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

Introducción

1.1 Estructura del documento

Capítulo 2

Motivación y Objetivos del Proyecto

En este capítulo se presenta la motivación para el desarrollo del presente PFC, así como los objetivos que deben alcanzarse tras concluir el desarrollo del mismo.

2.1 Motivación

La globalización y competitividad de los mercados ha obligado, y está obligando, a que las empresas vayan evolucionando y adaptando sus procesos de negocio y sus procesos de gestión de proyectos para adaptarse a esta nueva tendencia de mercados globales. Estas nuevas estrategias requieren de cooperación intensiva entre las compañías, de nuevos procesos de gestión de proyectos, de mejoras en las comunicaciones entre organizaciones que se encuentran en diferentes husos horarios, etc. Todo ello ha provocado que aparezcan nuevas organizaciones, llamadas *organizaciones virtuales*, que son aquellas que se encuentran deslocalizadas, con proyectos en los que participan empleados de cualquier parte del mundo y que utilizan las tecnologías de la información para sus procesos de comunicación, coordinación y control [**?**].

Anteriormente a este marco de globalización, las empresas gestionaban sus proyectos de manera co-localizada, es decir, en el mismo lugar y al mismo tiempo. Sin embargo, ahora se ven obligadas a desarrollar procesos de gestión que permitan la gestión de *proyectos virtuales*, es decir, proyectos que ya no están centralizados en un único lugar y se realizan de manera distribuida. En la Figura 2.1 se muestran las dimensiones de los proyectos que las compañías deben gestionar [**?**].

Sin embargo, este marco de *proyectos virtuales* y, por tanto, del desarrollo de los proyectos de manera distribuida, acarrea una serie de dificultades, siendo las más destacables los

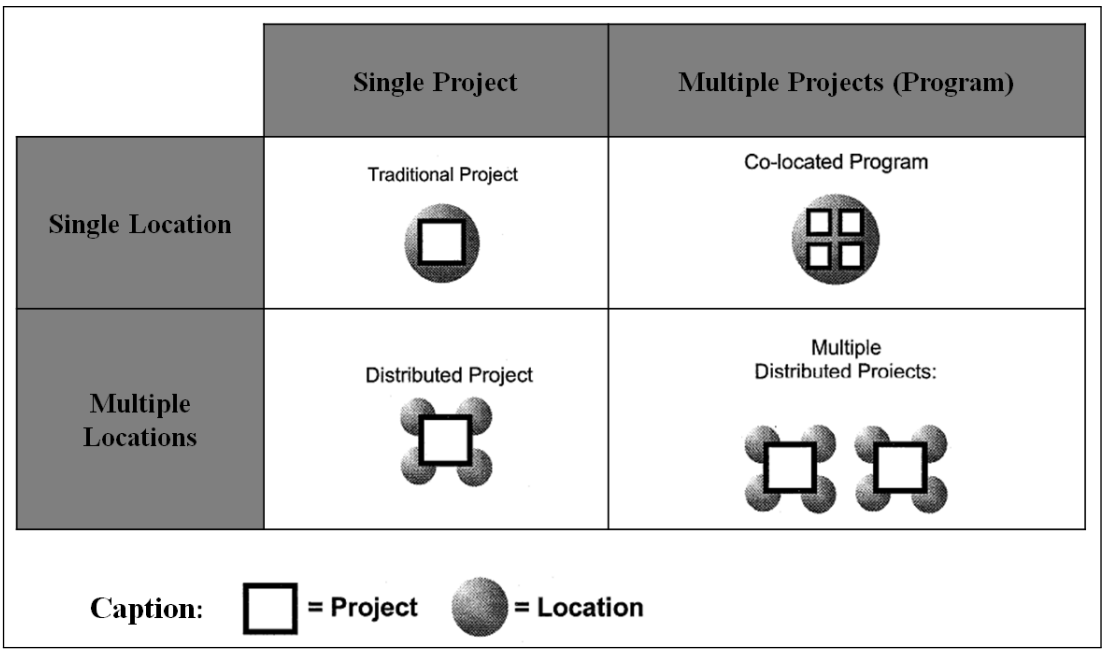

Figura 2.1: Proyectos según su localización

problemas de comunicación, coordinación y control entre las diferentes localizaciones de los centro de desarrollo de las organizaciones.

Otro de los problemas encontrados, muy relacionado con el anterior, es como las empresas, que ahora se encuentran en localizaciones diferentes y que tienen mayor problema para la comunicación y control, pueden crear, organizar y gestionar su conocimiento organizacional, recurso intangible imprescindible para que la empresa pueda innovar y ser competitiva.

Centrándonos en empresas dedicadas al sector de desarrollo software, gran parte de este conocimiento organizacional lo componen las decisiones tomadas durante el ciclo de vida de desarrollo de un proyecto software, tales como decisiones de diseño, decisiones de análisis del proyecto, etc., que con frecuencia no se documentan de manera adecuada o se documentan por un equipo de desarrollo y el resto no son conscientes de que dicha documentación existe, dificultando la posterior reutilización de dicho conocimiento. Este es un problema importante del desarrollo global de software, ya que al generarse conocimiento, información y toma de decisiones en los diferentes equipos de desarrollo, ésta queda distribuida en numerosas ocasiones si no es guardada previamente en un repositorio global. Es por este motivo que se produce una pérdida de dicha información y no se aprende de las decisiones tomadas en los proyectos.

Por tanto, la motivación de este PFC viene dada por la necesidad de resolver o minimizar algunos de los problemas que aparecen en las compañías que desarrollan software de manera global. Particularmente, la herramienta estará enfocada a la gestión de decisiones de proyectos software en desarrollo global, intentando minimizar todo lo posible los problemas de comunicación y control y falta de reutilización de "lecciones aprendidas" que aparecen en esta nueva forma de desarrollo deslocalizado (ver Capítulo 3 para una revisión de la literatura acerca del Desarrollo Global, Gestión de Conocimiento, Gestión de Decisiones y *Design Rationale*).

2.2 Objetivos

El objetivo principal del PFC consistirá en el diseño y construcción de una **herramienta** basada en Java que permita dar soporte a la **gestión de decisiones en proyectos software** en el paradigma de **Desarrollo Global de Software**. Por tanto, dicha herramienta debe permitir la creación, almacenamiento, recuperación, transmisión y aplicación de decisiones abordadas en un proyecto software, realizado de manera deslocalizada. Además, debe permitirse gestionar también los proyectos software sobre los que se toman decisiones.

De este modo, se intenta reducir o eliminar algunas de las problemáticas que aparece en el desarrollo global, como es la falta de comunicación y control entre los equipos de desarrollo, así como la falta de reutilización del conocimiento adquirido al desarrollar proyectos previos.

A continuación, se enumeran los objetivos parciales que se deben cumplir para alcanzar el objetivo principal. Para ello, la herramienta deberá:

- Representar las decisiones tomadas en los proyectos software de una manera visual, clara e intuitiva, evitando malentendidos, puesto que aunque el inglés pueda utilizarse como medio de comunicación, con frecuencia no es la lengua nativa de todos los miembros de los equipos de desarrollo y esto da lugar a malentendidos. De este modo, se representan dichas decisiones de la misma manera para todos los equipos de desarrollo y empleados, facilitando su transmisión y evitando confusiones.

- Proveer formularios para crear y modificar decisiones, estandarizando el conocimiento almacenado y facilitando su posterior recuperación y comunicación.

- Crear un mecanismo de comunicación síncrona, es decir, que si un empleado se encuentra visualizando decisiones de un determinado proyecto en la aplicación y, en ese momento, otro empleado de otra localización diferente realiza un cambio sobre ese proyecto, dicho cambio debe ser notificado al primer empleado en tiempo real, actualizando su vista. Además, deberá crearse también una notificación o alerta de manera automática, favoreciendo de este modo la comunicación asíncrona.

- Adaptación a diferentes idiomas.

- Dar soporte a la gestión de proyectos software, proveyendo formularios para dar de alta a nuevos proyectos, así como para su posterior modificación. De este modo, se favorece una estructura común de proyectos y se facilita su control entre los diferentes equipos de desarrollo que trabajan en esos proyectos.

- Permitir aspectos de control de proyectos por parte de los jefes de proyecto, generando gráficos e informes.

- Desarrollar un mecanismo de reutilización de decisiones entre diferentes proyectos software, de modo que decisiones de proyectos ya pasados puedan aplicarse a nuevos proyectos, con características similares. Para ello, se utilizan técnicas de Inteligencia Artificial, como es el Razonamiento Basado en Casos.

2.2.1 Resumen de objetivos

A modo de síntesis, en la Tabla 2.1 se resumen los objetivos que el sistema a desarrollar debe cumplir para alcanzar su objetivo principal. En dicha tabla se muestran además los desafíos del Desarrollo Global de Software que se intentan mitigar con la compleción de cada uno de esos objetivos.

Destacar que esta herramienta es en cierto modo un sistema para la gestión de conocimiento, ya que como se detalla en el Capítulo 3, permite la creación, almacenamiento, recuperación, transmisión y aplicación de decisiones. En el caso concreto del presente PFC, dicho conocimiento son las decisiones y alternativas planteadas durante las fases del ciclo de vida de los proyectos software, siguiendo de este modo el enfoque de *Rationale* (ver capítulo 3).

Id. Objetivo	Objetivo	Desafío GSD
O1	Permitir el acceso al sistema desde diferentes localizaciones	Comunicación y Control
O2	Facilitar y favorecer la gestión de decisiones en los proyectos software, evitando malentendidos debidos a diferencias socio-culturales	Comunicación y Gestión Conocimiento
O3	Favorecer la representación y visualización de la información almacenada, de manera clara e intuitiva, evitando malas interpretaciones	Comunicación y Gestión Conocimiento
O4	Facilitar la comunicación entre los equipos de desarrollo, notificando posibles cambios al instante	Comunicación
O5	Adaptación a diferentes idiomas	Comunicación
O6	Facilitar la gestión de proyectos software, utilizando una estructura común para su creación, modificación, etc.	Comunicación y Control
O7	Favorecer aspectos de control de proyectos, tales como la situación actual del proyecto, desarrolladores que en ellos trabajan, decisiones tomadas, etc.	Control
O8	Facilitar la reutilización de información entre proyectos software, aconsejando decisiones de proyectos similares	Control y Gestión Conocimiento

Tabla 2.1: Objetivos a cumplir en el PFC

Para terminar, a continuación se resumen los términos de mayor relevancia y que con más frecuencia aparecen en el presente documento, para facilitar la lectura y comprensión de posteriores capítulos:

- **Empresa**: el término *empresa*, o *compañía*, en el contexto de este PFC hace referencia a una organización dedicada y/o involucrada en el desarrollo de productos software.
- **Proyecto**: el término *proyecto* hace referencia a un proyecto software, desarrollado por los empleados de una compañía y cuyo desarrollo originará un producto software.
- **Decisión**: este término aparece mencionado en numerosas ocasiones a lo largo del presente documento. Dicho término hace referencia a las diferentes decisiones que los ingenieros informáticos realizan sobre un proyecto software, en cualquiera de las fases de su ciclo de vida. Así, dichas decisiones podrían ser decisiones de diseño del proyecto software, decisiones realizadas en el análisis, decisiones acerca de qué tecnologías utilizar, etc.
- **Usuario**: este término se refiere a cualquier empleado de una compañía dedicada al desarrollo software, independientemente del rol que dicho empleado tenga dentro de la compañía, y que puede utilizar el sistema.
- **Notificación**: este término, nombrado en algunas ocasiones como *alerta*, hace referencia a un aviso que el sistema genera para informar a los usuarios de cambios producidos en los proyectos.
- **Vista**: este término hacer referencia a una parte de la interfaz gráfica de usuario que presenta elementos visuales con información que es relevante al usuario.

Capítulo 3

Estado del Arte

La forma de desarrollar software está evolucionando hacia el Desarrollo Global de Software, donde los proyectos software se desarrollan por diferentes equipos de desarrollo que se encuentran deslocalizados en diversos países. Este nuevo paradigma de desarrollo implica una serie de ventajas, pero también introduce una serie de desafíos, siendo uno de estos desafíos la Gestión de Conocimiento.

En Desarrollo Global, la información proviene de fuentes muy diversas, en diferentes formatos e incluso idiomas, lo que dificulta conocer las decisiones tomadas en proyectos software, su documentación y, en general, la gestión de conocimiento y su reutilización. Es por ello que existe una tendencia a usar el método *Rationale*, el cuál proporciona mecanismos para capturar y representar las decisiones tomadas en cualquier fase del ciclo de vida del desarrollo del proyecto software, así como todo el razonamiento que se hace para tomar esas decisiones. Por tanto, gracias a *Rationale*, se facilita la generación, almacenamiento y representación de decisiones y su razonamiento.

Además, una vez que este tipo de información ha sido capturada y almacenada, otro aspecto destacado a tener en cuenta, y que también aparece en la Gestión de Conocimiento, es la recuperación y aplicación de dicho conocimiento. Por tanto, es importante que las decisiones capturadas y almacenadas por el método *Rationale* en los proyectos, puedan recuperarse y reutilizarse en otros proyectos con características similares. Para ello, puede utilizarse una técnica de Inteligencia Artificial, como es el Razonamiento Basado en Casos, que puede proporcionar soluciones a distintos problemas en base a experiencias pasadas.

En la imagen 3.1 se refleja de manera visual cómo estos conceptos están relacionados entre sí, según se ha detallado anteriormente.

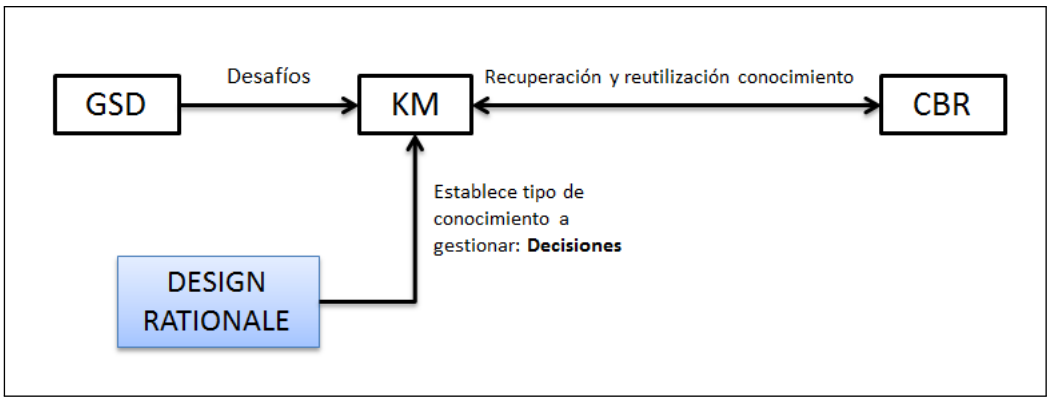

Figura 3.1: Relaciones entre Desarrollo Global, Gestión del Conocimiento, *Rationale* y Razonamiento Basado en Casos

De este modo, en las siguientes secciones se presenta el estado del arte sobre Desarrollo Global de Software, así como se detalla el método de *Rationale* y el Razonamiento Basado en Casos.

3.1 Desarrollo Global de Software

En los últimos años, la globalización económica está provocando que la industria y el comercio se adapten a nuevos modelos de negocios, en búsqueda de aumentar la competitividad y minimizar los costes, por lo que las empresas se han visto obligadas a evolucionar en su modo de trabajo. Así, se ha cambiado el concepto de *mercado nacional* por el de *mercado global* [**?**] [**?**].

Centrándose en el desarrollo de software, hasta hace unos años el desarrollo era llevado a cabo por Centros de Desarrollo Software (en adelante CDS) que estaban co-localizados en un mismo edificio. Sin embargo, debido a este nuevo modelo de economía global, la manera de desarrollar software está evolucionando en los últimos años, tendiendo a un desarrollo distribuido del mismo. Los CDS comenzaron a distribuirse en diferentes ciudades e incluso diferentes provincias de un mismo país, hasta llegar al punto en que los CDS se encuentran distribuidos en diferentes países y continentes. Esto es lo que se conoce como **Desarrollo Global de Software**, denominado a partir de este punto como GSD, por sus siglas en inglés Global Software Development. En la Tabla 3.1 se muestran los diferentes modelos de desarrollo de software existentes, en base a la dispersión geográfica.

EL GSD puede definirse como: "*el desarrollo de software que se realiza en localizaciones separadas geográficamente más allá de fronteras nacionales, de manera coordinada e involucrando participación en tiempo real (síncrona) e interacción asíncrona*" [**?**].

Localización	Tipo de desarrollo
Mismo lugar	Desarrollo tradicional (Co-localizado)
Mismo país	Desarrollo Distribuido de Software (DSD)
Distintos países	Desarrollo Global de Software (GSD)

Tabla 3.1: Modelos de desarrollo de software según dispersión geográfica

En este nuevo modelo de GSD, además de tener en cuenta que los *stakeholders*[1] se encuentran distribuidos geográficamente, hay que contar con el número de compañías que colaboran en un mismo proyecto. Un mismo proyecto bien puede ser desarrollado por una misma compañía, pero con CDS distribuidos en diferentes países, o bien puede llevarse a cabo por un conjunto de compañías que colaboran o están subcontratadas por la primera [**?**]. Así, surgen los conceptos de outsourcing, offshoring y nearshoring.

Por **outsourcing** se entiende la externalización o subcontratación de uno o varios servicios a terceros, ya que resulta más rentable, en términos de disminución de costes, que la compañía subcontratada realice estos servicios, ya que la mano de obra puedes ser más barata en un país diferente al país de la compañía contratante [**?**].

Relacionado con el *outsourcing*, está el concepto de **offshoring**, entendido como la relocalización de actividades de negocio, mediante empresas filiales de otros países [**?**]. Si estos países implicados en el proceso son relativamente cercanos al país de origen de la compañía que realiza el *outsourcing*, se habla de **nearshoring** [**?**].

En GSD, estas prácticas son cada vez más utilizadas, debido a la rentabilidad que proporcionan a la compañía. En la Figura 3.2 se puede observar el crecimiento del *offshoring* de los últimos años y la reducción de costes que conlleva.

Como se puede apreciar, el GSD y sus prácticas introducen una serie de beneficios o ventajas, pero también de dificultades o inconvenientes a tener en cuenta. En el siguiente apartado se desarrollan los más importantes.

[1] Stakeholders: *aquellos que pueden afectar o son afectados por las actividades de una empresa* [**?**].

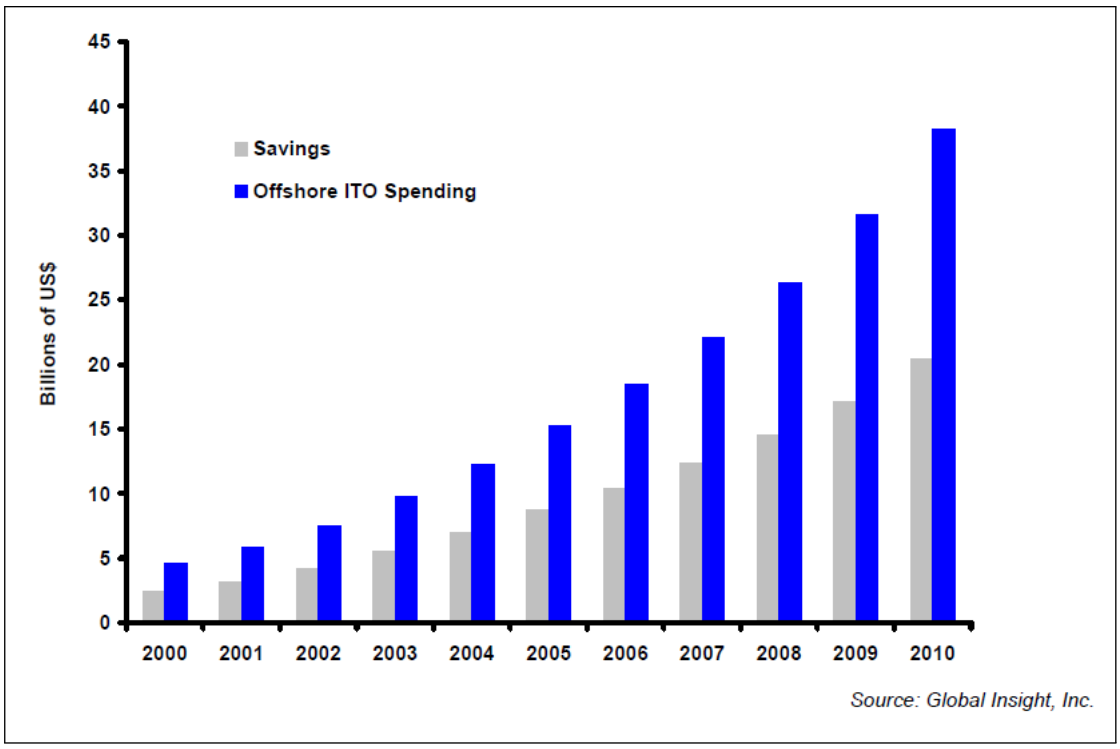

Figura 3.2: Incremento en actividades de *offshoring* y ahorro que conlleva [**?**]

3.1.1 Beneficios del Desarrollo Global de Software

A continuación se describen los beneficios más destacables del Desarrollo Global de Software, según [**?**] y [**?**].

3.1.1.1 Reducción de costes

La diferencia de salarios entre países puede ser enorme. Por ejemplo, el salario de un ingeniero de software en Estados Unidos puede llegar a ser varias veces más alto que el salario de un ingeniero similar en Asia o Sudamérica, por lo que las compañías buscan localizaciones alternativas que permitan recortar el coste de la mano de obra.

Gracias al GSD, las compañías tienen acceso a mano de obra más barata, debido a su deslocalización en diferentes países, pudiendo encontrar salarios mucho menores que en el país de origen. Ya que las compañías siempre buscan reducir el coste, éste ha sido el principal motivo que ha impulsado el GSD.

3.1.1.2 Aumento de la competitividad

El GSD provee la posibilidad de poder encontrar mano de obra más cualificada o más experimentada en diferentes países, de modo que las compañías pueden contratar esos recursos personales y expandir su desarrollo a zonas más cualificadas y competitivas.

3.1.1.3 Disminución de tiempo de lanzamiento al mercado

En GSD, se puede aplicar el modelo de desarrollo "*follow-the-sun*". Esto significa que, gracias a que existen varios centros de desarrollo en diferentes países, una compañía puede aprovechar las diferencias horarias entre sus diferentes centros de desarrollo para estar desarrollando software las 24 horas del día (ver Figura 3.3). De este modo, se consigue maximizar la productividad y disminuir el tiempo de salida al mercado (*time-to-market)* de un producto software.

De este modo, cada uno de los CDS localizados en cada país, pueden ir realizando diferentes actividades, como, por ejemplo, diseño, implementación, pruebas, etc. Para que este modelo de trabajo sea efectivo, debe existir una coordinación entre todos los CDS, que es una de las dificultades del GSD, como se comentará posteriormente.

Figura 3.3: Modelo de desarrollo *Follow-the-sun* [**?**]

3.1.1.4 Proximidad al mercado y al cliente

Al establecer filiales en aquellos países donde se localizan potenciales clientes, el GSD permite el desarrollo de software de una manera más cercana a sus clientes, aprovechando el conocimiento del mercado local y, por tanto, sus necesidades.

3.1.1.5 Mejoras en la innovación

Al disponer de CDS en diferentes países, se puede tomar ventaja de las diferentes nacionalidades, culturas, experiencias y habilidades de cada uno de ellos, pudiendo así evolucionar, innovar y enriquecer la compartición de conocimiento.

Como se ha visto, el GSD acarrea una serie de ventajas y beneficios que lo han convertido en un nuevo modelo de desarrollo utilizado por muchas empresas del sector. Sin embargo, existen también una serie de dificultades y problemas derivados de la deslocalización de los recursos, que se detallan en el siguiente apartado.

3.1.2 Desafíos del Desarrollo Global de Software

El GSD provee a las compañías y organizaciones unos beneficios muy prometedores. Sin embargo, existen dificultades a las que las compañías tienen que hacer frente, todas ellas debidas a la distancia y a la deslocalización (ver Figura 3.4) [**?**] [**?**].

Según [**?**], existen tres tipos de distancia que dificultan el GSD:

- **Distancia geográfica**. En GSD, se define como "*la medida de esfuerzo que un individuo necesita realizar para visitar otro punto, alejado del primero*". Por ejemplo, dos lugares dentro del mismo país con un enlace aéreo directo y vuelos regulares, se pueden considerar relativamente cercanos, aunque estén separados por grandes distancias kilométricas. Sin embargo, no se puede decir lo mismo de dos lugares que están cerca geográficamente (separación de pocos kilómetros) pero con poca infraestructura de transporte. Este último caso tendría una elevada distancia geográfica.

- **Distancia temporal**. En GSD, se define como "*la medida de la deslocalización en tiempo, experimentada por dos individuos que desean interactuar*". Esta distancia

normalmente va unida a la anterior, provocando que existan husos horarios diferentes entre dos puntos debidos a la distancia geográfica existente entre dichos puntos.

- **Distancia socio-cultural**. Se define como "*la medida en que un individuo comprende las costumbres (símbolos, normas y valores sociales) y cultura de otro individuo*". Aunque esta distancia es muy frecuente en el GSD, también aparece en equipos de desarrollo co-localizados, ya que cada miembro del equipo puede tener nacionalidades y culturas diferentes. Este tipo de distancia es el mayor desafío a superar en el GSD, ya que con mucha frecuencia se producen conflictos y malentendidos entre los diferentes equipos de desarrollo, debido al carácter multicultural y multinacional del GSD.

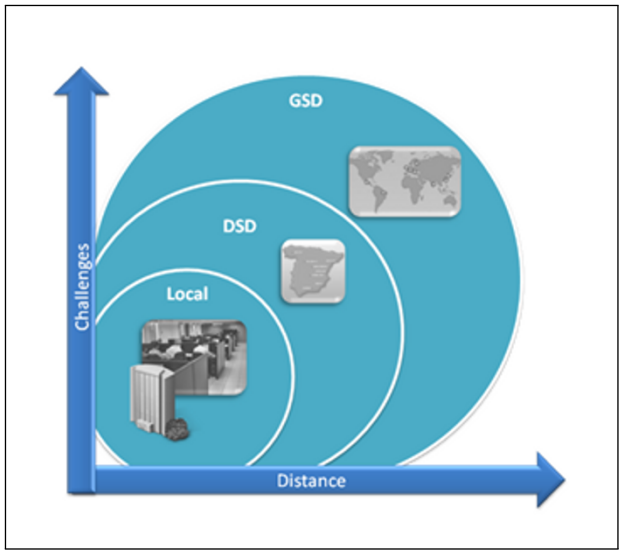

Figura 3.4: Influencia de la distancia en el desarrollo de software [**?**]

Estas distancias afectan a los tres grandes procesos que intervienen en el proceso de desarrollo de software: **comunicación**, **coordinación** y **control**, siendo los de comunicación y control los más problemáticos en GSD, según [**?**].

Por tanto, los desafíos y dificultades del GSD se agrupan en tres categorías, lo que se conoce como las tres "C" [**?**]:

- Desafíos en la ***Comunicación***, es decir, en el proceso de intercambio de conocimiento e información entre individuos.

- Desafíos en la ***Coordinación***, relacionados con objetivos e intereses comunes.

- Desafíos en el ***Control***, relacionados con la gestión del proyecto, informes, progreso, etc.

En la Tabla 3.2 se observa como afecta cada una de las distancias a los procesos de desarrollo software.

