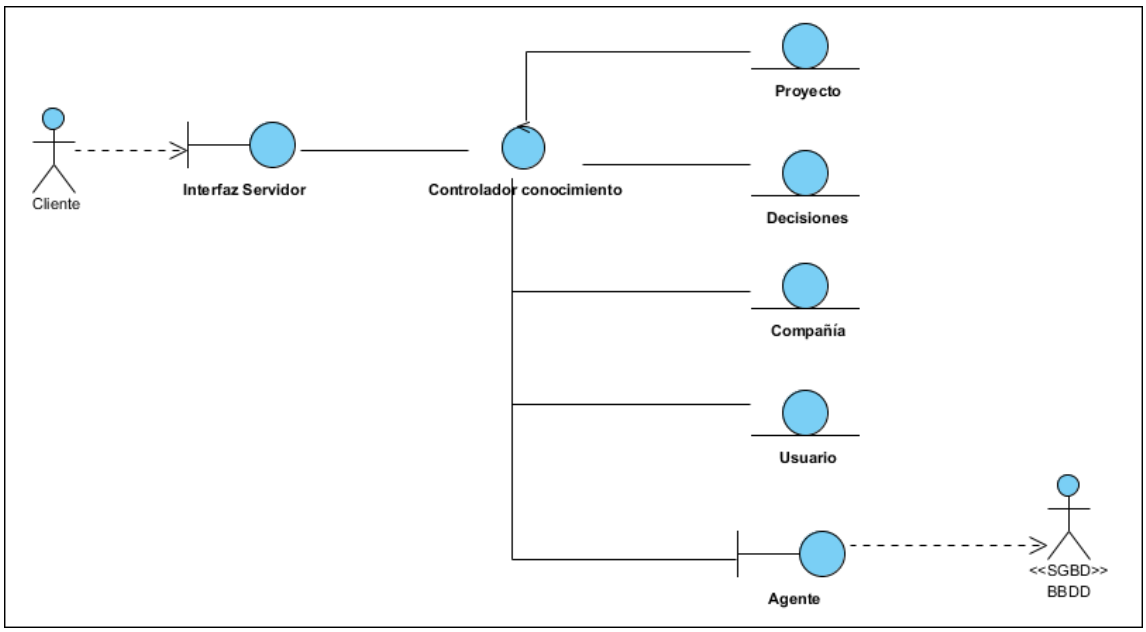

Figura 1.64: Diagrama de clases de análisis - Servidor - Exportar información

Servidor

En el diseño e implementación de esta funcionalidad, cabe destacar la utilización de **JAXB** (ver sección **??**) para serializar las clases que componen la jerarquía de decisiones y toda su información relacionada (ver Figura 1.42) a un archivo XML. Para ello, se utilizan anotaciones sobre las clases, indicando que esa clase y sus atributos deben convertirse a un nodo XML. Además, es necesario que exista una clase que represente al nodo raíz del fichero XML y que, por tanto, contenga toda la lista de decisiones. Dicha clase, como ya se ha comentado anteriormente, es la clase llamada *TopicWrapper*.

En el fragmento de código 1.15 se muestran las anotaciones de JAXB realizadas sobre la clase *TopicWrapper*, que engloba a todos los *topics* de un proyecto. En el fragmento de código 1.16 se muestran las anotaciones de JAXB para la clase *Knowledge*, para poder serializar sus atributos. En este caso cabe destacar la anotación *@XmlJavaTypeAdapter* sobre el atributo de tipo *fecha*, que se usa para invocar a una clase de Java encargada de formatear una fecha, devolviendo una cadena de texto con el formato deseado, que es *MM/dd/yyyy* en este caso.

De modo similar se realizan las anotaciones en el resto de clases que componen el diagrama de clases mostrado en la Figura 1.42.

```
@XmlRootElement (name = "Topics" )
@XmlAccessorType( XmlAccessType.FIELD )
public class TopicWrapper implements Serializable {

   private static final long serialVersionUID = -2825778853241760000L;

   @XmlElement( name = "Topic" )
   private ArrayList<Topic> topics = new ArrayList<Topic>();
```

Listado 1.15: Anotaciones de JAXB sobre la clase *TopicWrapper*

```
@XmlAccessorType( XmlAccessType.FIELD )
public abstract class Knowledge implements Serializable {

   private static final long serialVersionUID = -7039151251262020404L;

   protected int id;
   protected String title;
   @XmlJavaTypeAdapter(DateAdapter.class)
   protected Date date;
   protected String description;
   protected KnowledgeStatus status;
   @XmlElement( name = "Author" ) protected User user;
   @XmlElement( name = "File" )
   private Set<File> files = new HashSet<File>();
```