Proceso	**Distancia**		
	Geográfica	Temporal	Socio-Cultural
Comunicación	Dependiente de la tecnología	Uso de comunicación asíncrona	Malentendidos
Coordinación	Falta de conciencia de equipo	Modificación de calendarios laborales	Falta de confianza
Control	Incrementa la dificultad de gestión y control	Control asíncrono sobre rescursos remotos	Diferencias culturales en mecanismos de control

Tabla 3.2: Procesos de desarrollo software afectados por las distancias en GSD

En los siguientes apartados se resumen los desafíos más destacables:

3.1.2.1 Desafíos en la comunicación

Debido a la distancia geográfica, temporal y socio-cultural entre los distintos CDS, aparecen las siguientes dificultades en la comunicación [**?**] [**?**] [**?**].

3.1.2.1.1 Falta de comunicación

La posibilidad de un intercambio de información *face-to-face* (cara a cara, en español) desaparece o es prácticamente nula debido a la deslocalización de los equipos de desarrollo. Por ello, se depende casi en la totalidad de herramientas de comunicación, que son muy variadas y pueden no seguir estándares de comunicación, por lo que se provocan malentendidos.

3.1.2.1.2 Aumento del esfuerzo en la comunicación

Debido al uso de herramientas de comunicación, los tiempos de respuesta suelen ser altos y con frecuencia no se conoce personalmente a los miembros que intervienen en la comunicación ni su disponibilidad, por lo que la comunicación cada vez se vuelve más escasa y de menor calidad.

3.1.2.1.3 Interrupciones

Al encontrarse muchas veces los CDS en diferentes husos horarios, resulta muy complicado el poder contactar con individuos de otros equipos de trabajo y, en multitud de ocasiones, se produce la comunicación en horarios no adecuados o se producen interrupciones en el trabajo de un equipo, lo que aumenta el malestar en el trabajo.

3.1.2.1.4 Malentendidos debidos a diferencias lingüísticas

En GSD con frecuencia se producen malentendidos al comunicarse, debido a que los equipos de desarrollo hablan idiomas nativos diferentes. Estos fallos en la comunicación repercutirán posteriormente en el desarrollo, sobre todo si son malentendidos producidos durante la fase de análisis de requisitos del proyecto.

3.1.2.1.5 Malentendidos culturales

Al establecer una comunicación entre personas que no conocen totalmente las costumbres y cultura de la otra parte, puede provocar situaciones incómodas de manera involuntaria. Por ejemplo, en USA, prefieren especificar cada detalle en un documento, utilizar el teléfono y el e-mail de manera informal, mientras que en Japón prefieren el uso de medios más formales y prefieren la comunicación verbal a documentos escritos [**?**].

Por otro lado, en GSD el marco socio-cultural puede influir en opiniones diferentes sobre la propia naturaleza de llevar a cabo los procesos de desarrollo.

3.1.2.2 Desafíos en el control

Debido a la distancias existentes entre los distintos CDS, aparecen las siguientes dificultades en el control [**?**] [**?**].

3.1.2.2.1 Asignación de roles y estructuras de equipo

La deslocalización de los equipos de desarrollo complica y aumenta los costes en la asignación de roles y estructuras de equipos.

3.1.2.2.2 Gestión de los artefactos del proyecto

Cuando un proyecto involucra miembros de diferentes compañías o de diferentes CDS de la misma organización, hacer cumplir los procesos y normas del artefacto software es particularmente importante para mantener la consistencia e interoperabilidad entre los diferentes artefactos del proyecto.

3.1.2.2.3 Diferentes percepciones de la autoridad

La naturaleza de la autoridad dentro de un proyecto puede variar entre las diferentes culturas, por lo que pueden surgir problemas en el control de un proyecto si los equipos en diferentes lugares esperan que se maneje de manera diferente, y si estas expectativas diferentes no se identifican desde el principio.

3.1.2.3 Desafíos en la Gestión de Conocimiento

En la **Gestión del Conocimiento** también aparecen dificultades debidas a las distancias y desafíos presentes en el Desarrollo Global de Software, comentados anteriormente.

La gestión de conocimiento (en adelante KM, por sus siglas en inglés *Knowledge Management*) puede definirse como "*un proceso que permite crear, capturar, almacenar, buscar, recuperar, compartir, transferir y diseminar el conocimiento existente dentro de una organización, con el fin de incrementarlo y evitar su pérdida y sub-utilización*" [**?**] [**?**] [**?**].

En los siguientes apartados se comentan los desafíos encontrados en cada una de los cuatro procesos que componen la Gestión de Conocimiento dentro de una empresa, mostrados en la Figura 3.5.

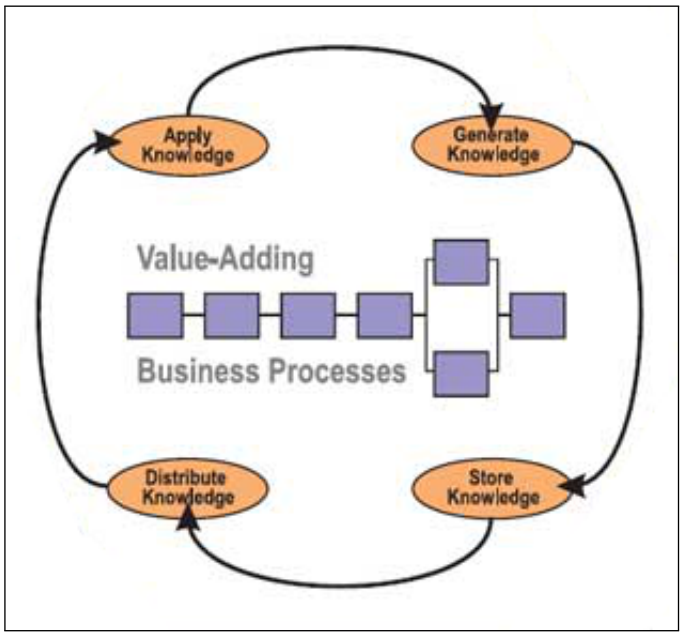

Figura 3.5: Ciclo de gestión de conocimiento [**?**]

3.1.2.3.1 Creación de conocimiento

La creación de conocimiento organizacional involucra crear nuevo conocimiento, o reemplazar ciertos contenidos con nuevo conocimiento tácito y explícito, obtenido tanto a través de colaboraciones e interacciones sociales, como a partir del propio proceso cognitivo de cada empleado [**?**].

En GSD, este proceso de creación de nuevo conocimiento se ve dificultado, ya que la información puede provenir de numerosas fuentes y lugares, con formatos y de tipos muy diferentes, en diferentes idiomas, etc. Además, las interacciones sociales se ven dificultadas por los desafíos de la comunicación, así como los procesos cognitivos de cada empleado, debido a las distancias socio-culturales. Por lo tanto, es difícil generar nueva información consistente y evitando duplicidades.

3.1.2.3.2 Almacenamiento y recuperación del conocimiento

Estrechamente ligado con el proceso anterior, se encuentra el almacenamiento y recuperación del conocimiento. Debido a que en GSD se manejan múltiples fuentes de conocimiento, y existen varias organizaciones que lo generan y utilizan, resulta complicado crear una base de conocimiento global y común para todas esas organizaciones que colaboran en GSD, debido a la multitud de información y formatos existentes (vídeos, documentos, procesos de negocio, etc.).

Para intentar mitigar los problemas que aparecen en estos procesos de KM, existe una tendencia a utilizar el enfoque de *Rationale*, o *Design Rationale*, para indicar qué tipo de información generar y almacenar, proveyendo así un tipo de información concreta, con un formato común y consistente que las empresas que colaboran en el desarrollo de proyectos software en GSD puedan crear, almacenar y recuperar, para su posterior aplicación. Este concepto de *Rationale* será presentado en la sección 3.2

3.1.2.3.3 Transmisión del conocimiento

El objetivo primordial de KM es que el conocimiento pueda ser utilizado y conocido por las empresas, para que puedan ser competitivas en el marco de globalización y sociedad de la información en el que nos encontramos.

La transmisión de conocimiento principalmente se produce por procesos de socialización, donde un grupo de individuos, normalmente con intereses comunes, comparten el conocimiento e intentan aprender y resolver problemas juntos [**?**].

En GSD, este proceso se ve dificultado debido a los desafíos que aparecen en la comunicación, ya que las herramientas de comunicación pueden no estar siempre disponibles y pueden impedir o dificultar esta transmisión de conocimiento. Además, debido a las diferencias socio-culturales, este proceso de compartición del conocimiento es más difícil de llevar a cabo, ya que dicho conocimiento puede ser interpretado de manera diferente en diferentes países, además de existir la barrera del idioma.

3.1.2.3.4 Aplicación del conocimiento

A la hora de aplicar el conocimiento recuperado, también existen dificultades, debido a todos los problemas que ya se han comentado, destacando la falta de consenso a la hora de aplicar dicho conocimiento, debido a las diferentes formas de gestionar y usar el conocimiento en las diversas organizaciones que colaboran en GSD, ya que cada una tendrá sus propios procesos de negocio y gestión.

3.2 *Design Rationale*

Design Rationale (o *Rationale* simplemente) es "*un método que permite capturar, representar y mantener registros de información acerca de las decisiones que son tomadas por los miembros de un equipo de desarrollo de un proyecto software*" [**?**]. Dicho método puede ser aplicado en cualquier fase del ciclo de vida del desarrollo, desde el análisis de requisitos hasta las pruebas.

De este modo, se pueden capturar todas las decisiones y alternativas que se van produciendo a lo largo del desarrollo del proyecto, y las razones de por qué son tomadas esas decisiones. Así, *Rationale* se centra en capturar [**?**]:

- Decisiones tomadas.
- Las razones detrás de la decisión tomada.
- Argumentos a favor o en contra de las decisiones tomadas.
- Decisiones evaluadas y su resultado.

Este enfoque de *Rationale* se plantea como una posible solución para mitigar los desafíos que aparecen en la gestión de conocimiento en GSD, comentados anteriormente. Gracias a esto, el conocimiento que en este caso interesa generar, almacenar, transmitir, reutilizar y aplicar son las decisiones tomadas en el desarrollo de un proyecto software. Por tanto, las diferentes organizaciones que participan en el desarrollo global de proyectos software utilizarán un mismo tipo de información, común y consistente.

De este modo, *Rationale* proporciona las siguientes ventajas a la hora de aplicarse en GSD, concretamente en la gestión de conocimiento:

- Proporciona un mecanismo común a las organizaciones para la captura y almacenamiento de las decisiones tomadas en el desarrollo de proyectos software, ya que se sigue el mismo formato y se define el tipo de información a generar.
- Facilita la representación de las decisiones y, por tanto, su transmisión, mejorando la calidad de futuras decisiones y la comunicación entre equipos de desarrollo.
- Facilita la recuperación de las decisiones, al estar bien definido el tipo de información que se ha almacenado. Del mismo modo, se facilita también su reutilización.

En lo que respecta a la captura de decisiones, en *Rationale* existen métodos muy variados. Sin embargo, algunos de los métodos que pueden emplearse para la captura de decisiones son [**?**] [**?**]:

- **Reconstrucción**: captura la información de una forma *cruda*, sin procesar, y luego la reconstruye de una manera más estructurada.
- **Método "Record and replay"**: las decisiones son capturadas de manera síncrona utilizando vídeo-conferencias, o de manera asíncrona, a través de discusiones por e-mail.
- **Método del "Aprendiz"**: se capturan las decisiones haciendo preguntas lógicas que coinciden, o no, con el punto de vista del diseñador.
- **Generación automática**: las decisiones se generan automáticamente a través de un historial. Este método tiene la capacidad para mantener las decisiones consistentes y actualizadas.
- **Método del "Historiador"**: es un método por el cual una persona u ordenador observa todas las acciones de los diseñadores, pero sin hacer ninguna sugerencia, y después describe todas esas acciones observadas.

En lo que respecta a representación del conocimiento, existen, principalmente, dos enfoques [**?**]:

- **"Causal Graph**": es un grafo causal donde cada nodo representa una decisión y los arcos representan restricciones entre ellas, como por ejemplo que una decisión depende

o proviene de otra anterior. En la Figura 3.6 puede observase un ejemplo de este tipo de grafo.

- **"Dialogue Map"**: es un grafo donde los nodos pueden representar una pregunta, una idea o un argumento a favor o en contra de dichas ideas y preguntas. De este modo, se puede ir discutiendo una decisión e ir añadiendo nuevas alternativas y justificaciones. En la Figura 3.7 puede observarse un ejemplo. Este tipo de grafo se basa en los sistemas **IBIS** (**I**ssue-**B**ased **I**nformation **S**ystems, o Sistemas de Información Basados en Preguntas), donde el conocimiento se almacena y representa de manera jerárquica, en forma de decisiones/preguntas y justificaciones de éstas. Este será el método de representación utilizado en la aplicación a desarrollar en el presente PFC.

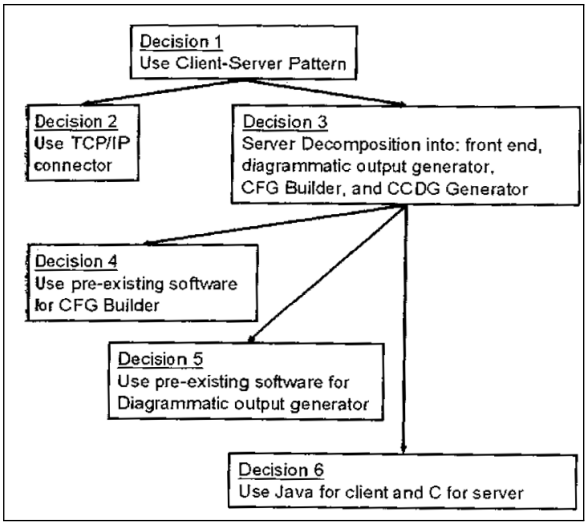

Figura 3.6: "Causal Graph" utilizado para representar decisiones en *Rationale* [**?**]

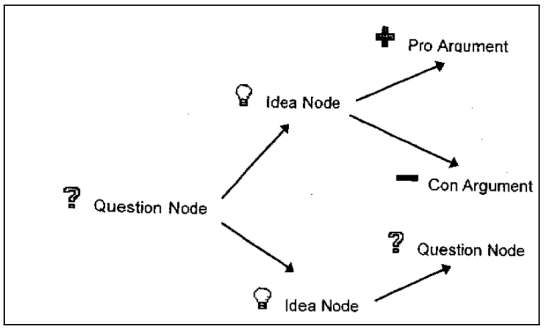

Figura 3.7: "Dialogue Map" utilizado para representar decisiones en *Rationale* [**?**]

3.3 Razonamiento basado en casos

Estrechamente relacionado con la gestión de conocimiento, está el concepto de **Razonamiento Basado en Casos** (en adelante CBR, por sus siglas en inglés Case-Based Reasoning), utilizado para la recuperación de conocimiento, o experiencias previas, y su aplicación y reutilización en nuevos casos. Por tanto, el CBR puede plantearse como una solución para mitigar los problemas de recuperación y aplicación del conocimiento en GSD, comentados con anterioridad. Además, en este caso, el proceso de CBR se simplifica ya que, gracias al uso de *Rationale*, el formato y tipo del conocimiento a recuperar y reutilizar es el mismo: decisiones tomadas en proyectos software.

El CBR puede definirse como "*un proceso en el que experiencias específicas son recuperadas, reutilizadas, revisadas y almacenadas para utilizarse en la solución de problemas similares*" [**?**]. De este modo, se pueden encontrar soluciones a nuevos problemas basándose en soluciones de problemas similares anteriores. Dichas soluciones puede que sea necesario adaptarlas para resolver el nuevo problema y se almacenarán en forma de nuevas experiencias (casos) para poder reutilizarlas en un futuro. Es por esto que un sistema CBR puede "aprender" a partir de nuevas soluciones [**?**].

A diferencia de otras técnicas y algoritmos de Inteligencia Artificial, como pueden ser Razonamiento Basado en Reglas o Algoritmos Genéticos, CBR no se considera como una tecnología, sino más bien como una metodología, que indica cómo resolver problemas a partir de soluciones previas almacenadas en el sistema, pero sin detallar una tecnología concreta [**?**].

Conceptualmente, el CBR se describe como un ciclo con cuatro grandes etapas [**?**] [**?**]:

- **Recuperación (*Retrieve*)** de casos similares al problema planteado.
- **Reutilización (*Reuse*)** de una solución propuesta por un caso similar.
- **Revisión (*Revise*)** de la solución propuesta, para adaptarla mejor a las condiciones del nuevo problema.
- **Retención (*Retain*)** de la nueva solución, pasando a formar un nuevo caso.

Estas cuatro etapas son las que constituyen la metodología del CBR. Así, para solucionar un nuevo problema, primero se debe obtener una descripción de éste, midiendo la similitud

del nuevo problema con otros problemas previos almacenados en el sistema. A continuación, se recuperan las soluciones de estos problemas similares y se reutiliza la solución de uno de estos casos, adaptándola si es necesario. Para terminar, este nuevo problema, junto a la solución encontrada, se almacena en el sistema, formando un nuevo caso.

Uno de los puntos críticos en el CBR es la función de similitud que se va a utilizar para buscar casos similares a uno dado. Por ejemplo, un problema se podría describir como una serie de atributos (o características) que pueden ser cuantificadas con un "peso" numérico, obteniendo un vector numérico, y utilizar como función de similitud la distancia Euclídea entre los vectores que representan cada problema.

Capítulo 4

Método de Trabajo

Para el desarrollo de este producto software se ha optado por utilizar la metodología genérica descrita por el **Proceso Unificado de Desarrollo** (en adelante PUD), propuesta por Rumbaugh, Booch y Jacobson [**?**].

Para el modelado de los diagramas del producto software se utilizará el Lenguaje Unificado de Modelado (UML, por sus siglas en inglés Unified Modelling Language).

4.1 Proceso Unificado de Desarrollo

El Proceso Unificado de Desarrollo es una evolución del Proceso Unificado de Rational (RUP), que define un "*conjunto de actividades necesarias para transformar los requisitos de usuario en un sistema software*" [**?**]. Más concretamente se puede definir como "*un marco de trabajo marco genérico que puede especializarse para una gran variedad de sistemas de software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyectos.*" [**?**].

Las principales características del PUD son:

- **Dirigido por casos de uso**. Un caso de uso representa un requisito funcional al cuál el sistema debe dar soporte para proporcionar un resultado de valor al usuario. Los casos de uso guían el proceso de desarrollo, ya que basándose en los casos de uso, los desarrolladores crean una serie de modelos para poder llevarlos a cabo. Todos los casos de uso juntos constituyen el **modelo de casos de uso**.

- **Centrado en la arquitectura**. Un sistema software puede contemplarse desde varios

puntos de vista. Por tanto, la arquitectura software incluye los aspectos estáticos y dinámicos más significativos del sistema y debe estar profundamente relacionada con los casos de uso, ya que debe permitir el desarrollo de los mismos.

- **Iterativo e incremental**. El esfuerzo de desarrollar un proyecto de software se divide en partes más pequeñas, llamadas **mini-proyectos**. Cada mini-proyecto es una **iteración**, compuesta por una serie de requisitos funcionales representados por casos de uso, y que abarcan entre dos y seis semanas de duración. Las iteraciones deben estar controladas y deben seleccionarse y ejecutarse de una forma planificada, siguiendo el esquema *requisitos, análisis, diseño, implementación y pruebas*, que es conocido como **flujo de trabajo**. En cada iteración, los desarrolladores identifican y especifican los casos de uso relevantes, crean un diseño utilizando la arquitectura seleccionada como guía, implementan el diseño mediante componentes y verifican que los componentes satisfacen los casos de uso. Finalmente, cada iteración aporta un **incremento** en la funcionalidad del sistema, por eso que el PUD se considere un proceso incremental (ver Figura 4.1) [**?**].

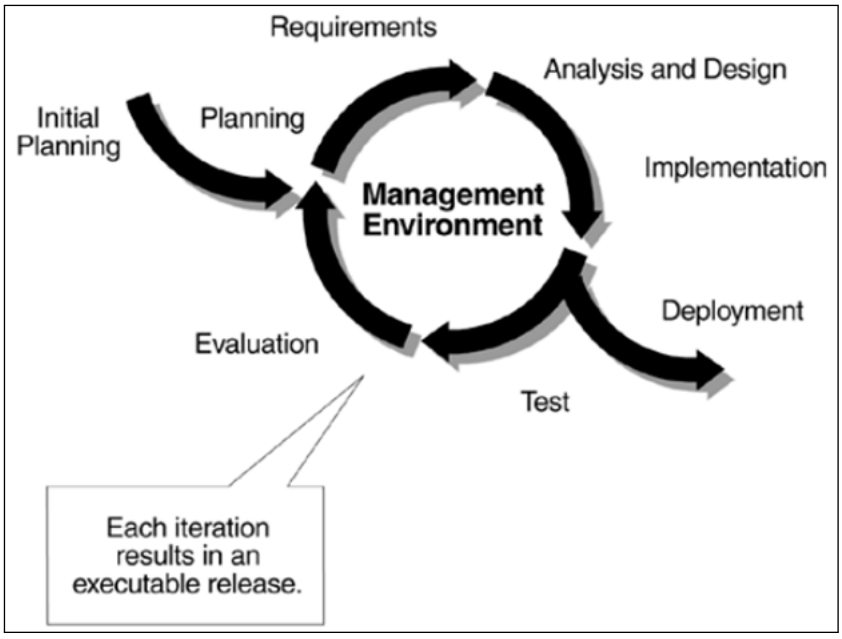

Figura 4.1: Proceso iterativo e incremental

4.1.1 Fases de desarrollo basado en PUD

En PUD, el ciclo de vida de un producto software se divide en ciclos, donde cada uno de estos ciclos compone una versión del producto, totalmente operativa y preparada para su implantación. Cada ciclo se compone de cuatro fases (Inicio, Elaboración, Construcción y Transición) y éstas, a su vez, se dividen en iteraciones que siguen el flujo de trabajo *requisitos, análisis, diseño, implementación y pruebas* (ver Figura 4.2).

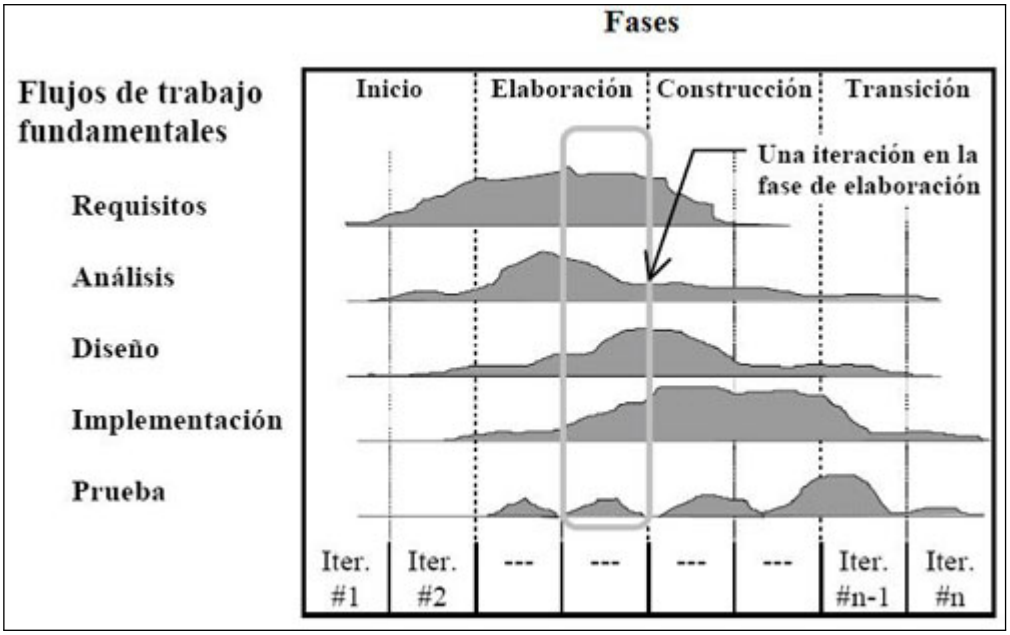

Figura 4.2: Fases y flujo de trabajo del PUD

A continuación se detallan cada una de las cuatro fases de cada ciclo del PUD [**?**] [**?**].

4.1.1.1 Fase de Inicio

En esta fase es donde se determina el alcance del proyecto, su viabilidad, riesgos potenciales y donde se realiza una planificación del proyecto. Normalmente, esta fase abarca una única iteración y los artefactos más relevantes utilizados son:

- **Modelo de Casos de Uso**: se obtiene un primer modelo de casos de uso simplificado, representando los requisitos funcionales del sistema.
- **Descripción de Riesgos**: se obtiene un documento que recoge posibles riesgos que pueden aparecer y afectar al ciclo de vida del producto.

- **Glosario**: se elabora un glosario de términos del dominio de aplicación.
- **Plan de Proyecto**: se elabora un plan de iteraciones a seguir para el desarrollo del producto software.

4.1.1.2 Fase de Elaboración

En esta fase, compuesta de una una o más iteraciones, se especifican en detalle la mayoría de los casos de uso identificados en la fase de inicio y se diseña la arquitectura del sistema, obteniendo la línea base de la arquitectura. Por arquitectura se entiende el "*conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, las interfaces entre ellos, su comportamiento, sus colaboraciones, y su composición*" [**?**].

En esta fase, se construyen los siguientes artefactos:

- **Arquitectura**: se obtiene el diseño de la arquitectura del sistema.
- **Modelo de Casos de Uso**: se obtiene un modelo de casos más detallados y con todos los requisitos funcionales.
- **Modelo de Análisis**: es un modelo compuesto de diagramas de clases de análisis y de secuencia, más detallado que el de casos de usos pero menos que el de diseño.
- **Modelo de Diseño**: es un modelo compuesto de diagramas de clases de diseño que describen el funcionamiento del sistema.

4.1.1.3 Fase de Construcción

En la fase de construcción es donde se lleva a cabo la implementación de cada una de las iteraciones en las que se ha dividido el desarrollo del producto software, a partir de los artefactos generados en la fase de elaboración. Aunque esta fase se centra en la implementación, puede haber iteraciones donde también haya tareas de requisitos, análisis y diseño, refinando modelos de etapas anteriores, pero éstas serán prácticamente nulas en iteraciones avanzadas de esta fase. Cada iteración se cierra con las pruebas realizadas al código implementado.

Esta fase abarca un número muy variable de iteraciones y suele ser la fase de más larga duración en el ciclo de vida del producto software. En esta fase se obtienen los siguientes artefactos:

- **Modelo de Diseño**: a partir del modelo de diseño de la fase anterior, se obtiene otro modelo refinado durante iteraciones de esta fase.
- **Modelo de Implementación**: es un modelo compuesto por diagramas de componentes, junto a sus relaciones y dependencias.
- **Modelo de Pruebas**: es el conjunto de casos de pruebas unitarias que cierran cada iteración de esta fase.
- **Modelo de Despliegue**: modelo que refleja los aspectos físicos y como se ejecutarán los componentes identificados en el modelo de implementación.

4.1.1.4 Fase de Transición

La fase de transición es el período en el cual el producto se convierte en una versión beta, es decir, se procede a su implantación pero se seguirá probando y, quizás, incrementando su funcionalidad. En esta fase se obtienen los siguientes artefactos:

- **Modelo de Despliegue**: se obtiene el modelo final a partir del de la fase anterior.
- **Modelo de Distribución**: es un modelo que muestra el funcionamiento físico del sistema.
- **Manuales**: el sistema se documenta y se generan los manuales de usuario para explicar su funcionamiento.

4.1.2 Flujo de trabajo en el PUD

Como se ha comentado en la sección 4.1.1, en cada una de las fases desarrolladas anteriormente se sigue un flujo de trabajo, compuesto por las disciplinas de captura de requisitos, análisis, diseño, implementación y pruebas.

Según en la fase e iteración en la que se encuentre el desarrollo, estas disciplinas serán más o menos relevantes en dicha iteración dentro de esa fase (ver Figura 4.2).

En los siguientes apartados se comentan las actividades a realizar en cada una de las disciplinas que forman el flujo de trabajo de cada fase del PUD [**?**] [**?**].

4.1.2.1 Captura de requisitos

Esta disciplina tiene una mayor relevancia en las fases de Inicio y Elaboración y tiene como objetivo el identificar los requisitos del producto para guiar su posterior desarrollo. Se realizan las siguientes actividades:

- Se identifican los requisitos funcionales del sistema a desarrollar.
- Se identifica el contexto o dominio de aplicación, para comprender los requisitos.
- Se identifican los requisitos no funcionales del sistema, como comunicación con otros sistemas, entorno de funcionamiento, etc.
- Los requisitos identificados se modelan utilizando un diagrama de casos de uso, que será más o menos detallado según la fase del PUD en la que nos encontremos.

4.1.2.2 Análisis

Esta disciplina tiene una mayor relevancia en la fase de Elaboración y tiene como objetivo la especificación detallada de los casos de uso obtenidos anteriormente, refinándolos con posibles nuevos requisitos. Se realizan las siguientes actividades:

- Se detallan los casos de uso obtenidos en fases anteriores.
- Se realiza el diagrama de clases de análisis.
- Se identifican nuevos requisitos que puedan surgir al refinar los casos de uso y generar los diagramas de análisis.

4.1.2.3 Diseño

Esta disciplina tiene una mayor relevancia en las fases de Elaboración y Construcción y tiene como objetivo el modelado del sistema y de su arquitectura para soportar los requisitos, tanto funcionales como no funcionales. Se realizan las siguientes actividades:

- Se realiza el modelo de diseño, con diagramas de clases detallados y las relaciones entre clases.
- Se identifica la arquitectura del sistema.
- Se realiza el modelo de despliegue.

4.1.2.4 Implementación

Esta disciplina tiene una mayor relevancia en la fase de Construcción y tiene como objetivo generar la herramienta software. Se realizan las siguientes actividades:

- Se realiza el modelo de implementación.
- Se detalla y refina la arquitectura del sistema.
- Se detalla y refina el modelo de despliegue.
- Se codifican todos los requisitos del sistema, obteniendo código ejecutable que le da soporte a dichos requisitos.

4.1.2.5 Pruebas

Esta disciplina afecta a todas las iteraciones y fases donde se haya producido una implementación, ya que es necesario probar que dicha codificación funciona correctamente. Su objetivo es crear los casos de prueba para cada iteración y crear un plan de pruebas del sistema global. Se realizan las siguientes actividades:

- Se realiza el modelo de pruebas, creando casos de prueba unitarios, de integración y de sistema.

4.1.3 Resumen del Proceso Unificado de Desarrollo

El PUD está compuesto de ciclos, donde cada ciclo consta de cuatro fases y cada fase, a su vez, se divide en una o varias iteraciones, donde se aplica un flujo de trabajo compuesto por cinco disciplinas. En la Tabla 4.1 se resumen las cuatro fases de un ciclo presentes en el PUD.

Fase	Resumen	Artefactos generados
Inicio	Se identifican los requisitos Se crea un glosario de términos Se analiza la viabilidad del proyecto Se realiza el plan de iteraciones	Modelo de casos de uso Glosario de términos Descripción de riesgos Plan de proyecto
Elaboración	Se crea un modelo de casos de uso detallado Se diseña la arquitectura Se crea un modelo de clases de análisis Se realiza el modelo de diseño	Modelo de casos de uso detallado Modelo de Diseño Arquitectura del sistema Modelo de Análisis
Construcción	Se implementan los requisitos del sistema Se definen pruebas Se crea un modelo de despliegue	Modelo de implementación Modelo de pruebas Modelo de despliegue Ejecutables del sistema
Transición	Se escribe la documentación y manuales de usuario Se explica como instalar e implantar el sistema	Documentación Manuales de usuario Modelo de distribución

Tabla 4.1: Resumen de las fases de un ciclo del PUD

En el capítulo 5 se expondrán los resultados obtenidos al aplicar PUD en el desarrollo del presente PFC, desarrollando cada una de las fases y qué artefactos se han ido obteniendo en cada una de ellas hasta llegar al sistema final.

4.2 Marco tecnológico de trabajo

En esta sección se expondrán las herramientas, tecnologías y librerías utilizadas durante todo el ciclo de vida del proyecto.

4.2.1 Herramientas para la gestión del proyecto

4.2.1.1 Subversion

Subversion es un sistema de control de versiones donde, a diferencia de CVS (Concurrent Versions System), los archivos versionados del repositorio no tienen cada uno un número de versión distinto, sino que todo el repositorio tiene un único numero de revisión. Además, Subversion permite acceder a repositorios de red.

En el proyecto, se utilizará un repositorio Subversión (o SVN) para gestionar las diferentes versiones del mismo. El repositorio SVN estará alojado en *Google Code*, de manera que será accesible en red. Para acceder a dicho repositorio, se utilizará la herramienta **TortoiseSVN**, que es un cliente de Subversion implementado como una extensión del shell de sistemas Windows y que permite gestionar el repositorio. En la Figura 4.3 se puede observar esta herramienta.

4.2.1.2 Maven

Maven es una herramienta de gestión y construcción de proyectos, creada por Jason van Zyl en 2002. Maven permite describir el proyecto a construir, sus dependencias con otros módulos y el orden de construcción utilizando un Modelo de Objetos del Proyecto (POM, por sus siglas del inglés Project Object Model), que es un fichero XML donde se describe el proyecto, sus dependencias, etc.

Maven tiene una arquitectura dinámica basada en *plugins* que son descargables desde su repositorio central, de tal modo que permite la reutilización para diferentes proyectos, cambiando los plugins a utilizar y su configuración y manteniendo patrones similares en diferentes proyectos.

Se utilizará esta herramienta, en su versión 2.2.1, para la gestión del proyecto y sus

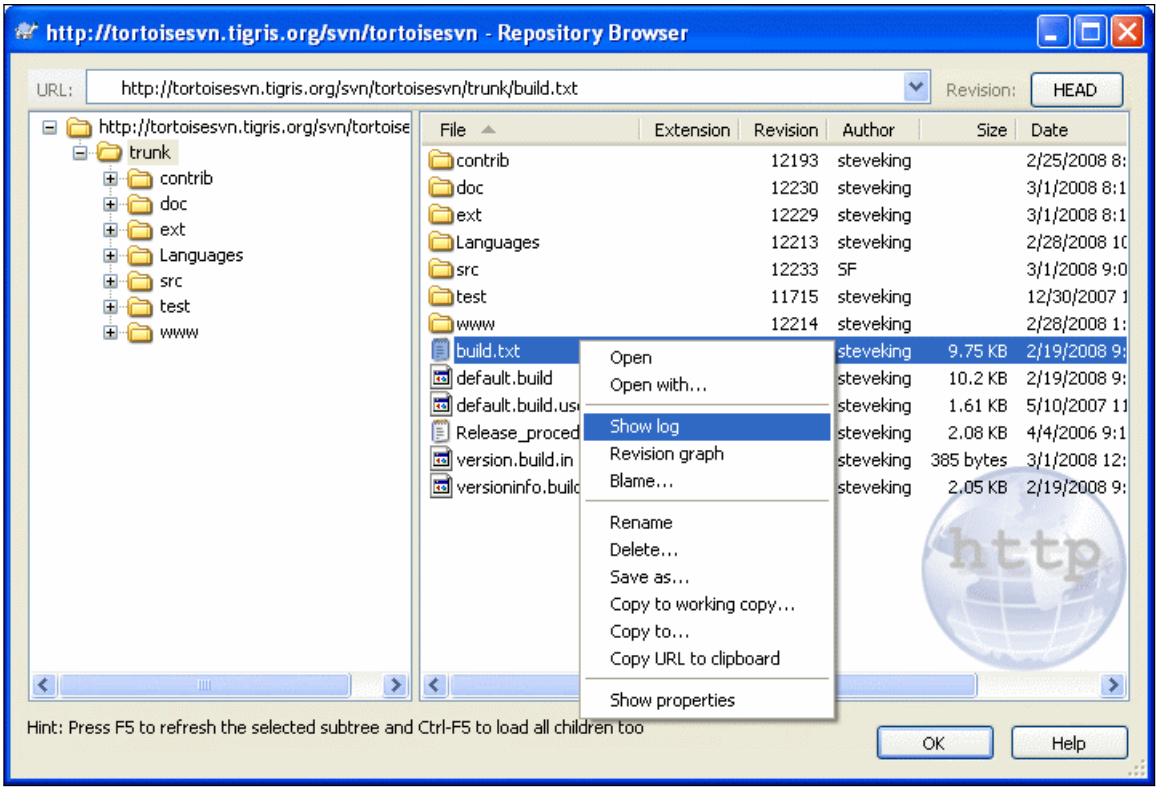

Figura 4.3: Herramienta *TortoiseSVN*

dependencias con otras librerías y módulos, usando el plugin de Maven para Eclipse llamado **m2eclipse**. En la Figura 4.4 se puede observar la interfaz de este plugin sobre Eclipse.

4.2.2 Herramienta para el modelado

4.2.2.1 Visual Paradigm

Visual Paradigm es una herramienta CASE (Computer Aided Software Engineering o Ingeniería de Software Asistida por Computador) profesional para el modelado UML (Unified Modeling Language o Lenguaje Unificado de Modelado) que soporta el ciclo de vida del desarrollo de un producto software. Permite crear diferentes tipos de diagramas y también permite la ingeniería inversa, es decir, a partir de código, generar diferentes diagramas.

Esta herramienta, en su versión 8.0, se utilizará para realizar los diagramas de casos de uso, de secuencia, de clases, etc. En la Figura 4.5 se puede observar la interfaz de esta herramienta.

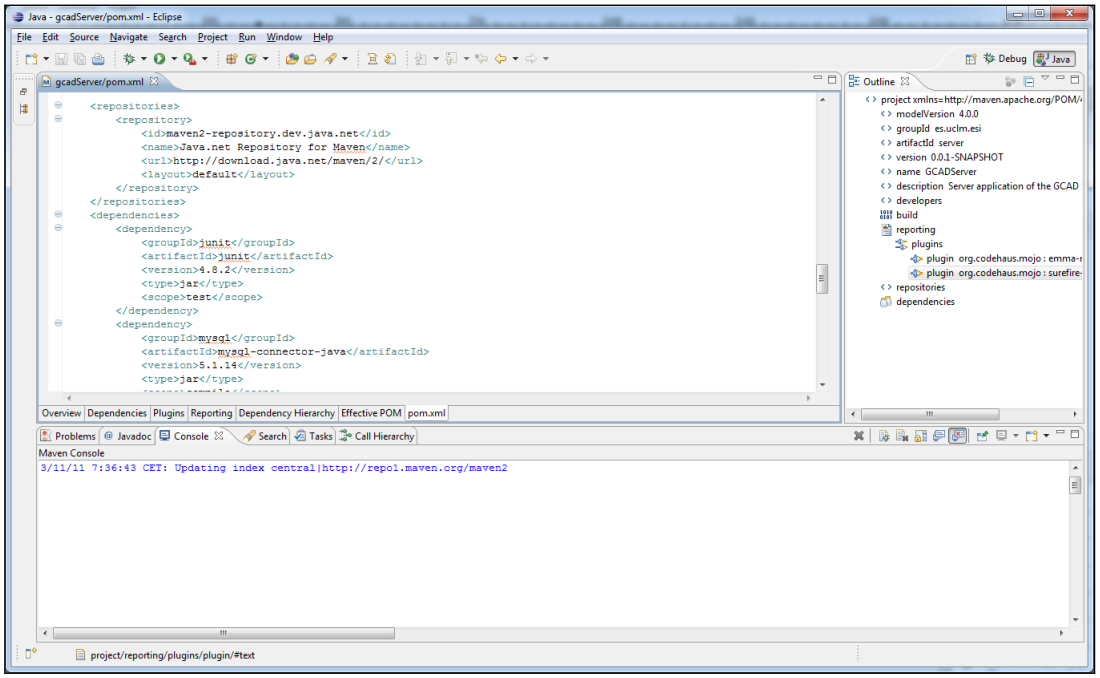

Figura 4.4: Plugin de Maven sobre Eclipse Helios

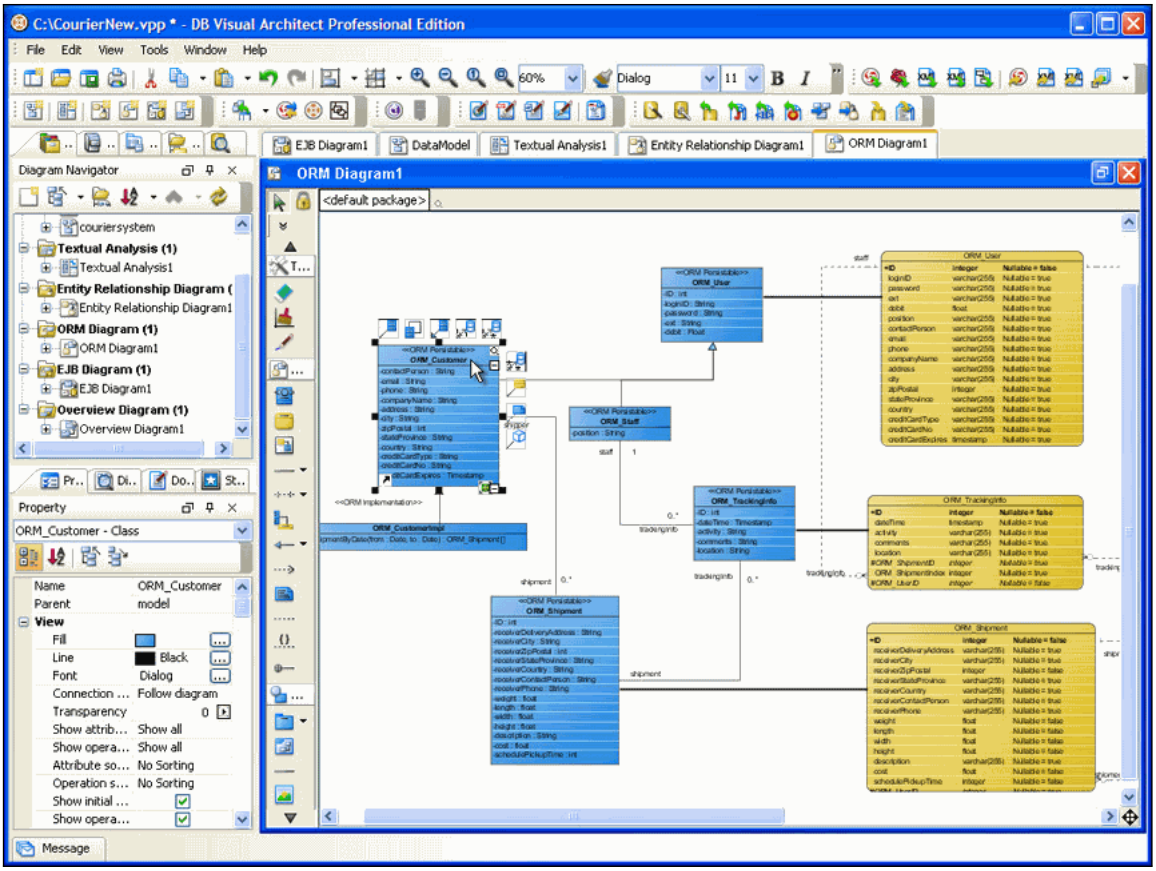

Figura 4.5: Herramienta Visual Paradigm

4.2.3 Herramientas y tecnologías para el desarrollo del proyecto

4.2.3.1 Eclipse Helios

Eclipse es un IDE (Integrated Development Environment o Entorno Integrado de Desarrollo) multiplataforma, de código abierto y basado en plugins para aumentar su funcionalidad.

Es el entorno de desarrollo utilizado para la codificación de proyecto utilizando el lenguaje de programación Java, junto a plugins utilizados para la gestión de proyectos con Maven (plugin **m2eclipse**) y diseño de interfaces gráficas con Swing (plugin **Jigloo**). En la Figura 4.6 se muestra una captura de pantalla de este IDE.

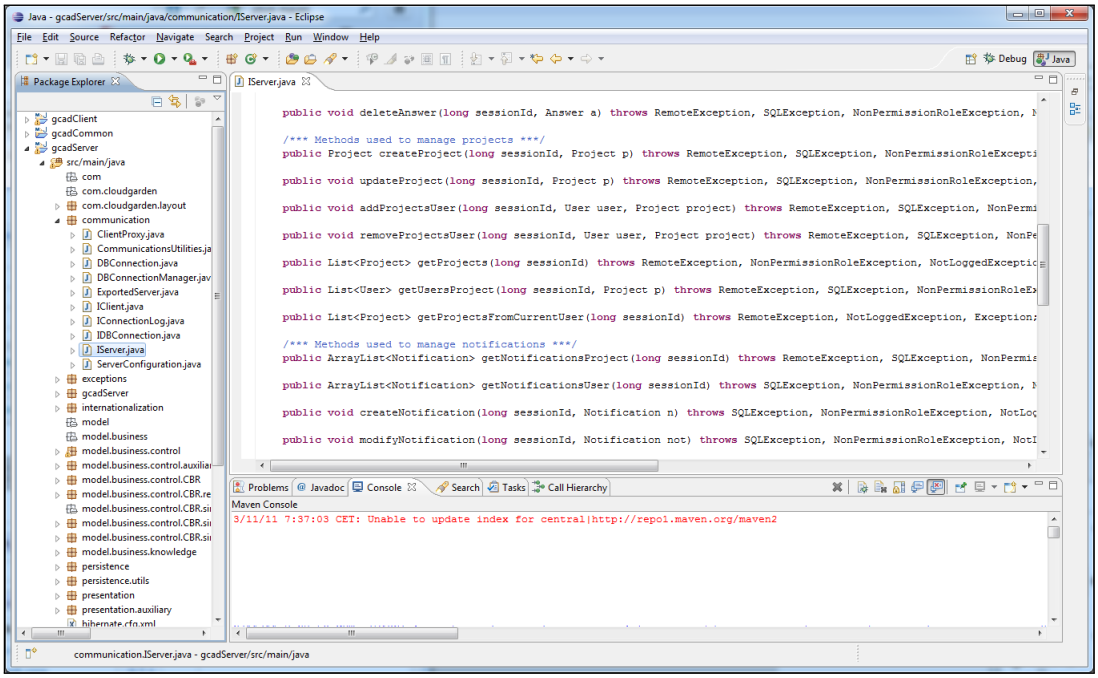

Figura 4.6: Eclipse Helios

4.2.3.2 Yahoo! PlaceFinder

Yahoo! PlaceFinder es un servicio web de geocodificación que permite la conversión de direcciones de calles o nombres de lugares en coordenadas geográficas, y viceversa. La respuesta a una petición de este servicio Web viene dada por un fichero XML, como el mostrado en la Figura 4.7.

Se utilizará para obtener y localizar en mapas las coordenadas geográficas a partir de

direcciones postales.

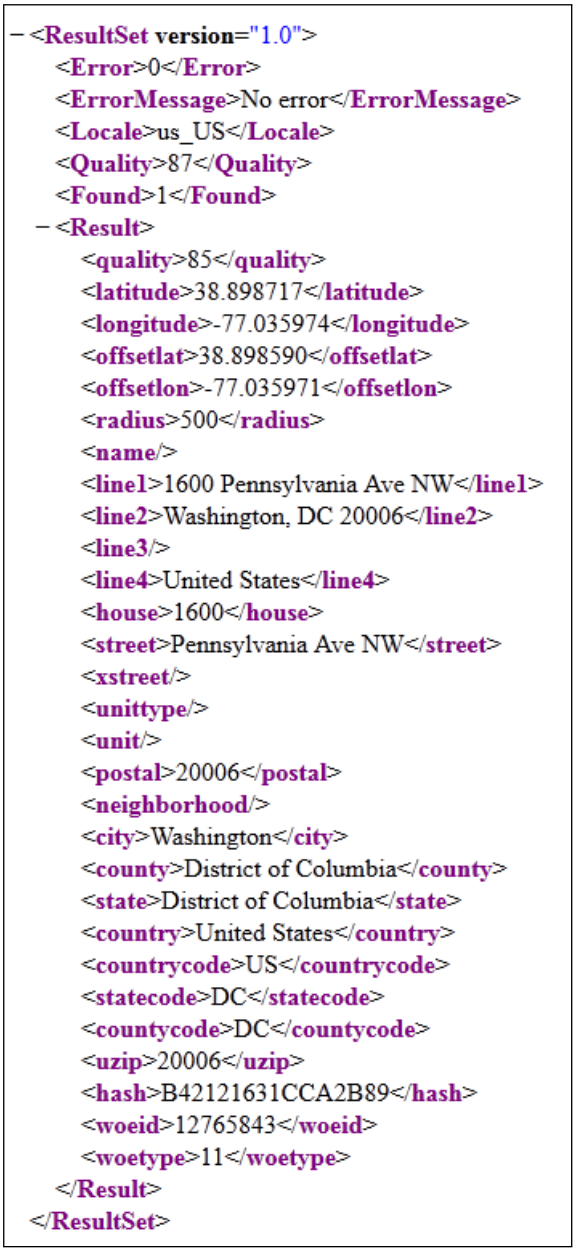

Figura 4.7: Respuesta XML del servicio Web Yahoo! PlaceFinder

4.2.3.3 OpenStreetMap

OpenStreetMap, conocido también como OSM, es un proyecto colaborativo para crear mapas del mundo libres y editables. Los mapas se crean utilizando GPS para capturar información geográfica, a través de ortofotografías y a través de otras fuentes libres. En la Figura 4.8 se puede apreciar un ejemplo de mapa obtenido con OpenStreetMap.

Se utilizará para mostrar mapas y poder localizar posiciones geográficas en ellos, usando el servicio Web comentado en el apartado 4.2.3.2.

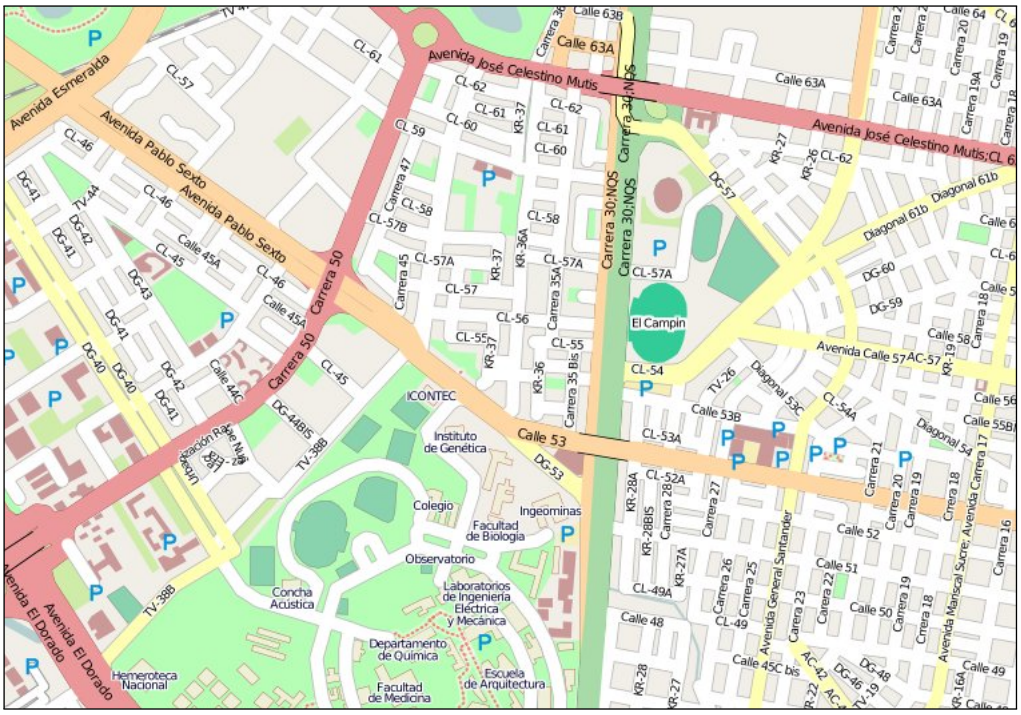

Figura 4.8: Ejemplo de mapa de OpenStreetMap

4.2.3.4 JAXB

Java Architecture for XML Binding es una tecnología de Java que permite la transformación de una jerarquía de objetos Java a un fichero XML, y viceversa. El proceso de serialización recibe el nombre de *marshalling*, mientras que el proceso inverso recibe el nombre de *unmarshalling* (ver Figura 4.9).

Se utilizará, en su versión 2.2.4, para poder convertir objetos del dominio de aplicación en ficheros XML, y viceversa.

4.2.3.5 JDOM

JDOM es una librería de Java de código abierto para el acceso y manipulación de ficheros XML optimizada para Java. Se creó para ser específicamente utilizado con Java, por lo que incluye sobrecarga de métodos, colecciones, etc, a diferencia del DOM creado por el

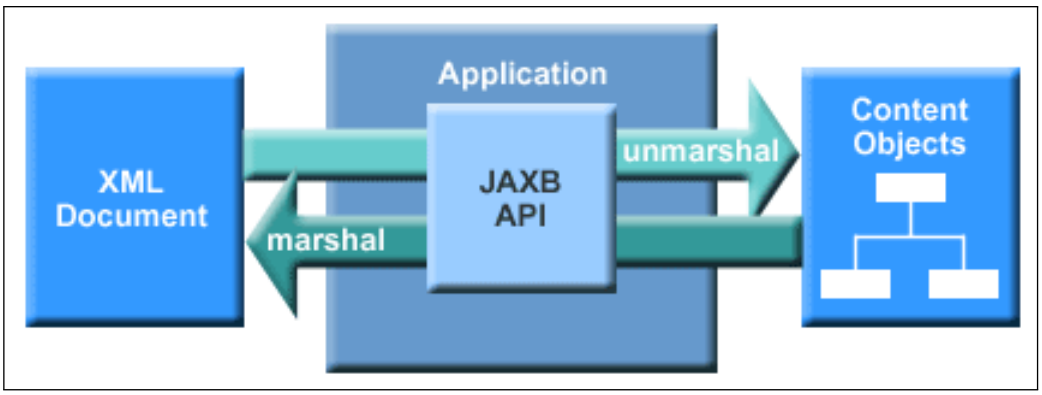

Figura 4.9: JAXB

consorcio W3C (World Wide Web), creado inicialmente para manipulación de páginas HTML con JavaScript. Señalar que JDOM no es un acrónimo de *Java Document Object Model*.

Dicha librería, en su versión 1.1, se utilizará para gestionar ficheros XML presentes en la aplicación, como, por ejemplo, los ficheros de perfiles.

4.2.3.6 Jaxen

Jaxen es una librería de Java de código abierto escrita en Java, adaptable a diferentes modelos de objetos como DOM, XOM, dom4j y JDOM, y que permite trabajar con expresiones *XPath* (XML Path Language).

Se utilizará en su versión 1.1.1 para la búsqueda de nodos en ficheros XML utilizando expresiones *XPath*.

4.2.3.7 Hibernate

Hibernate es un framework para el mapeo objeto-relacional (ORM, por sus siglas en inglés Object Relational-Mapping) utilizado en plataformas Java y que permite realizar el mapeo de atributos entre una base de datos relacional y un modelo de objetos, utilizando archivos XML o anotaciones sobre los objetos para realizar dicho mapeo. Proporciona también un lenguaje de consultas de bases de datos de alto nivel, llamado HQL (Hibernate Query Language).