Listado 1.16: Anotaciones de JAXB sobre la clase *Knowledge*

Gracias a las anotaciones de JAXB, el proceso de serialización (o *marshal* en inglés) es muy sencillo: sólo se necesita el tipo de la clase que contiene la anotación de elemento raíz del XML (*TopicWrapper* en este caso) y el objeto del dominio a serializar, obteniendo en este caso un array de bytes con la información serializada y en formato XML. En el fragmento de código 1.17 se muestra este método de serialización.

```
// Marshal domain class into XML file, using JAXB
public static <E> ByteArrayOutputStream marshal(Class<E> className, Object obj) throws
    JAXBException {
   JAXBContext jaxbContent = JAXBContext.newInstance(className);

   Marshaller marshaller = jaxbContent.createMarshaller();
   marshaller.setProperty(javax.xml.bind.Marshaller.JAXB_ENCODING, "UTF-8");
   marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
    marshaller.marshal(obj, baos);
    return baos;
}
```

Listado 1.17: Método de serialización utilizando JAXB

Cliente

Respecto a esta funcionalidad en el cliente, se envía al servidor el proyecto del cuál se desea extraer y exportar su información, recibiendo el array de bytes con toda la información serializada y guardando ese array en un fichero XML elegido por el usuario.

1.3.6.1.3 Pruebas

1.3.6.2 Grupo funcional F9: *Gestión de idiomas*

1.3.6.2.1 Análisis de casos de uso

Al igual que en el resto de casos, se comienza analizando los casos de uso que componen este grupo funcional, realizando las especificaciones de dichos casos de uso y generando sus diagramas de clases de análisis.

1.3.6.2.2 Diseño e implementación

El sistema debe dar soporte a la internacionalización, por lo que se ha diseñado e implementado un gestor de lenguajes. Dicho gestor será el encargado de, según el idioma elegido en la aplicación, mostrar la diferente información de la interfaz gráfica de usuario en ese idioma. Para ello, se han creado ficheros de propiedades (o *properties)* que contienen las cadenas de texto a internacionalizar, utilizando uno u otro según el idioma escogido. El nombre de dichos ficheros terminan con el código del idioma de cada país, para poder cargar uno u otro según el idioma.

Para terminar, señalar que los idiomas disponibles para configurar la aplicación se encuentran un fichero XML, para que no sea necesario una conexión a base de datos y se pueda modificar el idioma de la aplicación sin tener que acceder previamente a dicha base de datos.

Además, se facilita la extensibilidad y adición de nuevos idiomas, ya que sólo abría que crear su fichero de propiedades y agregar ese idioma al fichero XML.

En este PFC se han incluido los idiomas Español (España) e Inglés (Americano).

1.3.6.2.3 Pruebas

1.4 Fase de Transición

En esta fase, compuesta de una única iteración, es donde el producto software se prepara para su entrega al cliente, incluyendo los últimos detalles de implementación y pruebas (ver Figura **??**).

Al alcanzar esta última iteración, se prepara la versión ejecutable del producto al cliente, así como se realizan las últimas pruebas globales para comprobar el correcto funcionamiento del sistema.

Además, se realiza la documentación del producto software y se redactan los manuales de configuración y manuales de usuario del sistema. Dichos manuales pueden encontrarse en los apéndices XXX.

Bibliografía

[1]

[2] Carmel, E., J.A. Espinosa, and Y. Dubinsky: *"follow the sun" workflow in global software development*. Journal of Management Information Systems, 27(1):17–38, 2010.

[3] Fitzgerald, B. *et al.*: *Global software development: a multiple-case study of the realisation of the benefits*. 2010.

[4] Holzner, B. and J.H. Marx: *Knowledge application: The knowledge system in society*. Allyn and Bacon Boston, 1979.

[5] Insight, G.: *Inc (2005)." executive summary: The comprehensive impact of offshore software and it services outsourcing on the us economy and the it industry."*. Information Technology Association of America.

[6] Rooksby, J., I. Sommerville, and M. Pidd: *A hybrid approach to upstream requirements: Ibis and cognitive mapping*. In *Rationale management in software engineering*, pages 137–153. Springer, 2006.