Las ventajas de utilizar este framework es que permite al desarrollador abstraerse del código que hay que escribir para dar soporte a la persistencia, facilita la mantenibilidad,

proporciona independencia del proveedor del sistema gestor de base de datos y flexibilidad para adaptarse al esquema de tablas utilizado.

Se utilizará Hibernate, en su versión 3.5.4, para gestionar la persistencia de los objetos de dominio que deban ser persistentes en el sistema.

4.2.3.8 Swing

Swing es una librería gráfica para Java, la cual incluye *widgets* para generar interfaces gráficas de usuario, como son cajas de texto, botones, paneles, tablas, etc. Swing fue desarrollado para proveer un conjunto de elementos gráficos más sofisticado que su antecesor, AWT (Abstract Window Toolkit), de tal modo que los componentes Swing están escritos en Java y son independientes de la plataforma, a diferencia de AWT.

Swing sigue el modelo vista controlador (MVC), es independiente de la plataforma y extensible, ya que los componentes puede extenderse y sobrescribir las implementaciones por defecto.

Existe una extensión a Swing, llamada SwingX, que incluye nuevos componentes gráficos que normalmente se demandan en aplicaciones avanzadas (Rich Client Applications), como son selectores de fechas, framework para la autenticación de usuarios, auto-completado, paneles expandibles, etc.

Por otra parte, existe un framework para Swing, llamado Swing Application Framework, que forma parte de la JSR 296 (Java Specification Request) y que permite gestionar la arquitectura, ciclo de vida, hilos, manejo de eventos y almacenamiento del estado de una aplicación Swing.

Estos tres componentes, Swing, SwingX y Swing Application Framework se utilizarán para el desarrollo de una interfaz gráfica de usuario extensible, personalizable y adaptable. Además, para realizar el diseño de la interfaz, se uitlizará el plugin para eclipse **Jigloo**.

4.2.3.9 JUNG

Jung (*Java Universal Network/Graph Framework*) es una librería que provee mecanismos para el modelado, análisis y visualización de datos que pueden ser representados en una red o en un grafo. Su arquitectura está diseñada para soportar variedad de representaciones

de entidades y sus relaciones, como grafos dirigidos, no dirigidos, hipergrafos, grafos con arcos paralelos, entidades con metadatos, etc. Además, incluye numerosos algoritmos para la visualización de grafos, para su optimización, cálculo de distancias, métricas, etc. En la Figura 4.10 se puede observar un grafo generado con este framework.

Se utilizará, en su versión 2.0, para la creación y visualización de grafos.

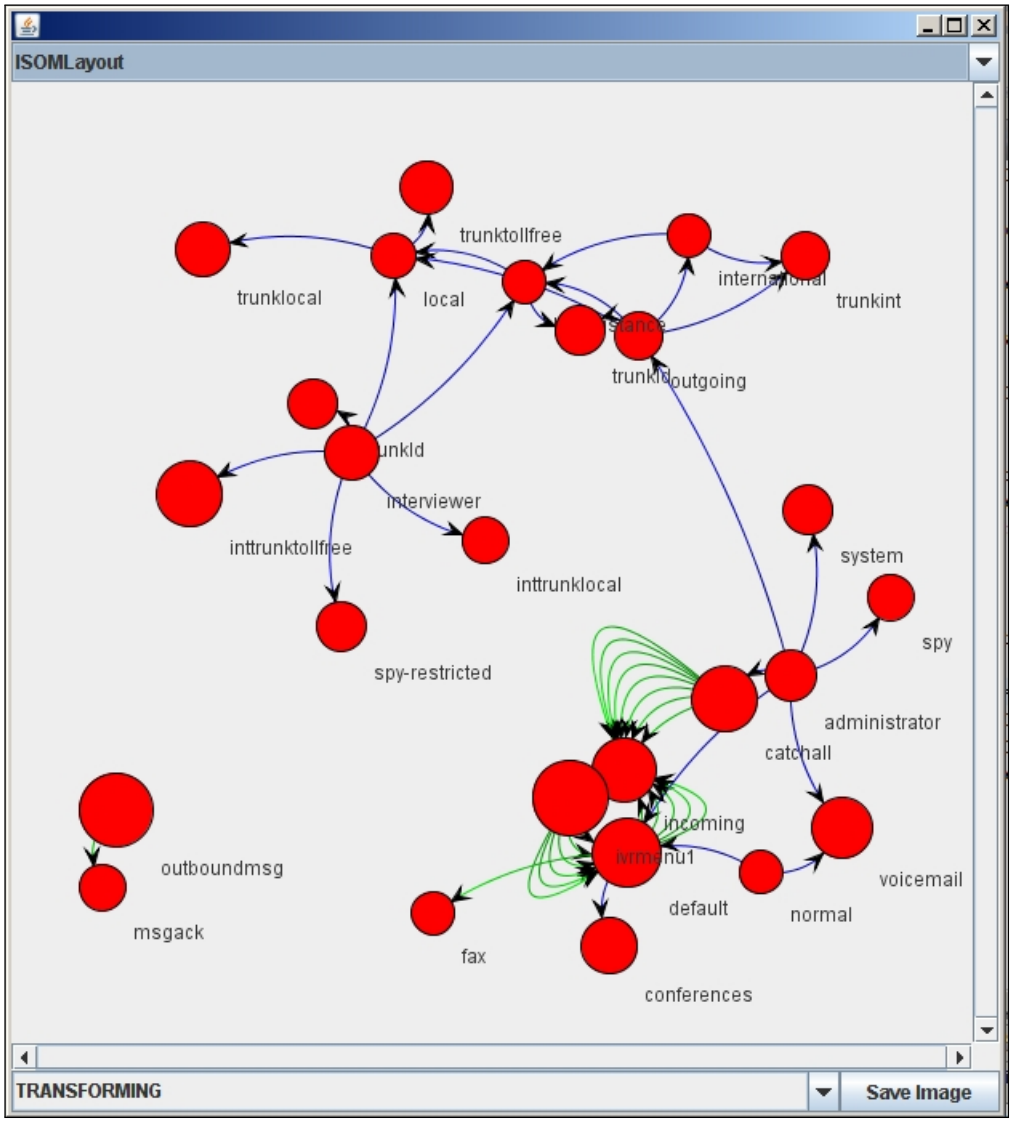

Figura 4.10: Ejemplo de grafo generado con JUNG

4.2.3.10 iText

iText es una librería que permite la creación y manipulación de documentos PDF. Dicha herramienta permite:

- Servir documentos PDF a un navegador.
- Generar documentos PDF a través de archivos XML o bases de datos.
- Añadir marcadores, número de páginas, marcas de agua, etc.
- Dividir, concatenar y gestionar páginas PDF

Se utilizará, en su versión 5.1.0 para Java (disponible también para .NET) para la generación de documentos PDF.

4.2.3.11 JFreeChart

JFreeChart es una librería escrita en Java que permite la creación y visualización de gráficos. En la Figura 4.11 se puede observar ejemplos de gráficos generados con esta librería. Las características de esta librería son:

- Proporciona un API para generar una amplia variedad de gráficos.
- Diseño flexible y extensible.
- Soporte para diferentes tipos de salida, como componentes de Swing, imágenes (PNG y JPEG) y gráfixos vectoriales (EPS y SVG)
- Es *OpenSource*

Se ha utilizado en su versión 1.0.13 para la generación y visualización de gráficos.

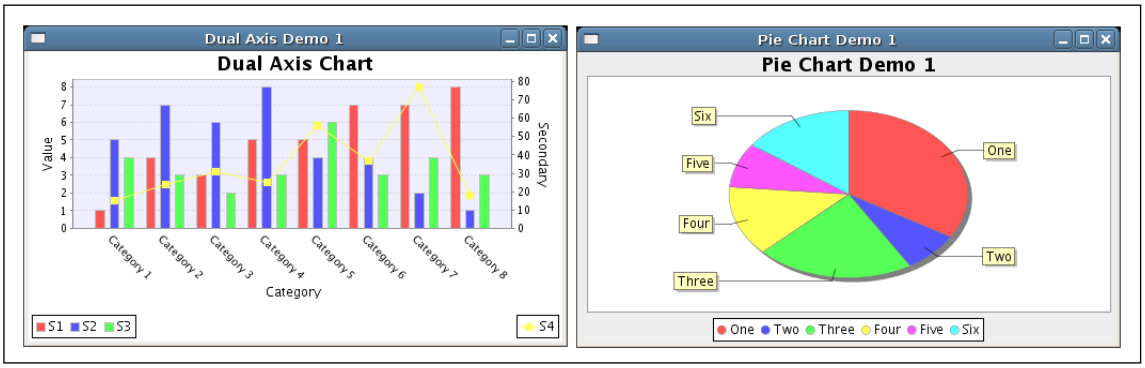

Figura 4.11: Ejemplo de gráficos generados con JFreeChart

4.2.3.12 RMI

RMI (*Remote Method Invocation*) es un mecanismo ofrecido por Java que permite la creación de aplicaciones Java distribuidas, en las cuales los métodos de objetos remotos pueden ser invocados desde otras máquinas virtuales de Java, posiblemente distribuidas en diferentes máquinas. RMI hace uso de la serialización de objetos para poder invocar sus métodos remotamente. Se caracteriza por su facilidad de uso, al estar integrado con Java y permite el paso de objetos por referencia, recolección de basura distribuida y paso de tipos arbitrarios.

Normalmente, una aplicación RMI se compone de un servidor, que crea objetos remotos y crea sus referencias para hacerlos accesibles remotamente, y un cliente, que invoca métodos de esos objetos remotos, obteniendo las referencias que el servidor facilita. La invocación remota de métodos se compone de los siguientes pasos:

- Se encapsulan los parámetros, utilizando la serialización de Java.
- Se invoca el método, quedando el invocador o cliente a la espera.
- Se ejecuta el método remoto y el servidor devuelve la respuesta al invocador, si la hay.
- El cliente recibe la respuesta y continua su ejecución, como se se hubiese utilizado el método de un objeto local.

RMI sigue una arquitectura compuesta de cuatro capas:

1. **Capa de aplicación**: es la capa de implementación de las aplicaciones cliente y servidor, donde se crean y exportan los objetos remotos, haciendo uso de la interfaz *Remote*.
2. **Capa de proxy**: también llamada *capa stub-skeleton*. Es la capa que interactua con la anterior y donde se producen las llamadas a métodos remotos.
3. **Capa de referencia remota**: responsable de manejar las llamadas remotas, mantener las referencias a los objetos exportados, etc.
4. **Capa de transporte**: responsable del transporte de datos de una máquina a otra, utilizando el protocolo JRMP (Java Remote Method Protocol).

RMI se utilizará para la comunicación e invocación de métodos entre la aplicación cliente y el servidor del sistema.

4.2.3.13 Java Reflection API

La reflexión es una técnica utilizada por los programas para examinar o modificar su estructura (introspección) y comportamiento en tiempo de ejecución.

La reflexión se utiliza cuando la clase de un objeto a crear se conoce en tiempo de ejecución, para enumerar los miembros de una clase, para depurar programas, conociendo los miembros privados de las clases, etc. Sin embargo, no se debe abusar de está técnica, ya que introduce problemas de rendimiento, al resolver dinámicamente los tipos de objetos, y problemas de seguridad, al exponer los miembros privados de una clase.

Se utilizará para crear algunos objetos en tiempo de ejecución, ya que sólo en ese punto se conocerá su tipo.

4.2.4 Herramientas y tecnologías para bases de datos

4.2.4.1 MySQL

MySQL es un sistema de gestión de bases de datos (SGBD) relacional, multiusuario y multihilo, de software libre. MySQL está escrito en C y C++ y el *parser* de SQL está escrito en el lenguaje *yacc*. Es una herramienta multiplataforma y con soporte para diferentes lenguajes de programación utilizando los adaptadores adecuados para cada lenguaje.

Entre sus características, destacan las siguientes:

- Soporte para el lenguaje ANSI SQL 99, así como para sus extensiones.
- Soporte multiplataforma, multiusuario y multihilo.
- Procedimientos almacenados.
- Triggers.
- Cursores.
- Vistas actualizables.
- Sistema de procesamiento de transacciones distribuidas, con dos fases de *commit*.

- Soporte de conexiones seguras (SSL).
- Soporte de subconsultas.
- Soporte unicode.
- Soporte para replicación de las bases de datos.

Se utilizará la versión MySQL Commnunity Server 5.5.17 como sistema gestor de bases de datos del sistema.

4.2.4.2 MySQL WorkBench

MySQL Workbench es una herramienta CASE que permite el modelado de bases de datos MySQL en un entorno visual (ver Figura 4.12). Sus características principales son:

- **Modelado**: proporciona las herramientas necesarias para el modelado Entidad-Interrelación (modelo ER) de la base de datos: además, soporta ingeniería directa, obteniendo el código SQL a partir del modelo, e ingeniería inversa.
- **Desarrollo**: proporciona herramientas visuales para crear, ejecutar y optimizar sentencias SQL.
- **Administración**: proporciona una consola visual para administrar fácilmente el servidor MySQL, cuentas de usuario, permisos, etc.

Se utilizará MySQL Workbench 5.2 CE para el modelado de la base de datos, para el desarrollo SQL y para administrar la base de datos MySQL.

4.2.5 Herramientas para la documentación del proyecto

4.2.5.1 LaTeX

LaTeX es un lenguaje de marcado utilizado para la creación de documentos, especialmente libros y documentos científico-técnicos. Está formado por un gran conjunto de macros (u órdenes) del lenguaje TeX, de código abierto y que permite crear libros, tesis y artículos técnicos con una elevada calidad tipográfica, comparable a la de una editorial científica.

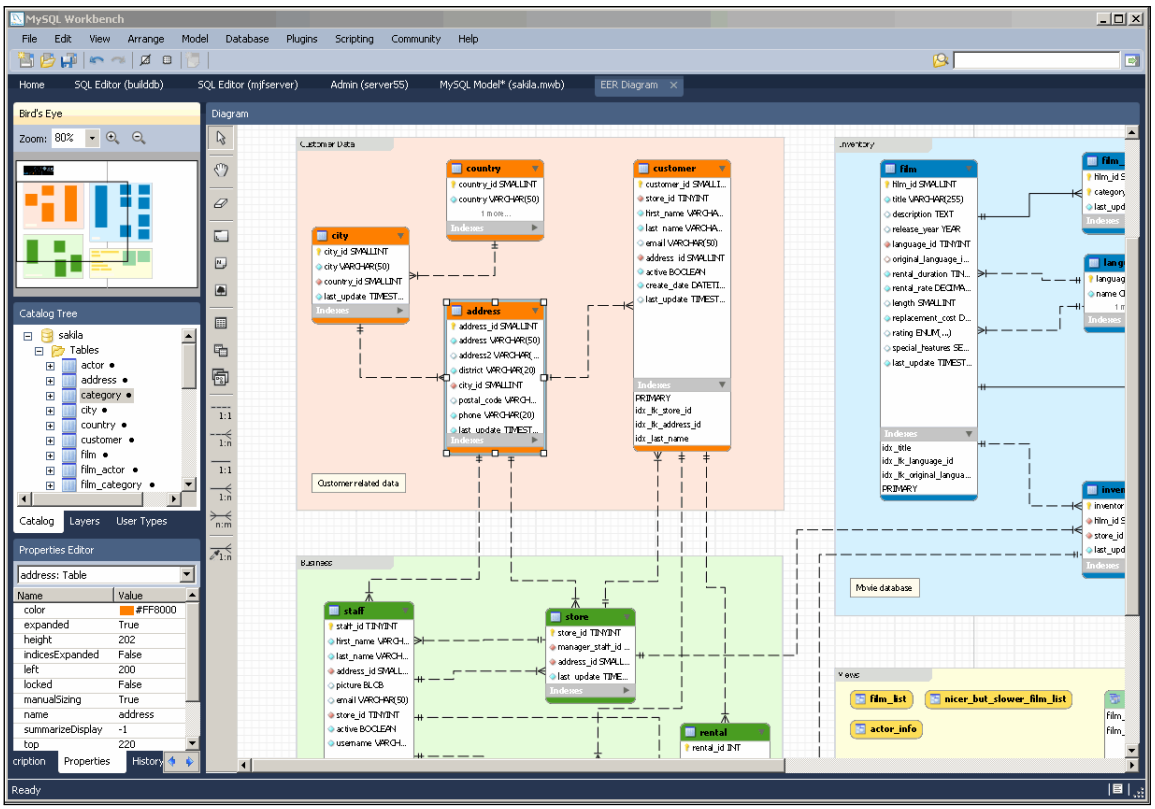

Figura 4.12: MySQL Workbench

A diferencia de los editores de texto habituales, conocidos como *WYSIWYG* (es decir, *What You See Is What You Get*, o *lo que ves es lo que obtienes*), LaTeX permite centrarse exclusivamente en el contenido del documento, sin preocuparse de los detalles del formato del texto. También posee capacidades gráficas para representar fórmulas complejas, ecuaciones, notación científica, etc y permite estructurar de una manera muy sencilla el documento, con capítulos, secciones, generación automática de índices, etc. Es por estas razones por las que LaTeX se ha extendido rápidamente por todo el sector científico y técnico, llegando a convertirse incluso en un uso obligado para presentar publicaciones en ciertos congresos y revistas.

Para crear un documento con LaTeX, hay que utilizar un editor de texto para crear los archivos fuente, con las macros y el contenido adecuado para, posteriormente, procesarlo, compilarlo y generar la salida, normalmente en PDF, aunque se pueden generar diferentes salidas como DVI, PS, etc.

Para la generación del presente documento, se ha utilizado LaTeX, usando como editor de textos **TexnicCenter**, el cual es un editor de software libre para sistemas Windows que

proporciona las herramientas necesarias para la composición y compilación de textos escritos en LaTeX (ver Figura 4.13).

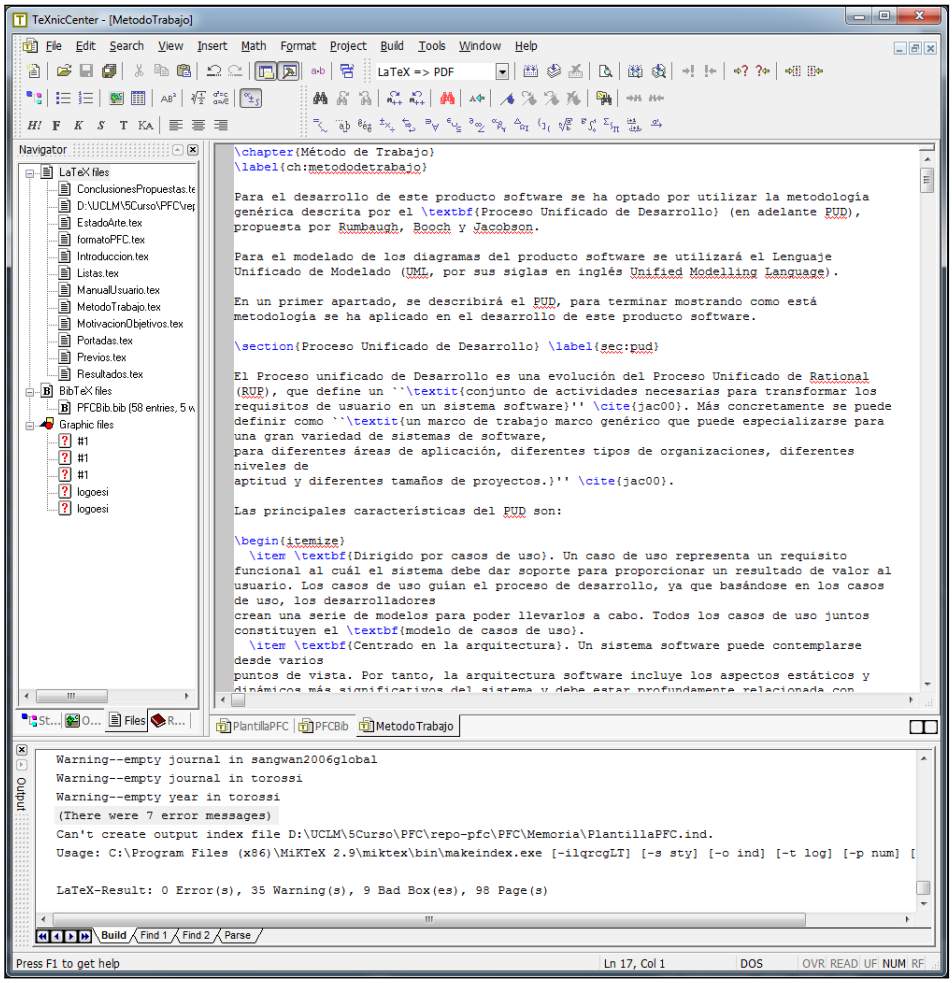

Figura 4.13: Editor TeXnicCenter

4.2.5.2 BibTeX

BibTeX es un software de gestión de referencias para dar formato a dichas referencias, usado normalmente en conjunto con LaTeX. Facilita la cita de fuentes con un formato consistente, separando la información bibliográfica de la presentación de la misma.

BibTeX utiliza un fichero con extensión *.bib* para definir la lista de elementos bibliográficos, que pueden ser artículos, tesis, libros, manuales, parte de libros o artículos, etc. Todos estos tipos de bibliografía deben definirse con una sintaxis concreta, indicando elementos como autor, título, editorial, fecha de publicación, URL, etc.

Se ha utilizado BibTeX para formatear las citas literarias y bibliográficas de este documento. Además, se ha utilizado **JabRef**, que es una herramienta visual basada en Java que permite editar los tipos bibliográficos utilizados en el archivo *.bib* de BibTeX (ver Figura 4.14).

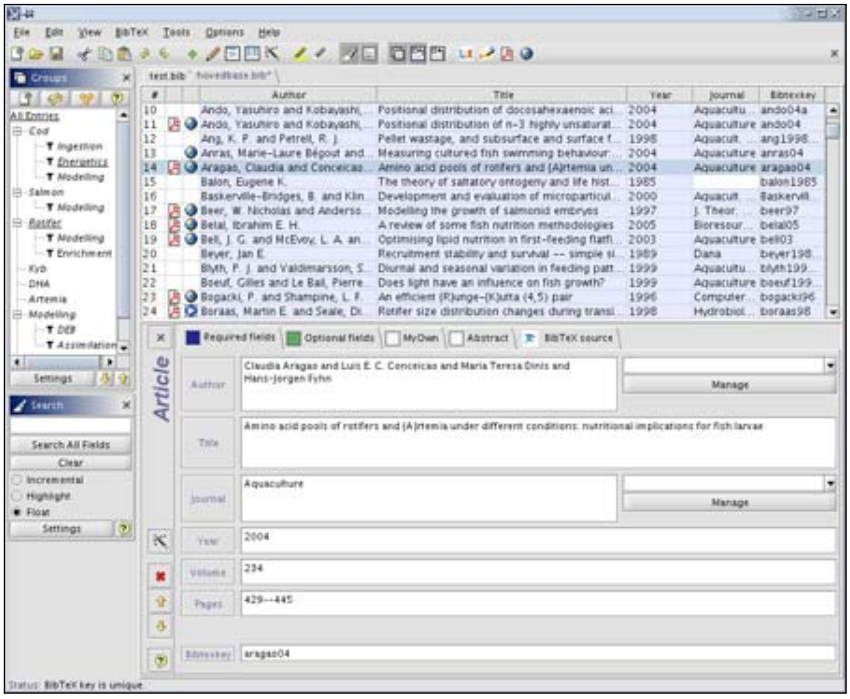

Figura 4.14: Herramienta JabRef

Capítulo 5

Resultados

En este capítulo se expondrán los resultados obtenidos al aplicar la metodología descrita en el capítulo 4 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.

5.1 Fase de Inicio

En esta fase, como se muestra en la Figura 5.1, 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.

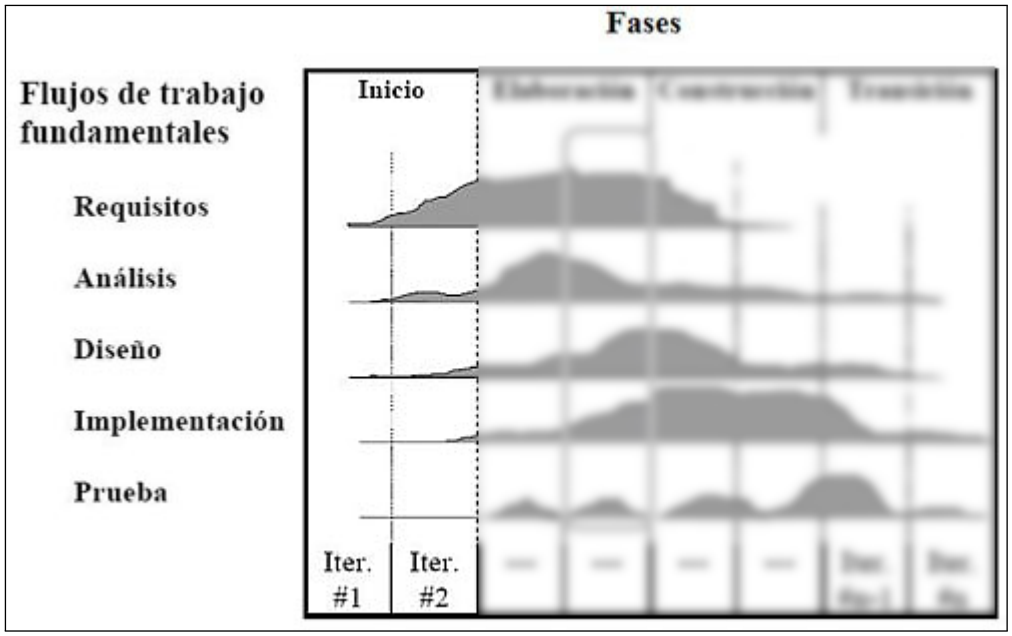

Figura 5.1: Fase de inicio en el PUD

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

5.1.1 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 3.

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 2.

5.1.1.1 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 5.2.

Junto a los requisitos capturados del sistema, se identificaron los roles de usuario mostrados en la Tabla 5.3, así como los requisitos, o acciones, que cada uno de estos roles podían ejecutar. En la Tabla 5.1 se muestran las acciones que cada usuario puede realizar en el sistema.

5.1.1.2 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 5.1: 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 5.2: 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 5.3: Roles identificados en el sistema

5.1.2 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 5.2. 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 5.1.1.2), 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.

5.1.2.1 Modelo de casos de uso para el subsistema cliente

En la Figura 5.2 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.

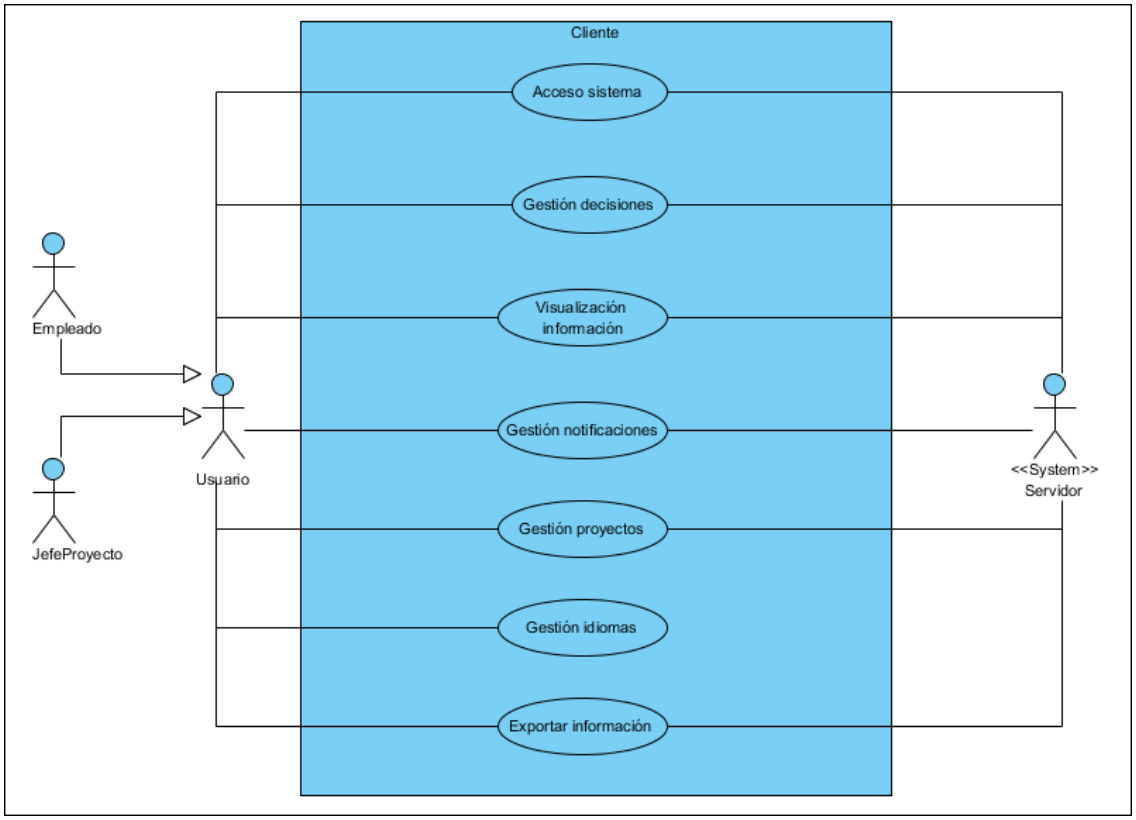

Figura 5.2: Diagrama de casos de uso - Cliente - v1.0

5.1.2.2 Modelo de casos de uso para el subsistema servidor

En la Figura 5.3 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*.

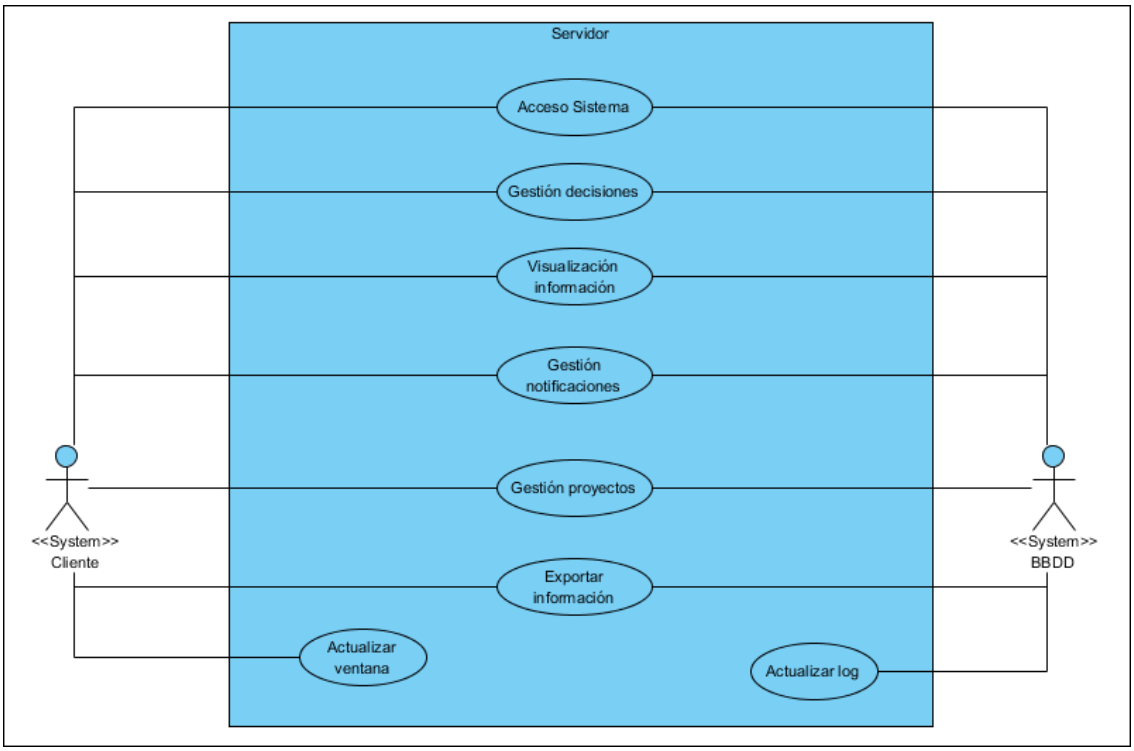

Figura 5.3: Diagrama de casos de uso - Servidor - v1.0

5.1.3 Glosario de términos

El glosario de términos está compuesto por los términos que fueron definidos en la sección 2.2.1.

5.1.4 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.

5.1.5 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 5.7.

5.2 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 5.4).

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.

5.2.1 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 5.4: Primera versión del plan de iteraciones

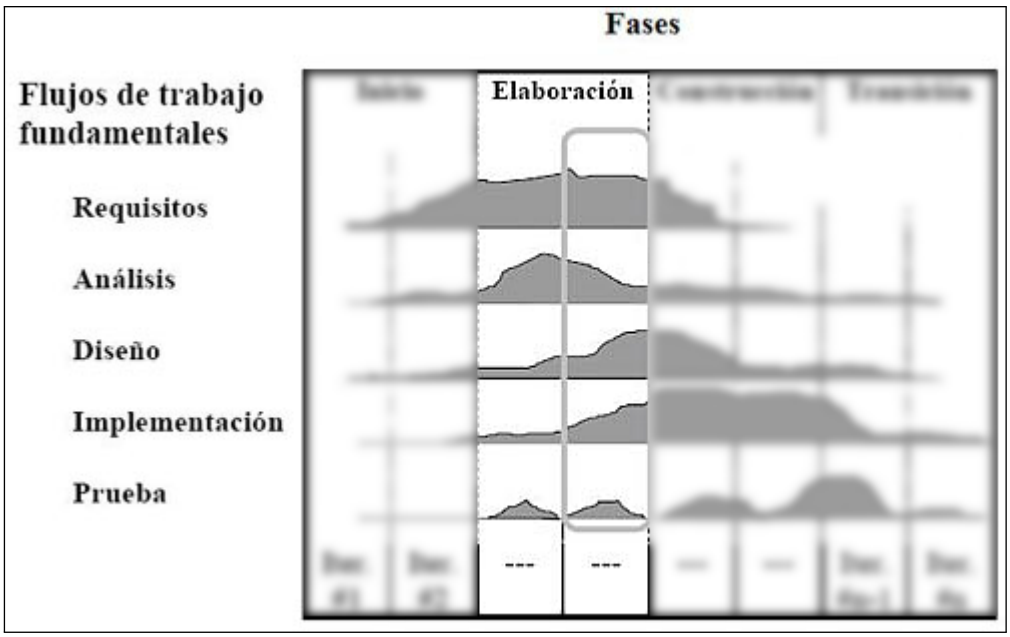

Figura 5.4: Fase de elaboración en el PUD

- 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.

5.2.1.1 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 5.5 se muestran los nuevos grupos funcionales actualizados, incluyendo los nuevos requisitos que se han identificado en esta fase.

Por otra parte, en la Tabla 5.6 se muestran las acciones que se pueden realizar en el sistema según el rol del usuario.

5.2.1.2 Modelo de casos de uso

En la Figura 5.5 se muestra el diagrama de casos de uso que representa los grupos funcionales actualizados en el subsistema cliente. Del mismo modo, la Figura 5.6 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 5.5. 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 5.5: 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 5.6: Acciones que un usuario puede realizar en el sistema - v2.0

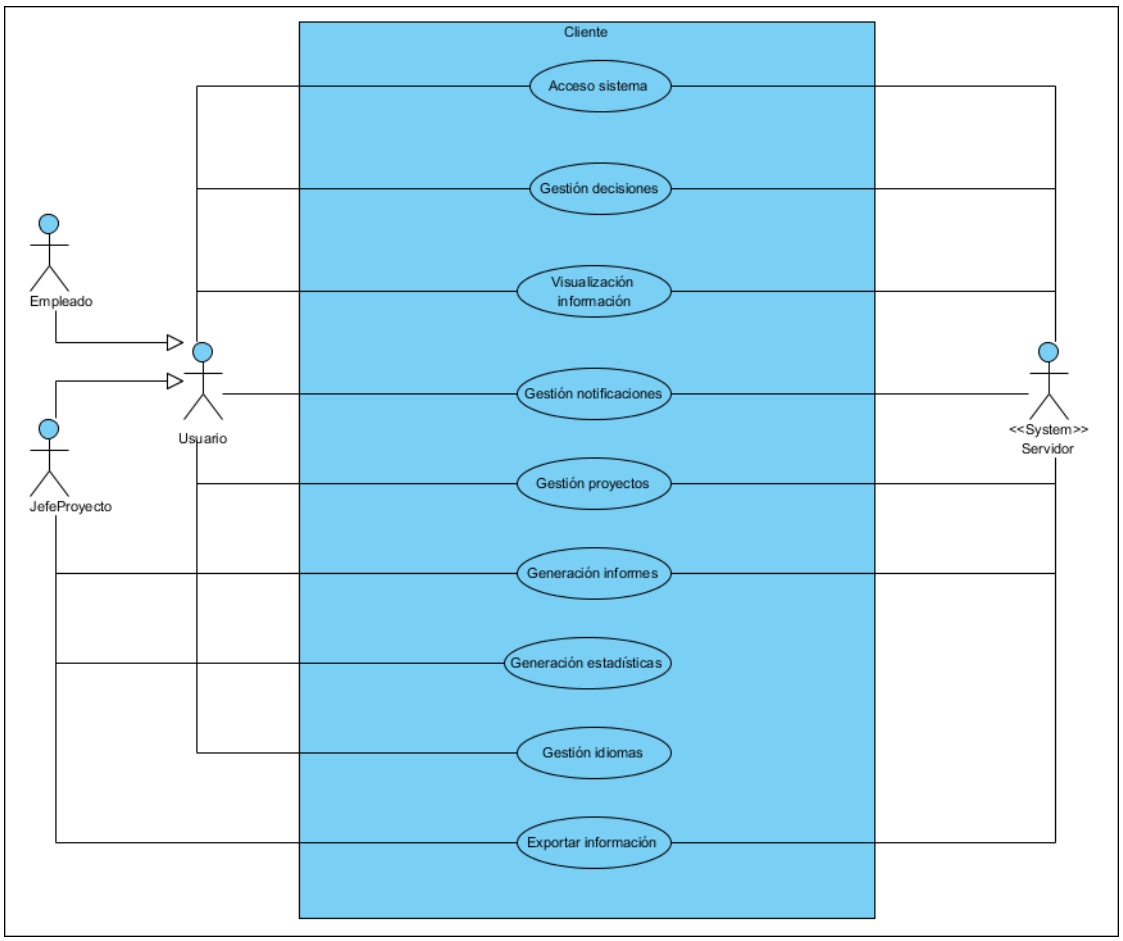

Figura 5.5: Diagrama de casos de uso - Cliente - v2.0

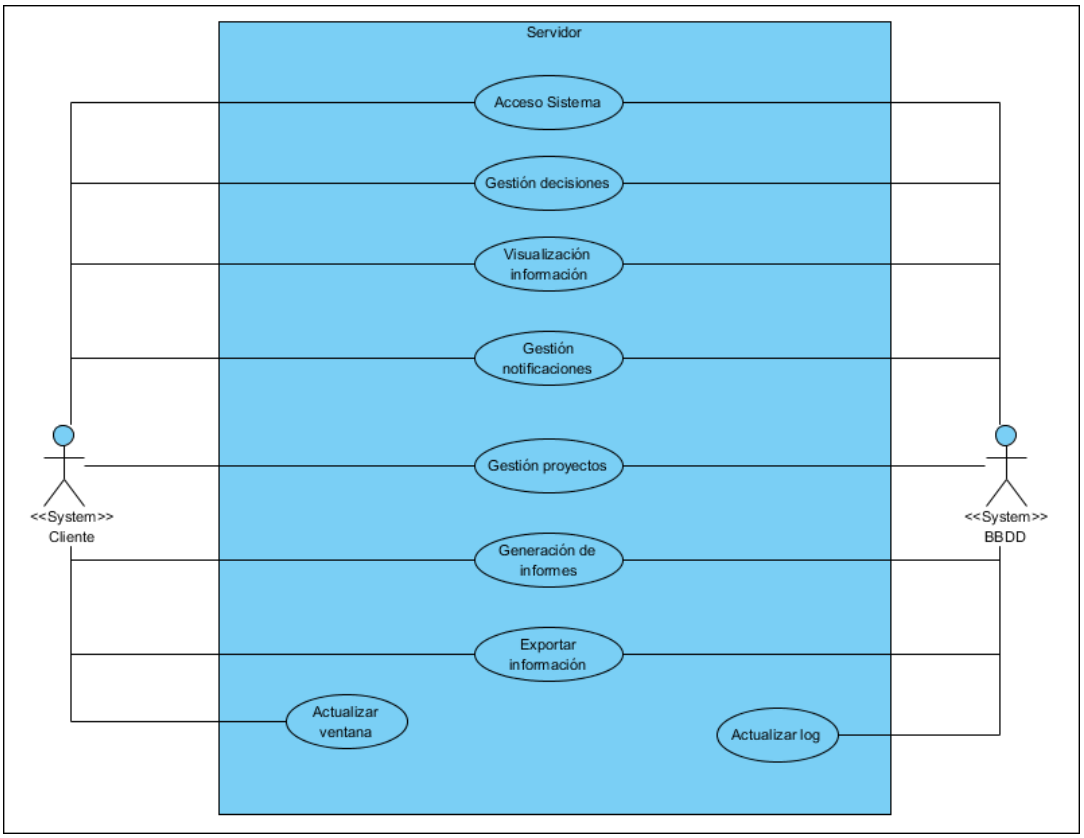

Figura 5.6: Diagrama de casos de uso - Servidor - v2.0

5.2.1.2.1 Acceso al sistema

En la Figura 5.7 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 5.8 se muestra el diagrama de casos de uso para el servidor.

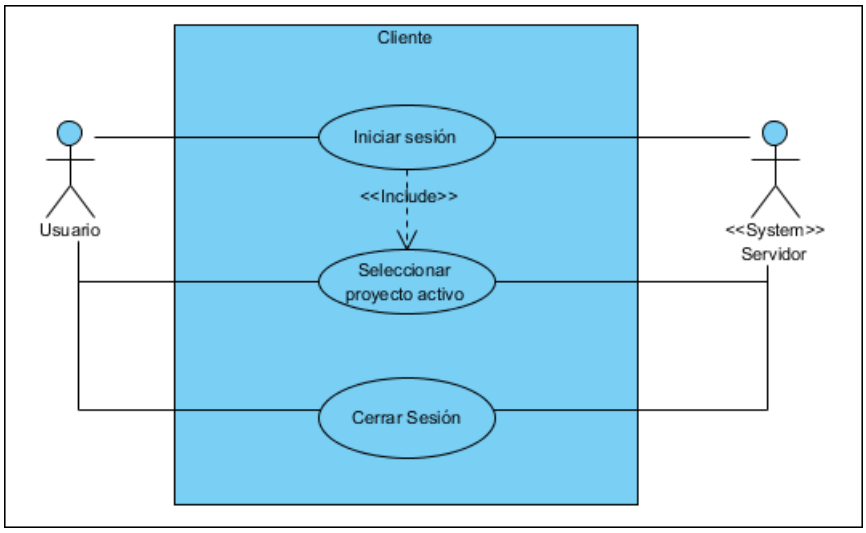

Figura 5.7: Diagrama de casos de uso - Cliente - Acceso al sistema

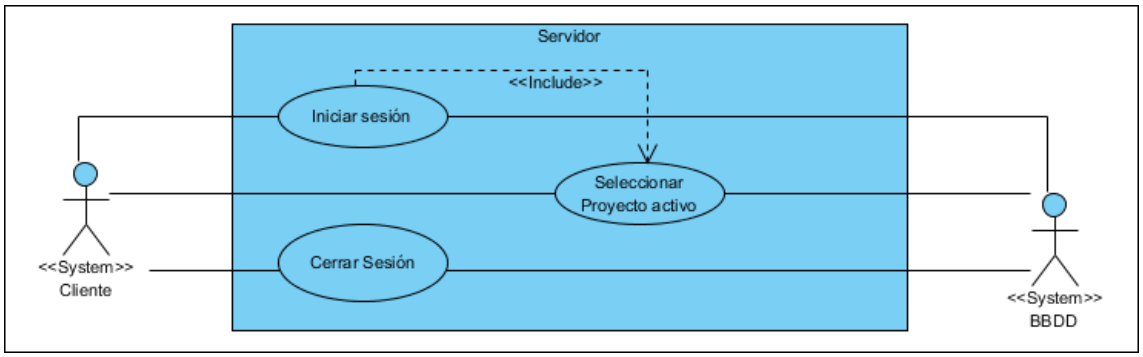

Figura 5.8: Diagrama de casos de uso - Servidor - Acceso al sistema

5.2.1.2.2 Gestión de decisiones

En la Figura 5.9 se muestra el diagrama de casos de uso para el cliente, mientras que en Figura 5.10 refleja el diagrama de casos de uso para el servidor.

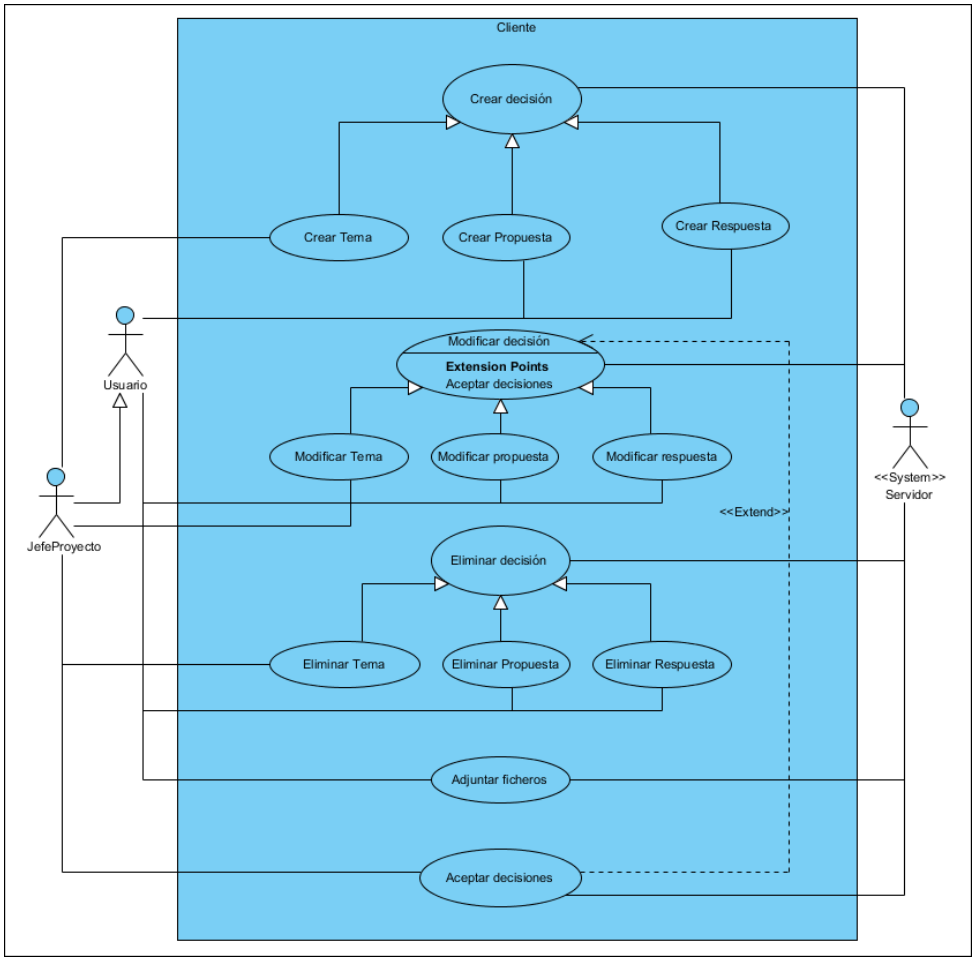

Figura 5.9: Diagrama de casos de uso - Cliente - Gestión decisiones

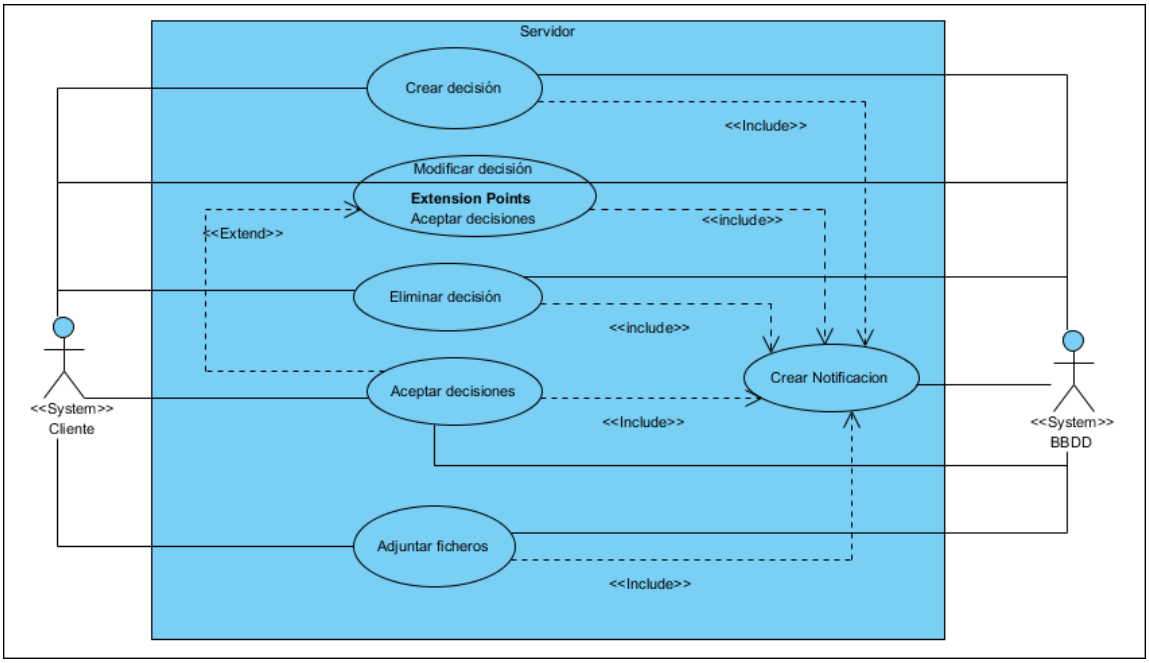

Figura 5.10: Diagrama de casos de uso - Servidor - Gestión decisiones

5.2.1.2.3 Visualización de información

En la Figura 5.11 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 5.12 se observa el diagrama de casos de uso para el servidor.

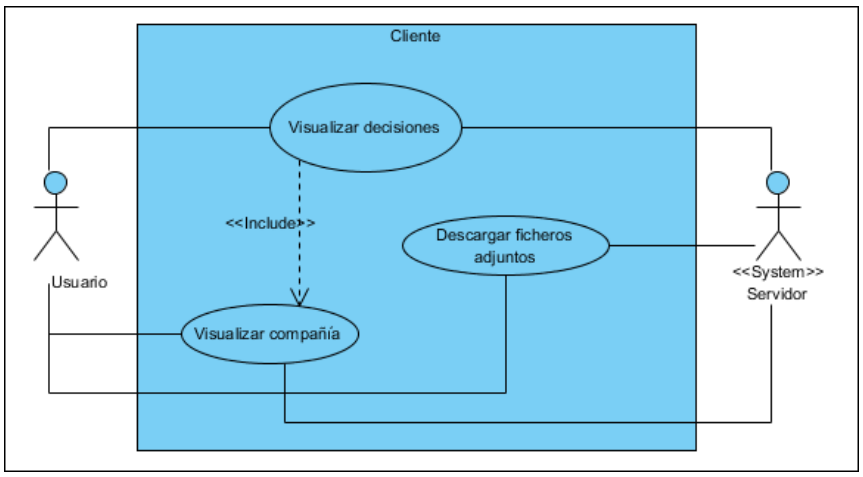

Figura 5.11: Diagrama de casos de uso - Cliente - Visualización información

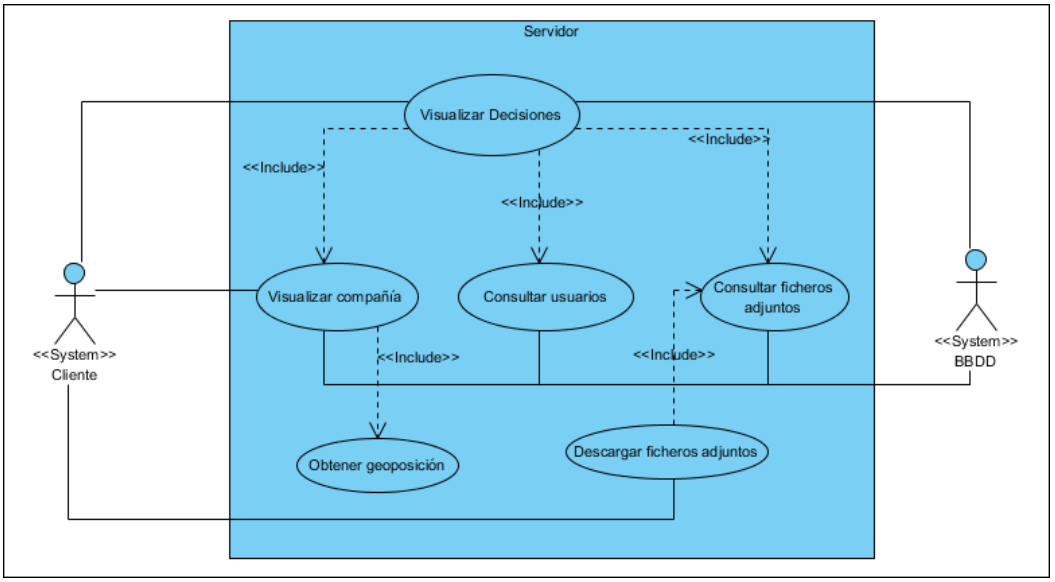

Figura 5.12: Diagrama de casos de uso - Servidor - Visualización información

5.2.1.2.4 Gestión de notificaciones

En la Figura 5.13 se muestra el diagrama de casos de uso para el cliente, mientras que la Figura 5.14 refleja el diagrama de casos de uso para el servidor.

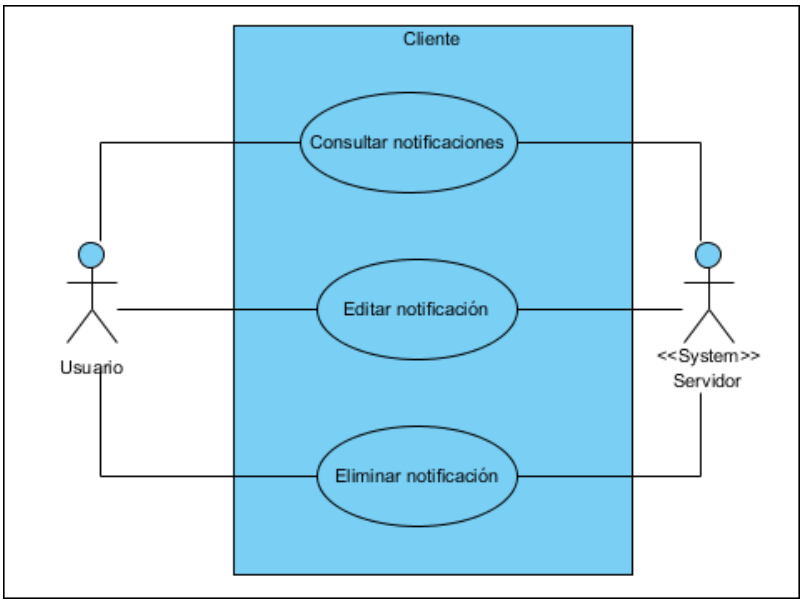

Figura 5.13: Diagrama de casos de uso - Cliente - Gestión notificaciones

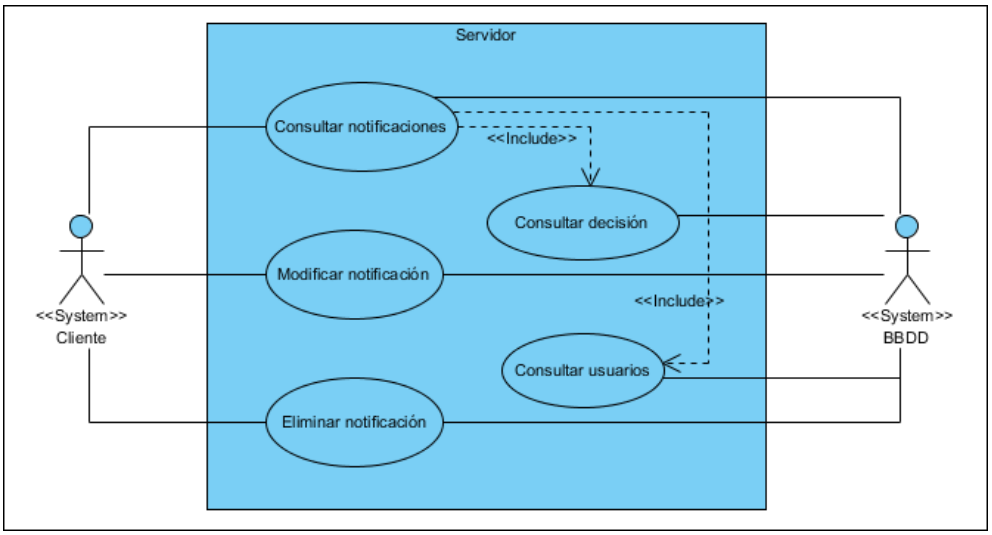

Figura 5.14: Diagrama de casos de uso - Servidor - Gestión notificaciones

5.2.1.2.5 Gestión de proyectos

En la Figura 5.15 se muestra el diagrama de casos de uso para el cliente, mientras que la Figura 5.16 refleja el diagrama de casos de uso para el servidor.

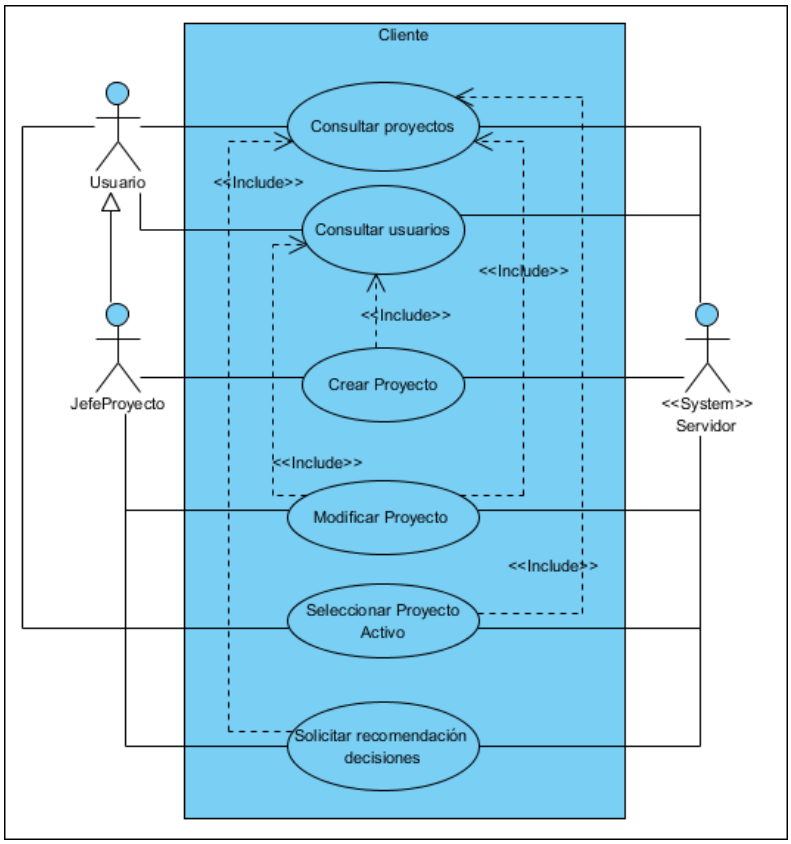

Figura 5.15: Diagrama de casos de uso - Cliente - Gestión Proyectos

5.2.1.2.6 Generación de estadísticas

En la Figura 5.17 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.

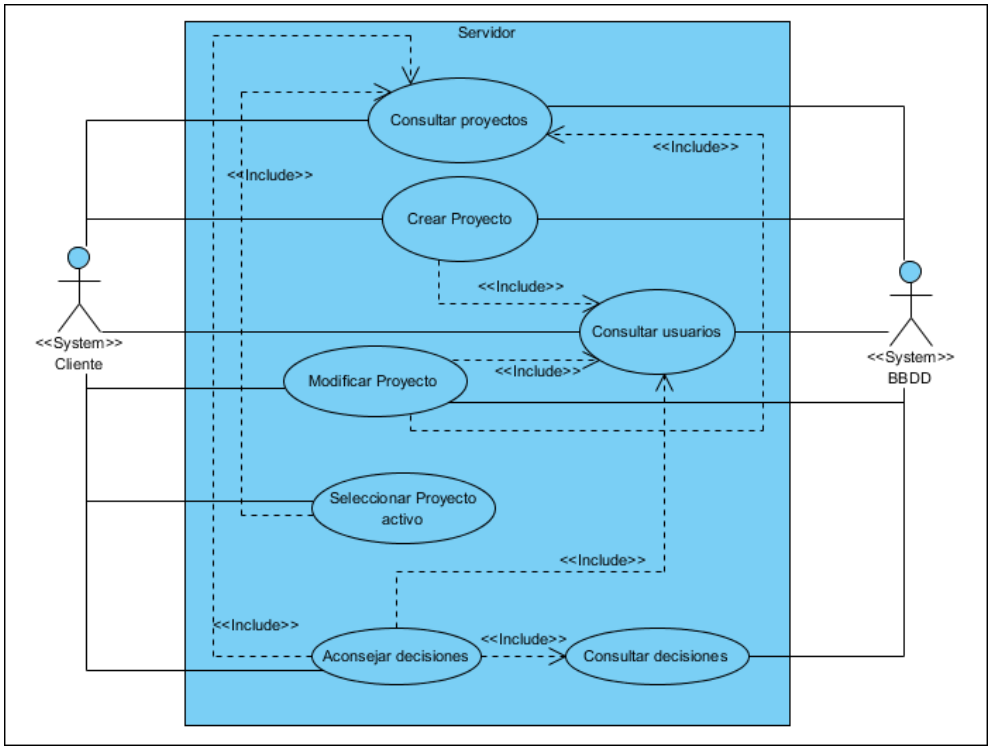

Figura 5.16: Diagrama de casos de uso - Servidor - Gestión Proyectos

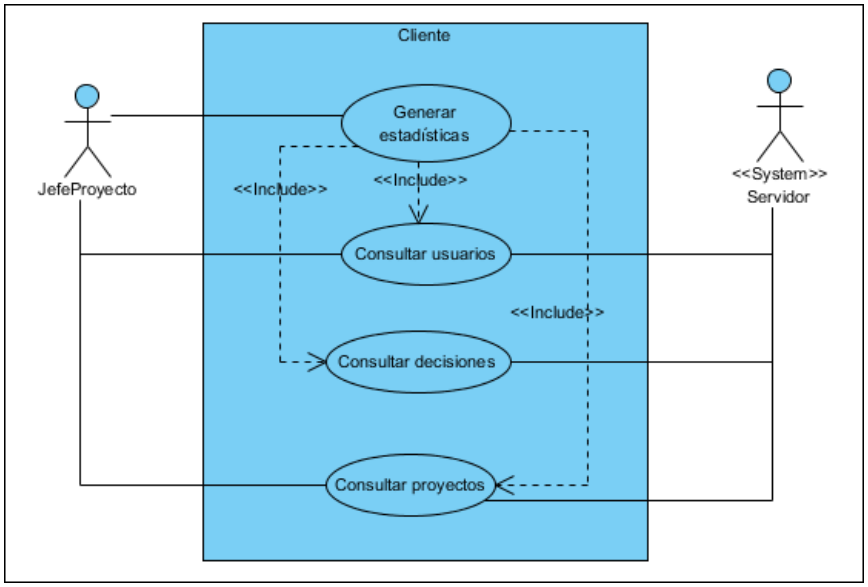

Figura 5.17: Diagrama de casos de uso - Cliente - Generación Estadísticas

5.2.1.2.7 Generación de informes

En la Figura 5.18 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 5.19 se muestra el diagrama de casos de uso para el servidor.

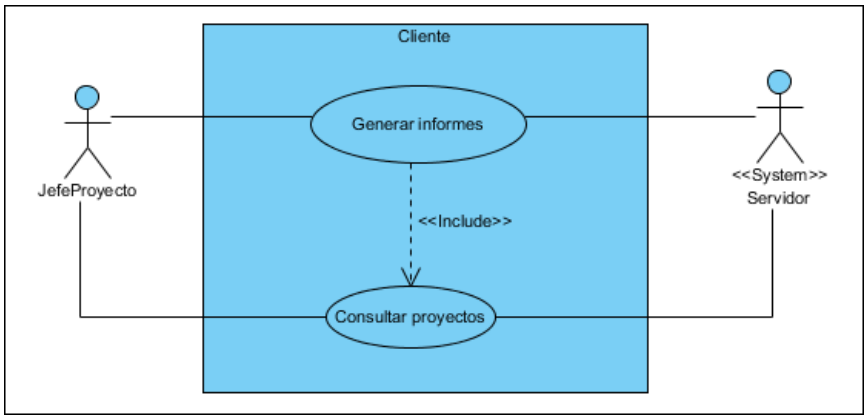

Figura 5.18: Diagrama de casos de uso - Cliente - Generación Informes

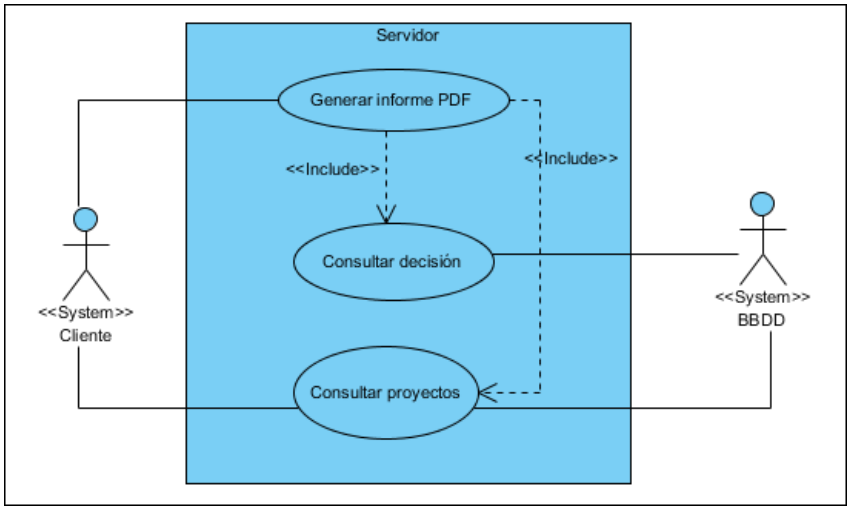

Figura 5.19: Diagrama de casos de uso - Servidor - Generación Informes

5.2.1.2.8 Gestión de idiomas

En la Figura 5.20 se muestra el diagrama de casos de uso para el cliente.

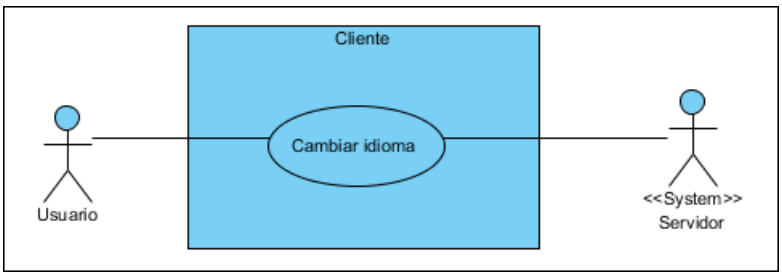

Figura 5.20: Diagrama de casos de uso - Cliente - Gestión Idiomas

5.2.1.2.9 Exportar información

En la Figura 5.21 se muestra el diagrama de casos de uso para el cliente, mientras que en la Figura 5.22 se muestra el diagrama de casos de uso para el servidor.

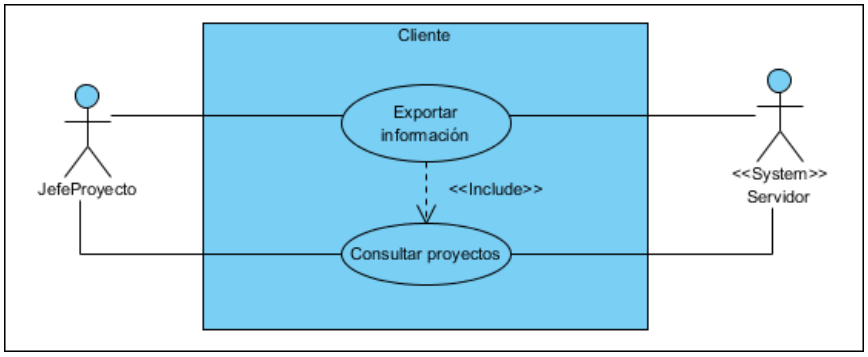

Figura 5.21: Diagrama de casos de uso - Cliente - Exportar información

5.2.1.3 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 5.7, 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 5.7: Primera versión del plan de iteraciones

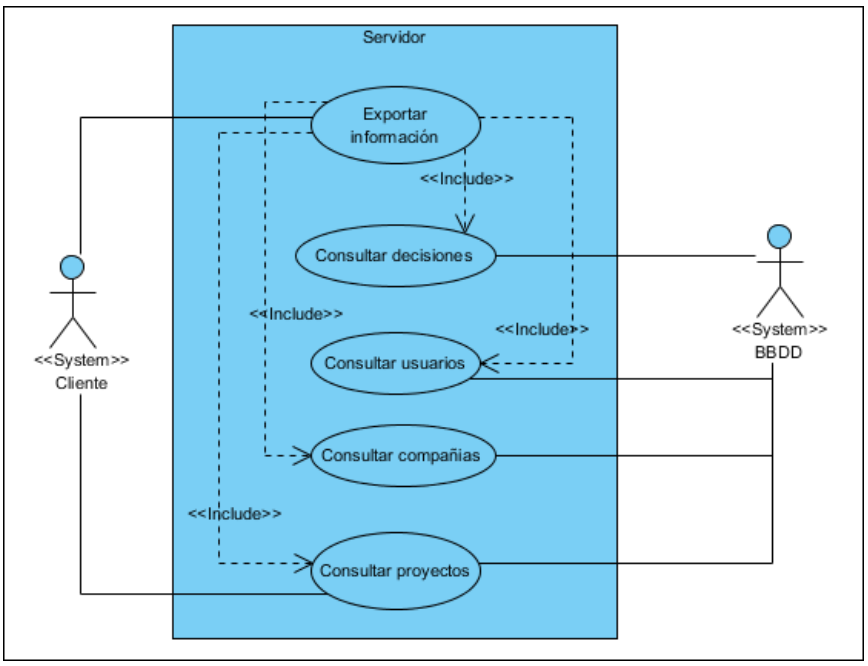

Figura 5.22: Diagrama de casos de uso - Servidor - Exportar información

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

5.2.1.4 Arquitectura del sistema

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

5.2.1.4.1 Arquitectura cliente-servidor

Como se comentó en el apartado 5.1.1.2, 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 4.2). 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 5.23 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.

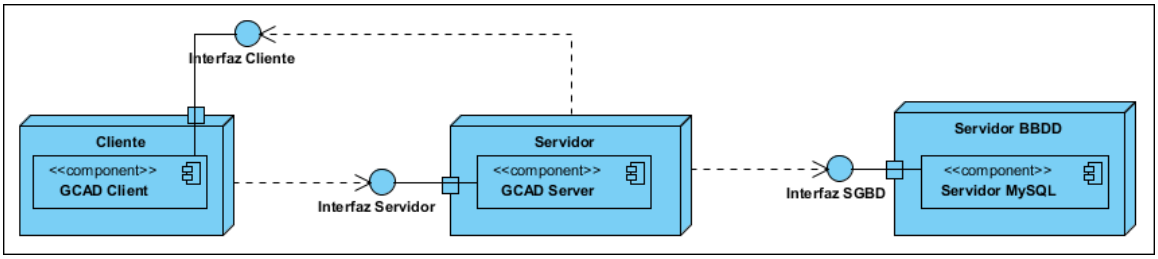

Figura 5.23: Arquitectura cliente-servidor

5.2.1.4.2 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 5.24).

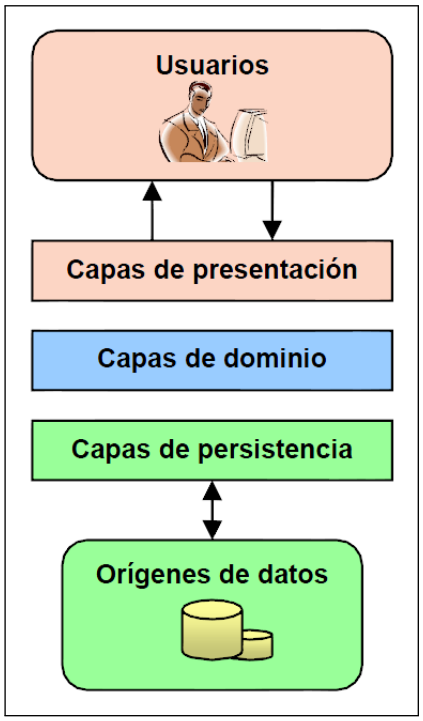

Figura 5.24: Arquitectura multicapa

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 5.25 puede observarse esta arquitectura multicapa y las relaciones entre dichas capas.

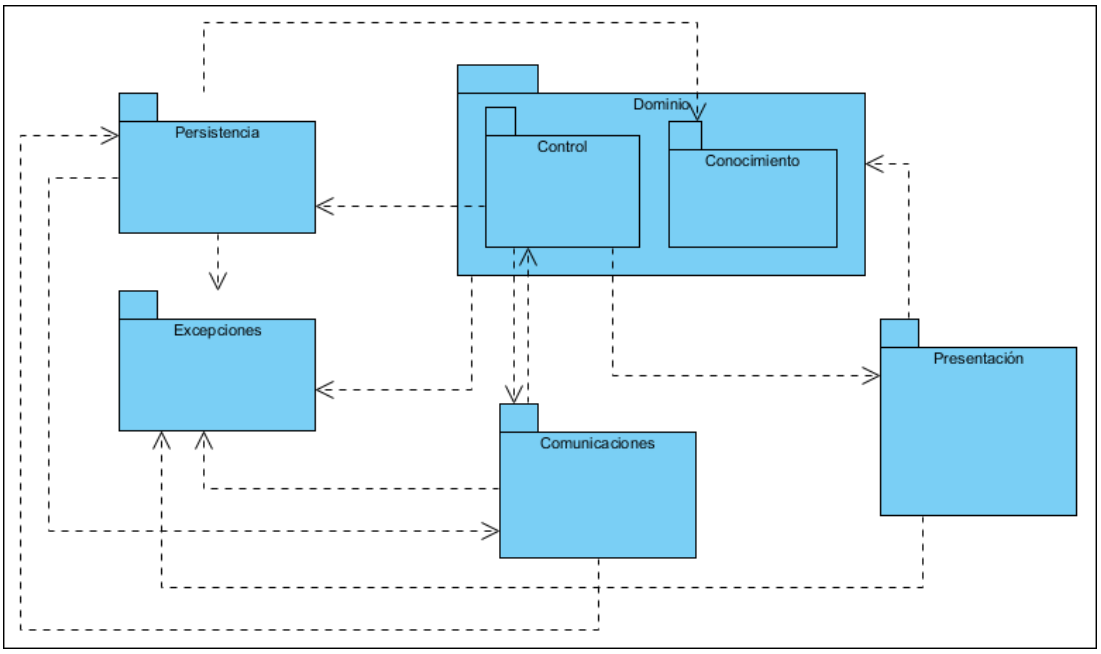

Figura 5.25: Arquitectura multicapa

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.

5.2.1.4.3 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.

5.2.2 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 5.2.1.4, 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.

5.2.2.1 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 5.1.1.1, 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 5.3.2.1.2. 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 5.26 se puede observar este diagrama de clases de dominio.

5.2.2.2 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 5.3.3.2.1.

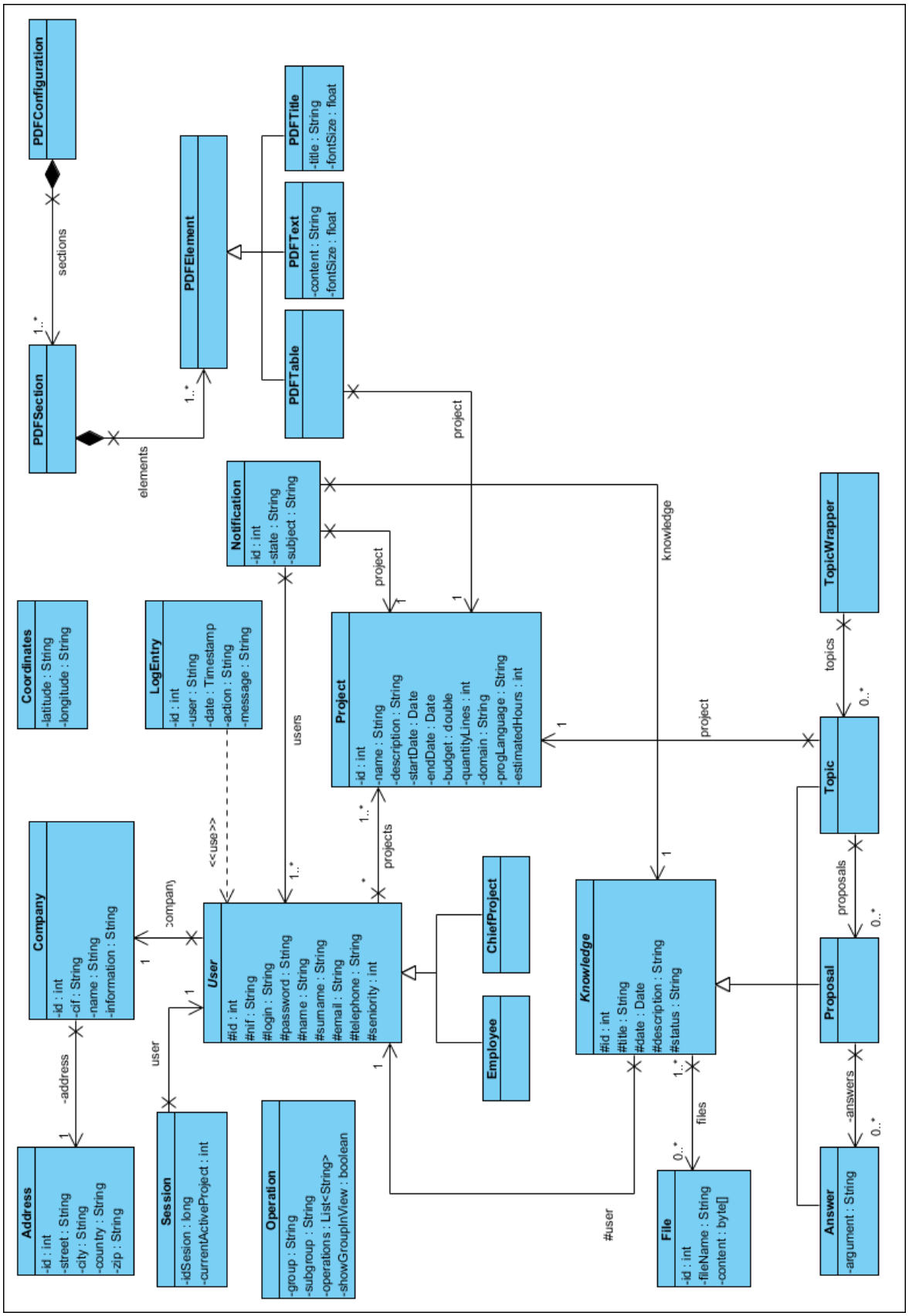

Figura 5.26: Diagrama de clases de dominio

Teniendo en cuenta dichas consideraciones, se obtiene el modelo EER (Entidad-Interrelación Extendido) de la base de datos, mostrado en la Figura 5.27.

Para terminar, cabe destacar que este modelo se ha creado utilizando la herramienta **MySQL Workbench** (ver sección 4.2), 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.

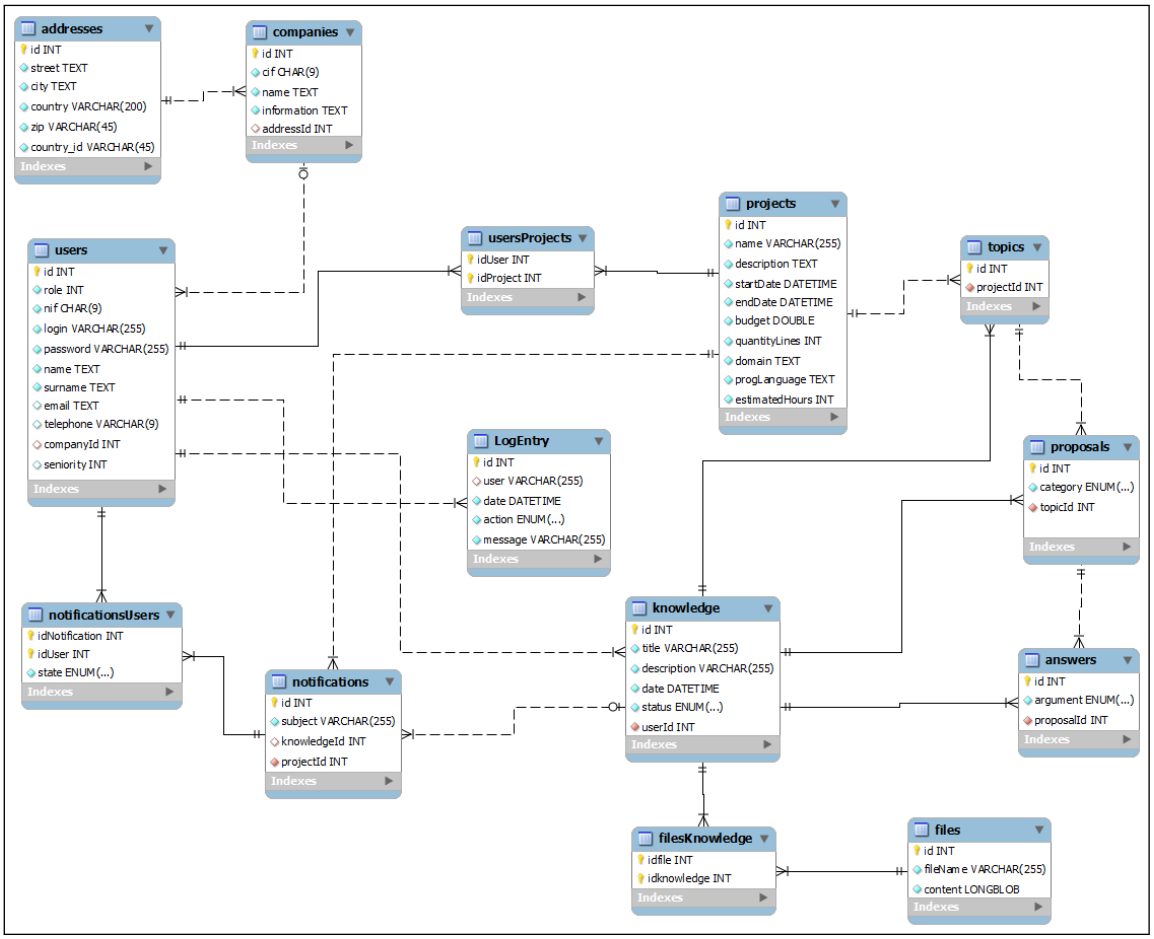

Figura 5.27: Diagrama EER de la base de datos

5.2.2.3 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 5.23).

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

5.2.2.3.1 Comunicación entre cliente y servidor

El primer paso es hacer que las clases de dominio mostradas en la Figura 5.26 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 5.28 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).

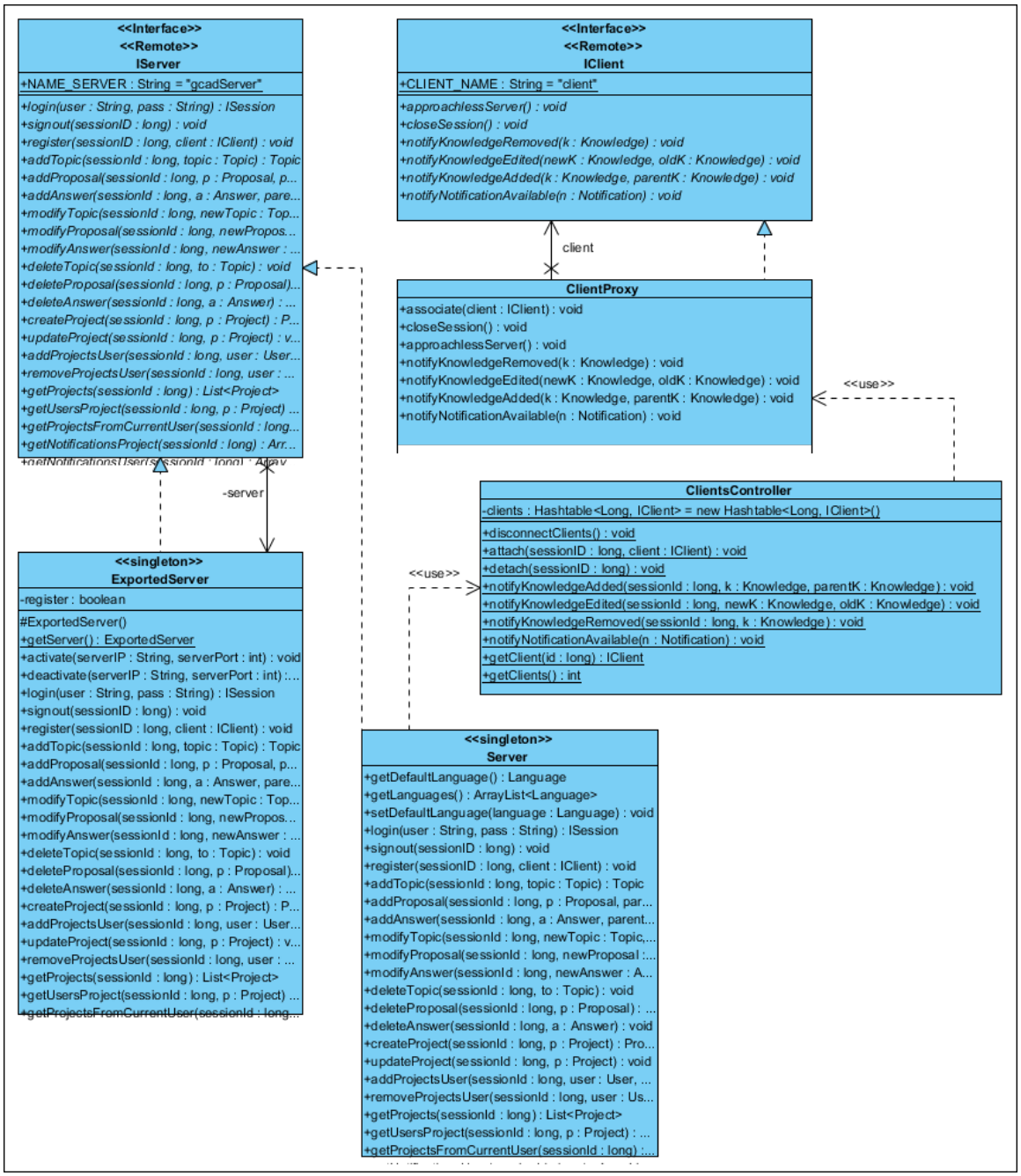

Figura 5.28: Diagrama de clases - Capa de comunicación cliente-servidor

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 5.1 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 5.1: 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 5.2 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 5.2: 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.

5.2.2.3.2 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 5.29 se puede observar el diagrama de clases que representa este observador.

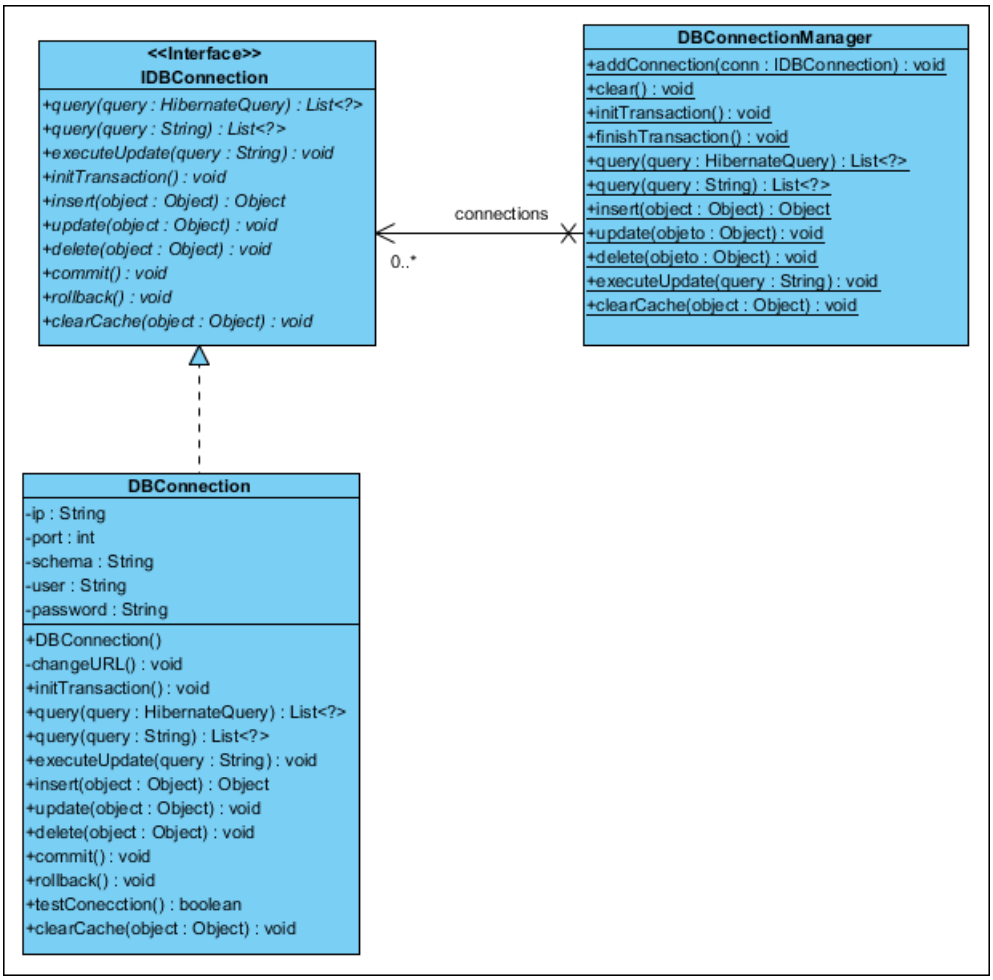

Figura 5.29: Diagrama de clases - Capa de comunicación para bases de datos

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 5.30 se puede observar el diagrama de clases que representa este observador.

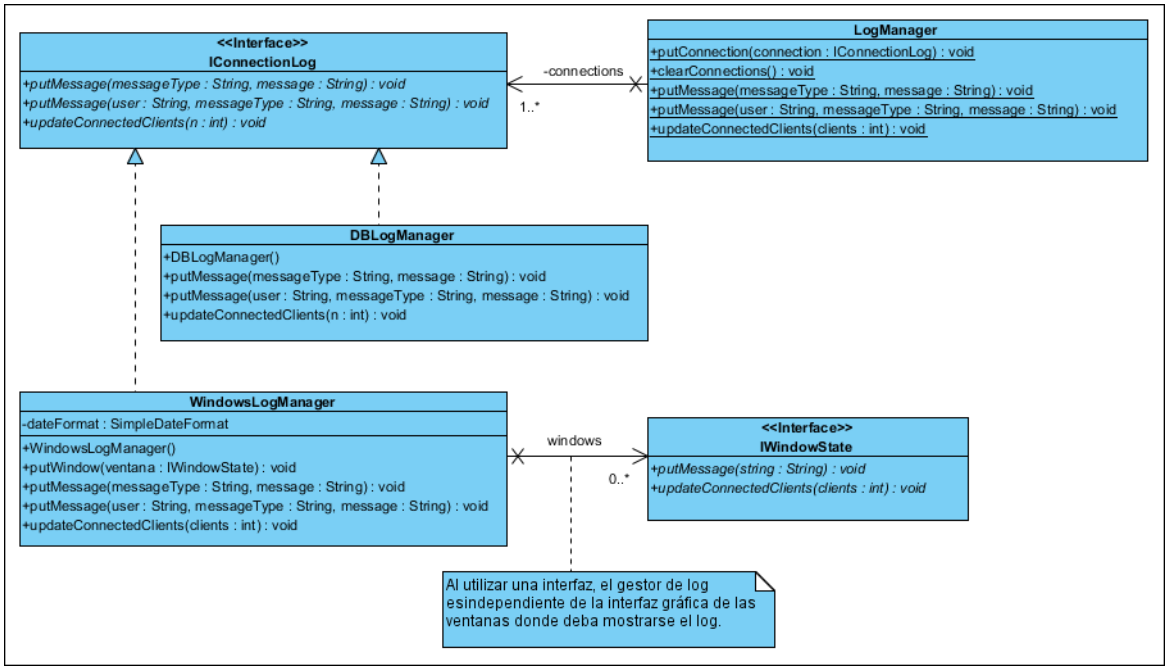

Figura 5.30: Diagrama de clases - Capa de comunicación para gestionar el log

5.2.2.4 Pruebas

Para concluir con esta iteración, se diseñan los casos de prueba unitarios para probar la comunicación entre el subsistema servidor y el cliente, y del servidor con la base de datos. Para ello, se diseño un conjunto de casos de pruebas, o *test cases*, agrupados bajo una *suite* de pruebas. Dicha *suite* es ejecutada posteriormente por **JUnit** (ver 4.2), obteniendo los resultados de las pruebas, pudiendo corregir los errores encontrados, si los hubiera.

A continuación se muestran algunos de los casos de prueba implementados y probados para esta iteración.

Caso de uso: Comunicación entre subsistemas
Caso de prueba: 1
Descripción: Comunicación entre el servidor y la base de datos
Precondiciones: Ninguna
Resultado: Se crea una conexión con la base de datos y se insertan y recuperan algunos objetos de prueba de ella, realizándose correctamente dichos accesos a la base de datos

Tabla 5.8: Descripción del caso de prueba 1 para la comunicación entre subsistemas

Caso de uso: Comunicación entre subsistemas
Caso de prueba: 2
Descripción: Comunicación entre el servidor y el cliente
Precondiciones: Ninguna
Resultado: El servidor es exportado por RMI y se comunica con el cliente, invocando operaciones y actualizando su estado de manera correcta

Tabla 5.9: Descripción del caso de prueba 2 para la comunicación entre subsistemas

En la Figura **??** se muestra una captura de pantalla del *suite* de casos de pruebas implementado, junto a su ejecución con JUnit. Un aspecto a destacar en este caso es la implementación de un subsistema cliente de prueba, o *dummy*, necesario para que el servidor pudiera comunicarse con él y probar dicha comunicación y la invocación de sus métodos.

De un modo análogo, en el subsistema cliente se implementa también un *test suite* para probar la comunicación con el servidor, siendo necesario también la creación de un servidor

dummy.

En cuanto a la cobertura de código obtenida por los casos de prueba, en el Apéndice **??** se presenta el informe final de cobertura alcanzado con las pruebas de todos los casos de uso. Dicho informes es generado por el plug-in de eclipse **Eclemma** (ver sección 4.2).

5.3 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 5.31. Esta fase, formada por 6 iteraciones, es la fase de más duración del desarrollo del producto software.

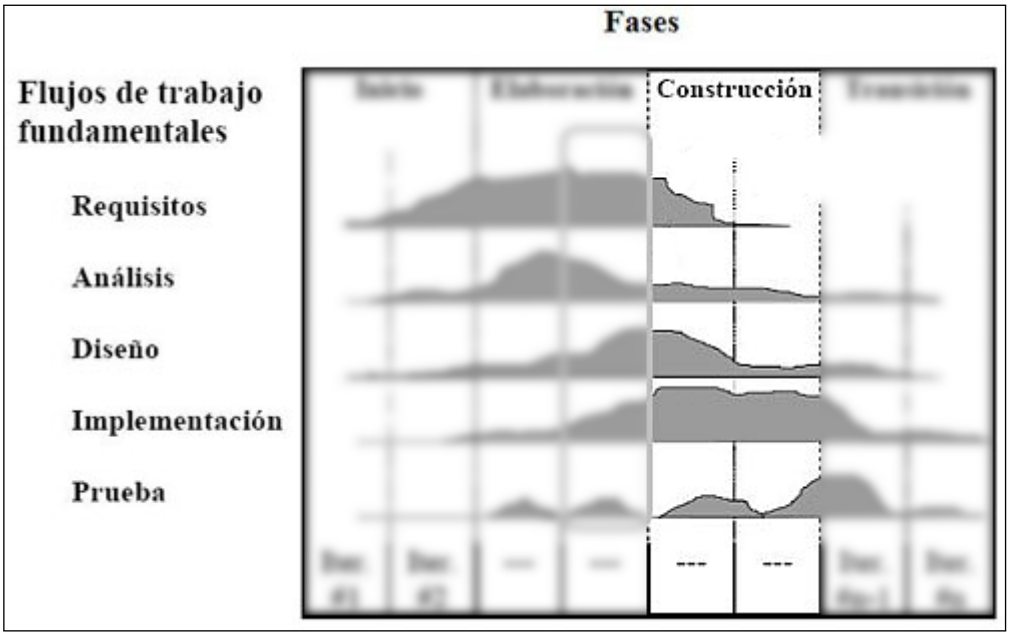

Figura 5.31: Fase de construcción en el PUD

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.

5.3.1 Iteración 3

Siguiendo los casos del grupo funcional *F1: Acceso al sistema* (ver Figura 5.12), 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.

5.3.1.1 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 5.8.

5.3.1.1.1 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 5.10 se describe el caso de uso *Login*.

En la Figura 5.32 se muestra el diagrama de clases de análisis para el subsistema cliente. La Figura 5.33 refleja el diagrama de clases de análisis para el subsistema servidor.

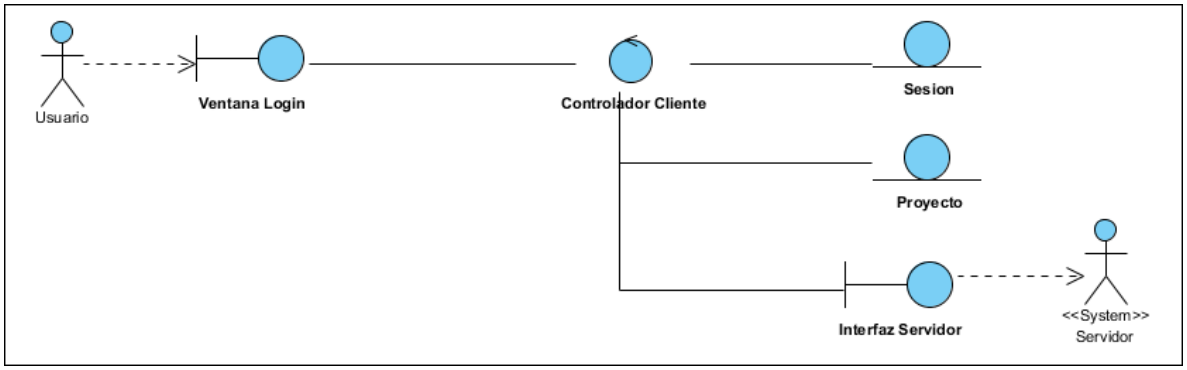

Figura 5.32: Diagrama de clases de análisis - Cliente - Login

5.3.1.1.2 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 5.34 se muestra el diagrama de secuencia para el caso de uso *Login* en el cliente, mientras que la Figura 5.35 refleja el diagrama de secuencia para el servidor.

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 5.10: Especificación del caso de uso *Login*

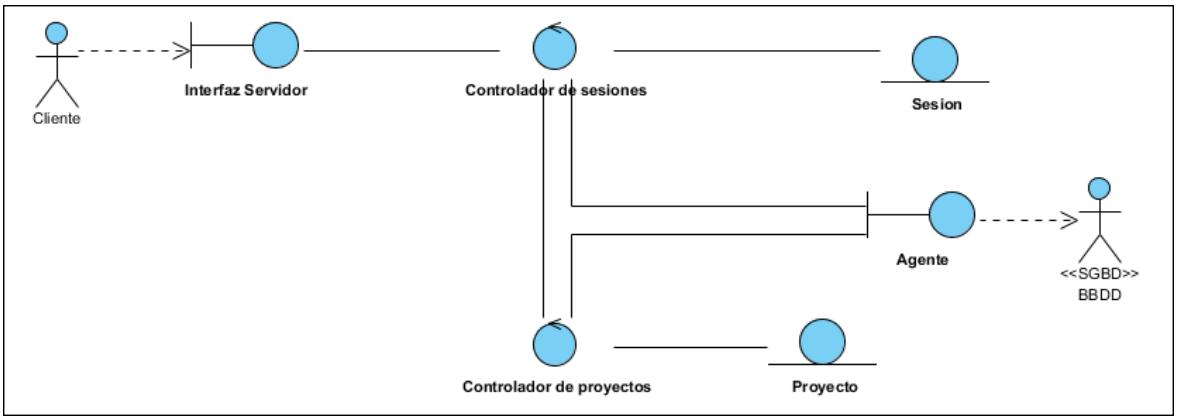

Figura 5.33: Diagrama de clases de análisis - Servidor - Login

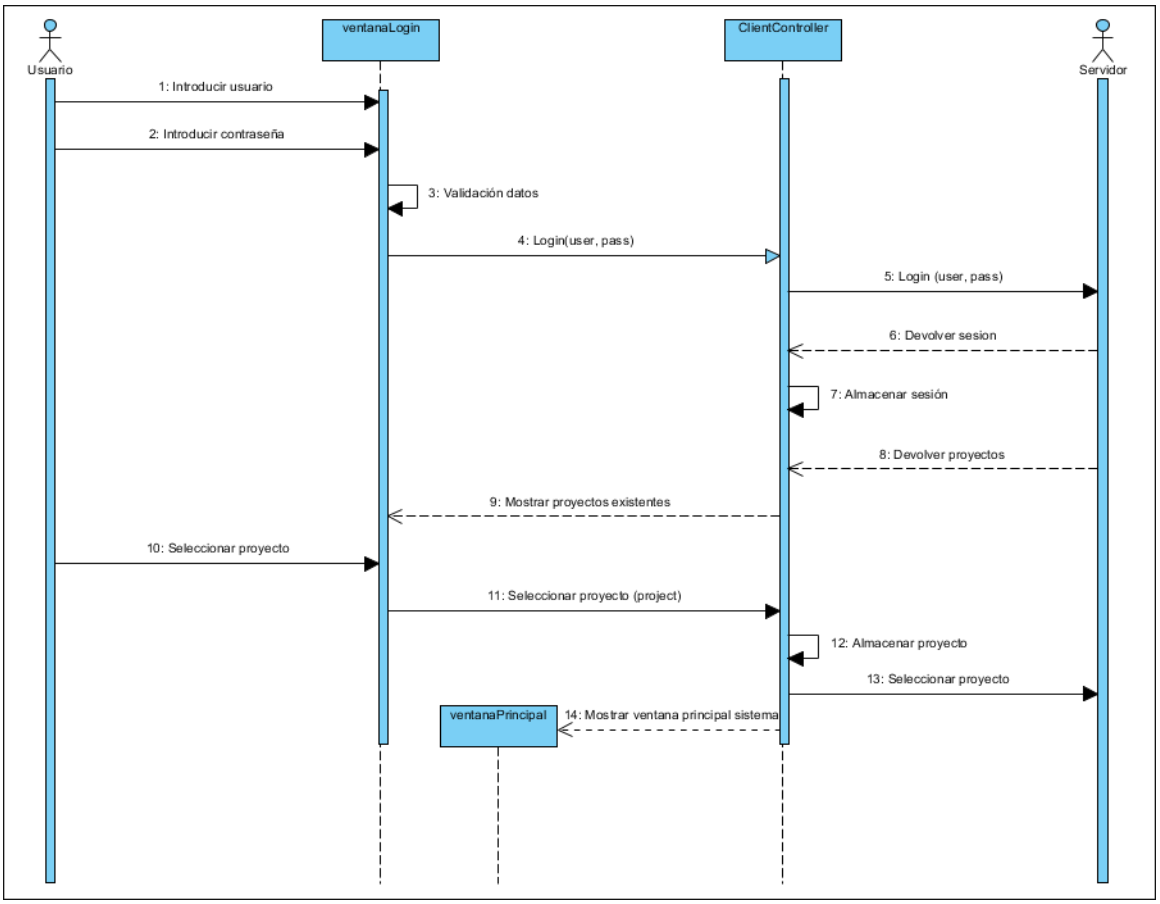

Figura 5.34: Diagrama de secuencia - Cliente - Login

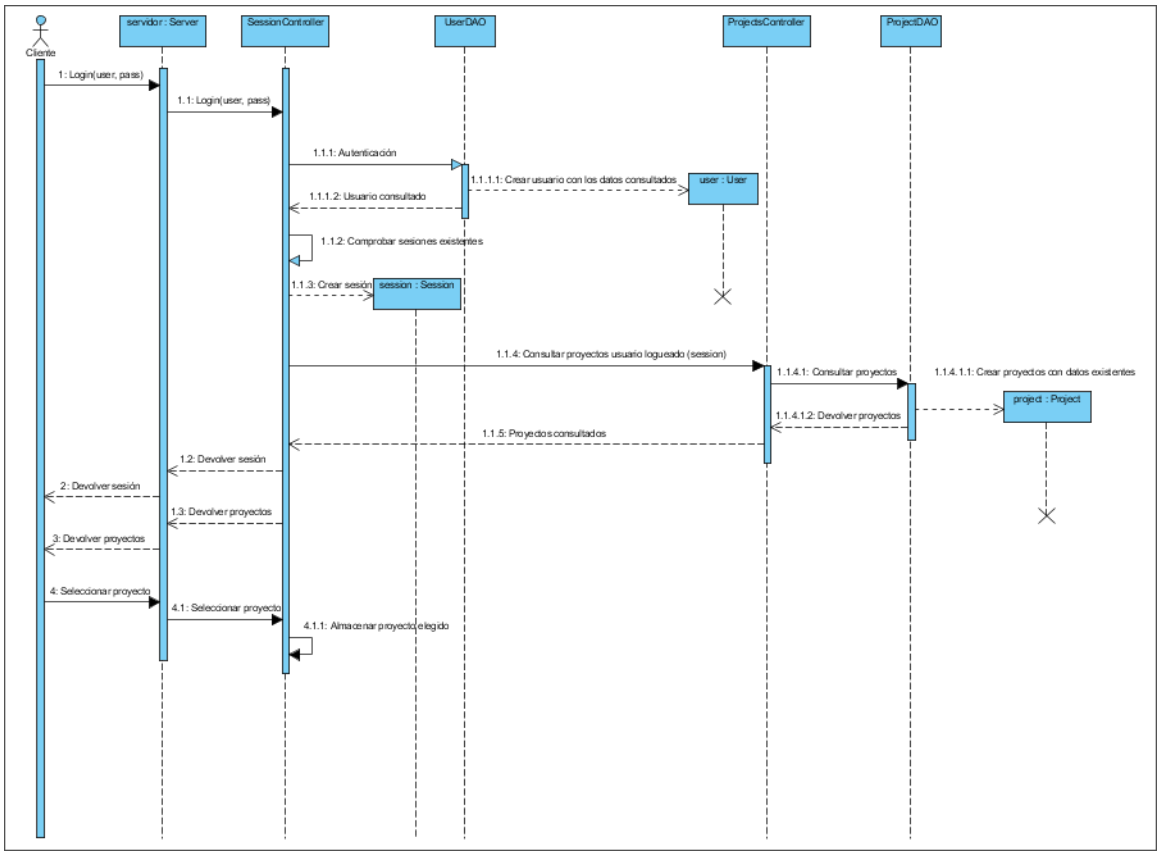

Figura 5.35: Diagrama de secuencia - Servidor - Login

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 sistema, según la especificación de requisitos (ver sección 5.1.1.1), como se observa en el diagrama de clases de la Figura 5.26.

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 4.2).

En la Figura 5.36 se muestra el diagrama de clases para el gestor de sesiones.

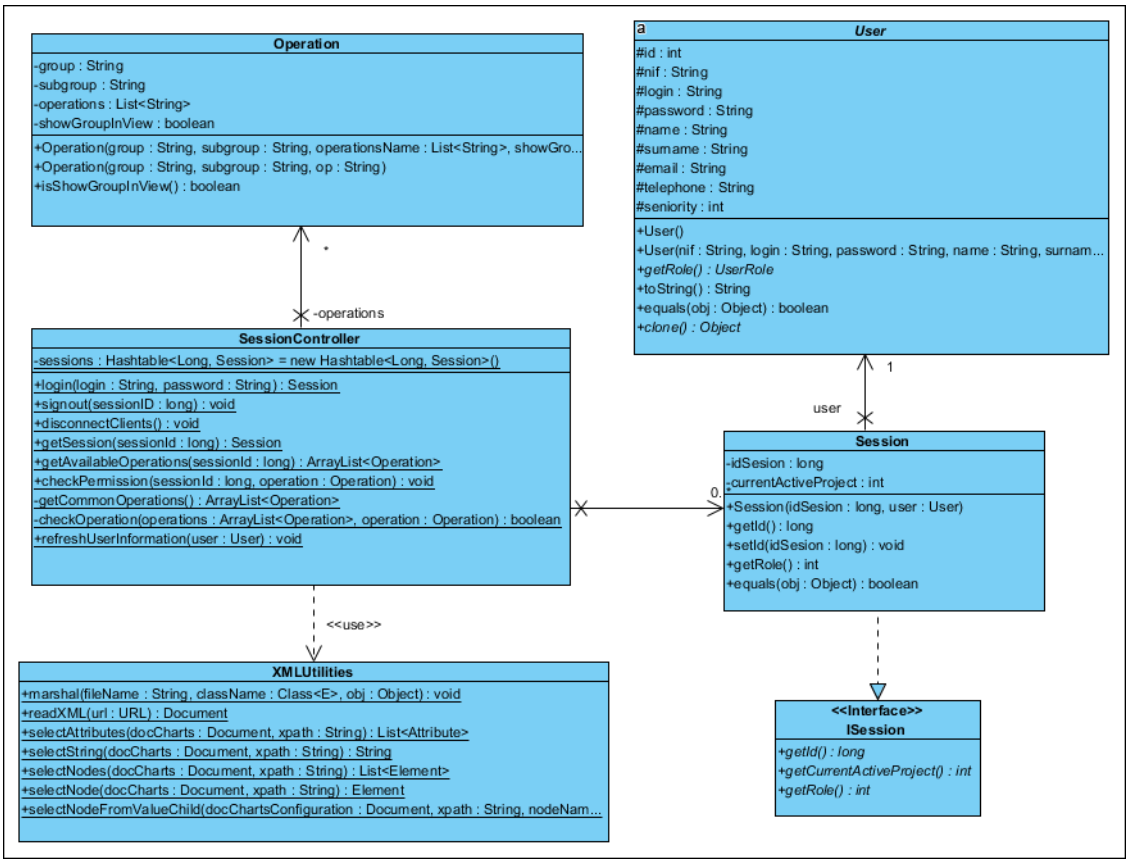

Figura 5.36: Diagrama de clases - Gestor de sesiones

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 4.2) 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 5.37) animado, para proporcionar al usuario un *feedback* visual y saber que la operación se está realizando y que debe esperar.

Figura 5.37: Ejemplo de *spinner* de carga

En el fragmento de código 5.3 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 5.3: Fragmento de código para acceder al sistema en el cliente

5.3.1.2 Pruebas

Al final de esta iteración se diseñan y ejecutan las pruebas unitarias correspondientes a esta funcionalidad del sistema. Para ello, al igual que en la iteración anterior, se diseñan los casos de prueba para los diferentes escenarios de los casos de uso que componen esta funcionalidad y se ejecutan con JUnit. Además, en el subsistema cliente, se elabora una *checklist* para asegurase que el comportamiento de los diferentes elementos que componen la interfaz gráfica de usuario es el esperado y adecuado.

5.3.2 Iteración 4

Siguiendo los casos de uso del grupo funcional **F3:** *Visualización información* (ver Figura 5.12), 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.

5.3.2.1 Grupo funcional F3: *Visualización información*

5.3.2.1.1 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 5.11 se describe el caso de uso *Visualizar decisiones*.

La Figura 5.38 representa el diagrama de clases de análisis para el subsistema cliente. En la Figura 5.39 se muestra el diagrama de clases de análisis para el subsistema servidor.

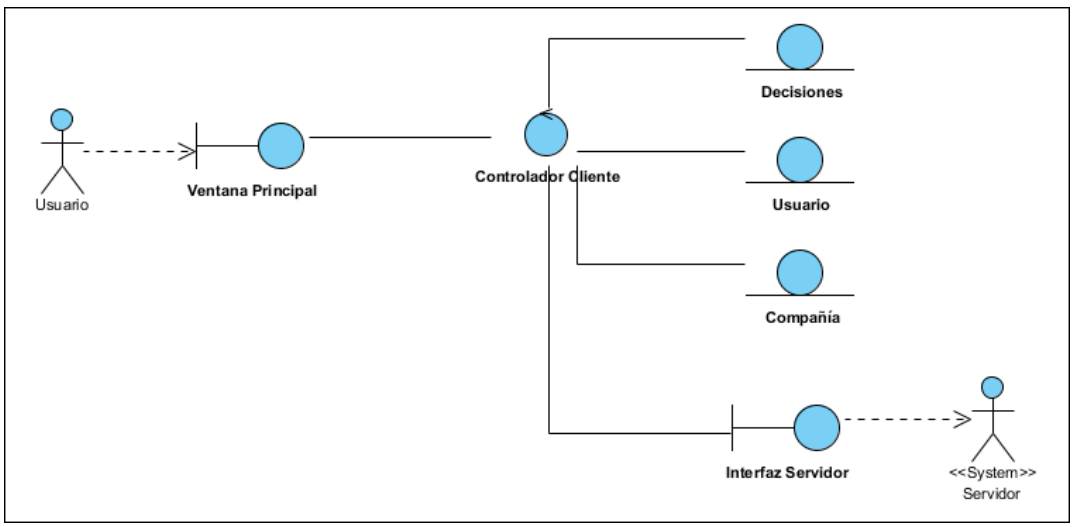

Figura 5.38: Diagrama de clases de análisis - Cliente - Visualizar Decisiones

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 5.11: Especificación del caso de uso *Visualizar decisiones*

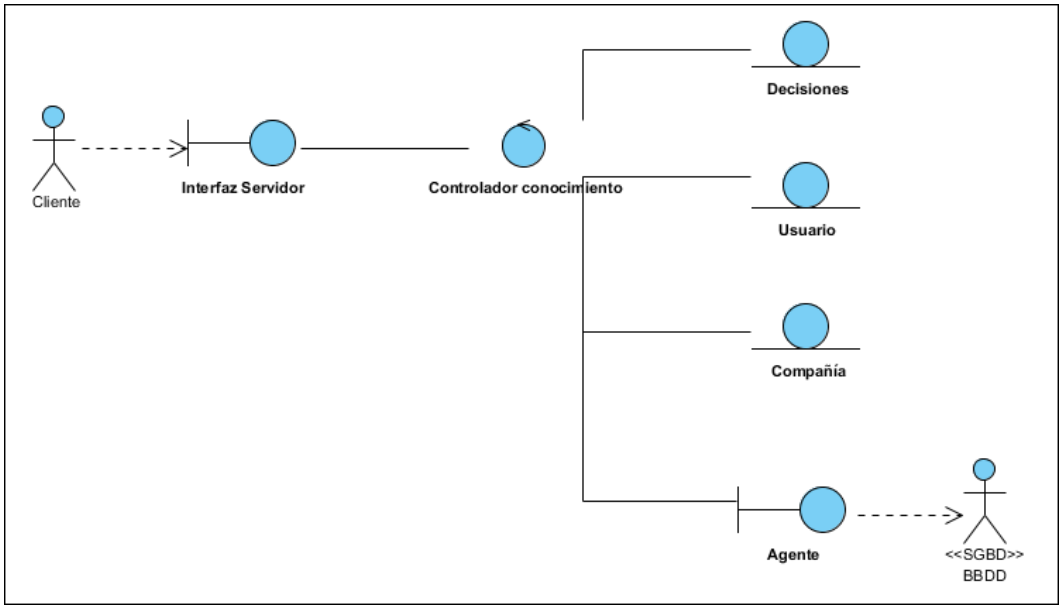

Figura 5.39: Diagrama de clases de análisis - Servidor - Visualizar Decisiones

5.3.2.1.2 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 5.35 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.

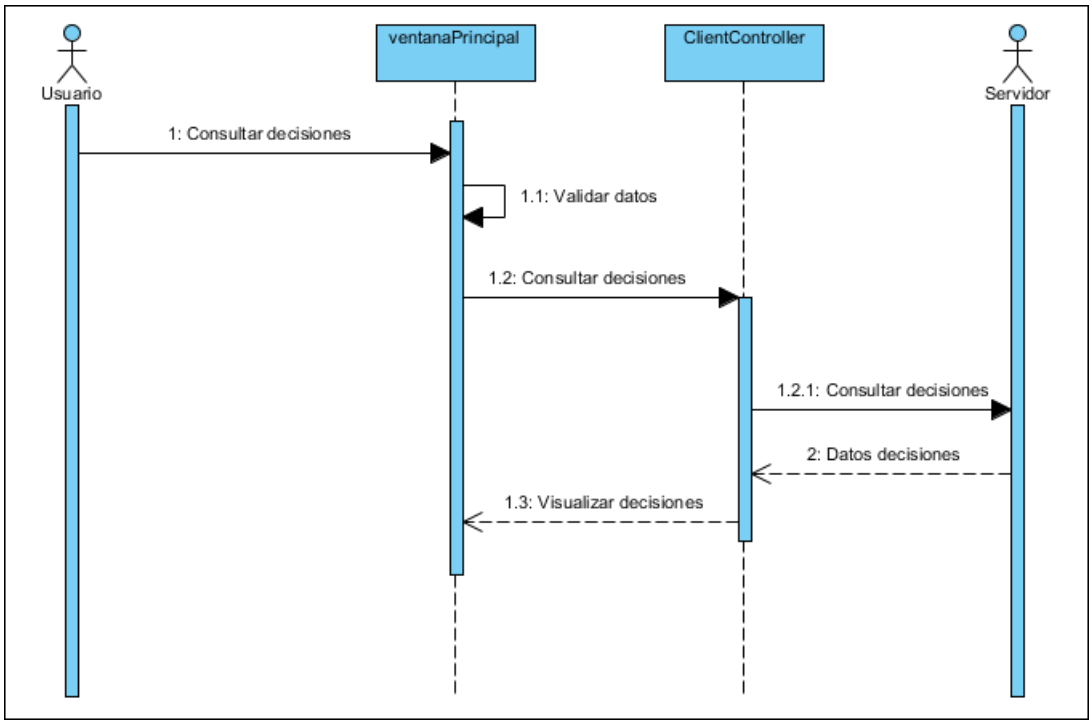

Figura 5.40: Diagrama de secuencia - Cliente - Consultar decisiones

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

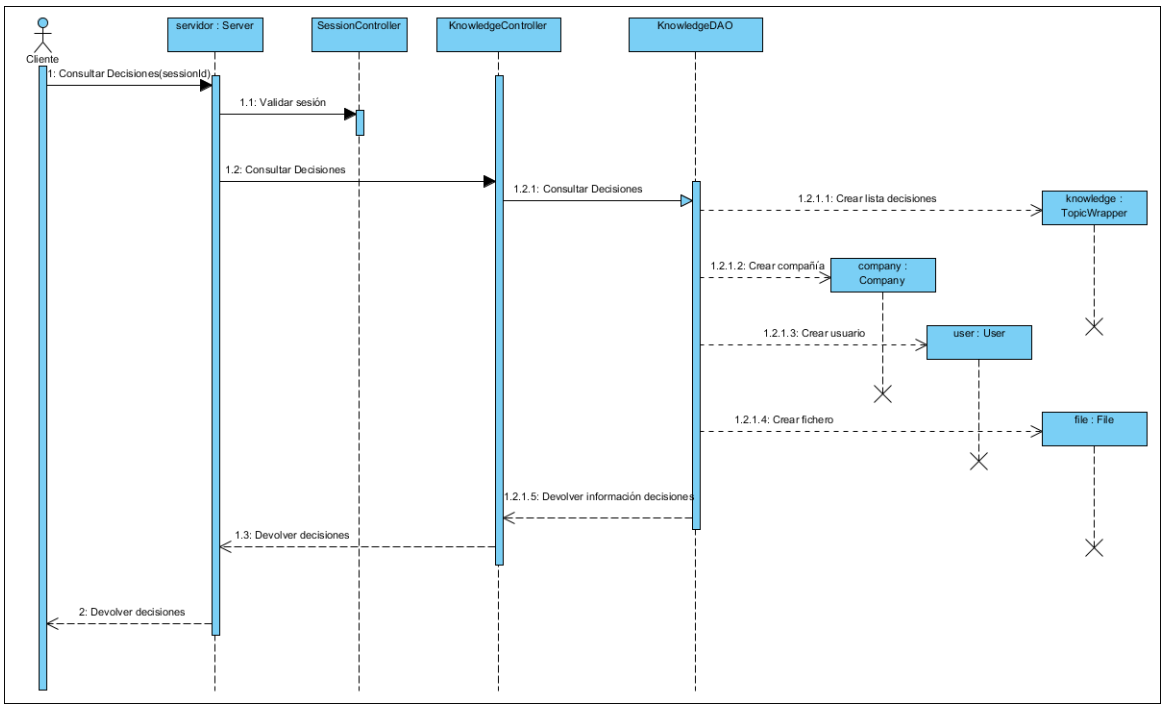

Figura 5.41: Diagrama de secuencia - Servidor - Consultar decisiones

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 5.36).

Como se puede apreciar en el diagrama de clases de la Figura 5.42, 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 5.42, 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 composición y jerarquía de conocimiento. Un ejemplo de esta estructura jerárquica puede encontrarse en los foros de debate presentes en la Web (ver Figura 5.43).

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 5.3.6.1.

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 5.42). 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 5.41, 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 4.2, 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

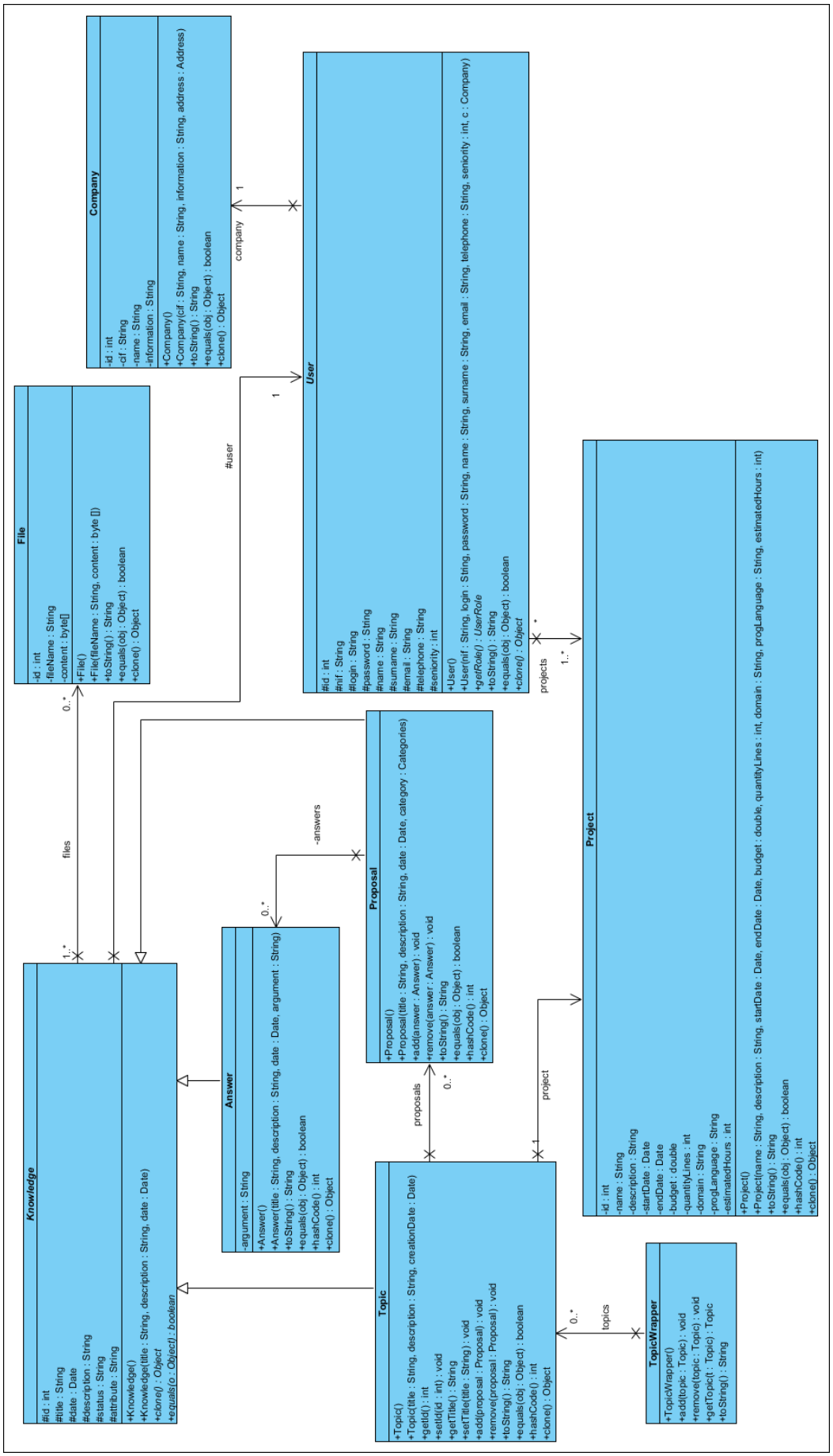

Figura 5.42: Diagrama de clases - Jerarquía de decisiones

Figura 5.43: Ejemplo de jerarquía de decisiones en foros de debates

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 5.4. 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 5.4: Respuesta del servicio Web Yahoo! PlaceFinder

En el Listado 5.5 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 5.5: 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 4.2). 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 3.2. 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 4.2), 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 5.6 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 5.6: Fragmento de código para mostrar mapas geoposicionados

5.3.2.1.3 Pruebas

Al igual que en iteraciones anteriores, se diseñan los casos de prueba para los diferentes escenarios de los casos de uso que componen esta funcionalidad y se ejecutan con JUnit. Además, en el subsistema cliente, se elaboran diferentes *checklist* para asegurase que el comportamiento de los diferentes elementos que componen la interfaz gráfica de usuario es el esperado y adecuado. A modo de ejemplo, en la Tabla **??** se muestra una *checklist* para comprobar la correcta visualización de las decisiones y de su información.

5.3.3 Iteración 5

Siguiendo los casos de uso del grupo funcional **F2:** *Gestión de decisiones* (ver Figura 5.10) y del grupo funcional **F4:** *Gestión de notificaciones* (ver Figura 5.14), 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.

5.3.3.1 Grupo funcional F2: *Gestión de decisiones*

5.3.3.1.1 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 5.10). 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 5.12 se describe el caso de uso *Modificar Decisión - Modificar Propuesta*.

En la Figura 5.44 se muestra el diagrama de clases de análisis para el subsistema cliente. En la Figura 5.45 se muestra el diagrama de clases de análisis para el subsistema servidor.

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.

Aceptar o rechazar decisión

Este caso de uso se ha modelado como una extensión al caso de uso anterior, *Modificar Decisión* (ver Figura 5.10). Esto es así porque, aunque también es una modificación de una

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 5.12: Especificación del caso de uso *Modificar decisión - Modificar Propuesta*

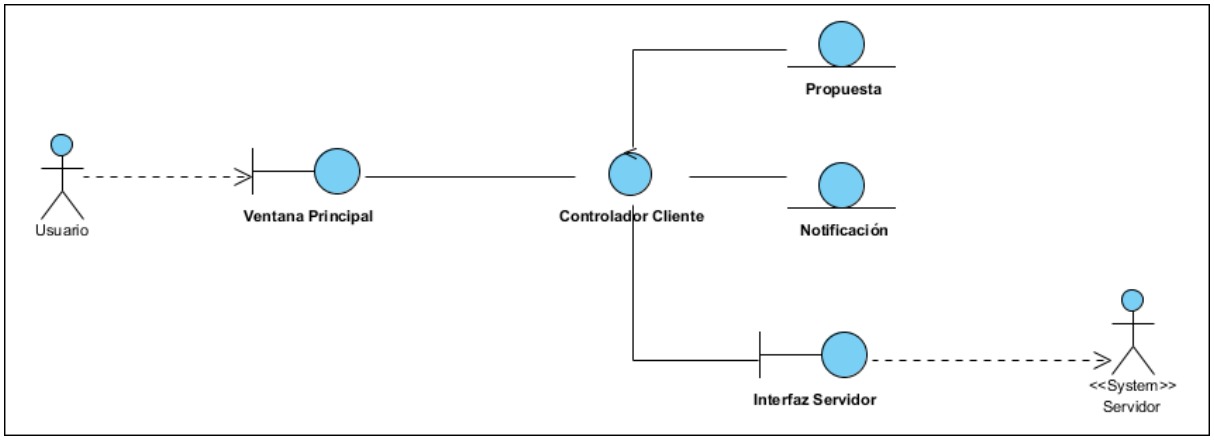

Figura 5.44: Diagrama de clases de análisis - Cliente - Modificar Propuesta

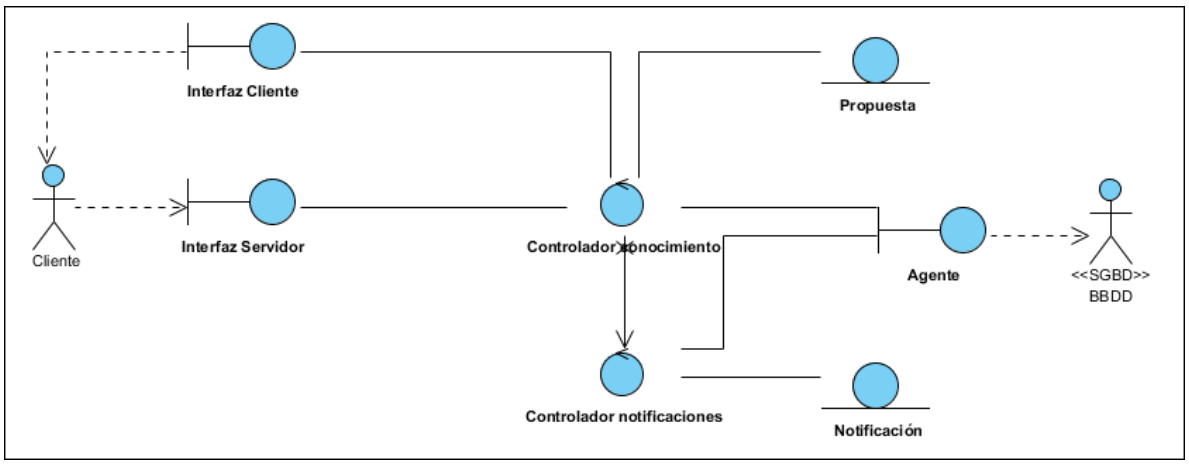

Figura 5.45: Diagrama de clases de análisis - Servidor - Modificar Propuesta

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 5.13 se describe el caso de uso *Aceptar o rechazar decisión*.

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 5.13: Especificación del 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.

5.3.3.1.2 Diseño e implementación

En las Figuras 5.46 se muestra los diagramas de secuencia para el caso de uso *Crear decisión - Crear Propuesta* en el subsistema cliente, mientras que la Figura 5.47 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.

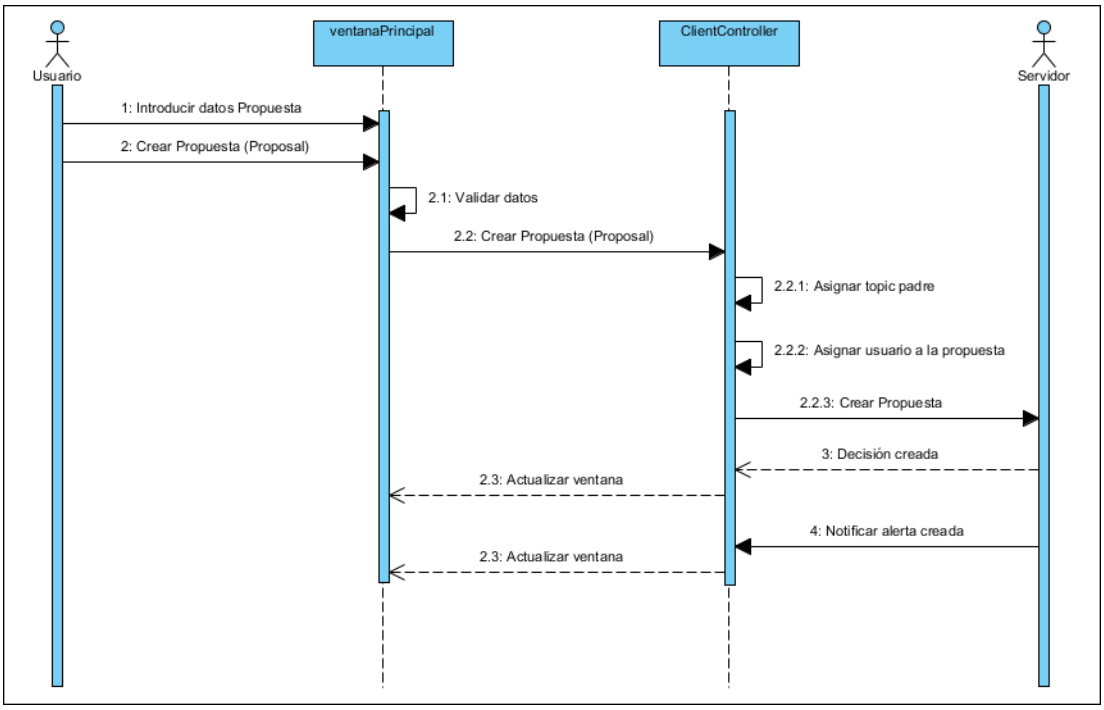

Figura 5.46: Diagrama de secuencia - Cliente - Crear decisión (*Topic*)

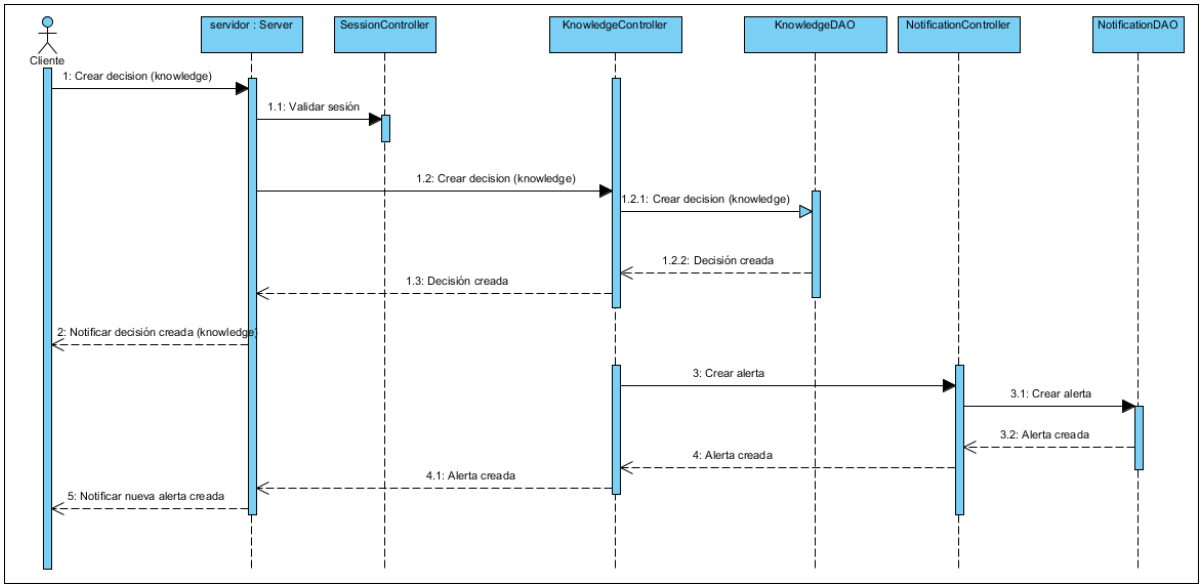

Figura 5.47: Diagrama de secuencia - Servidor - Crear decisión

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 5.48 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

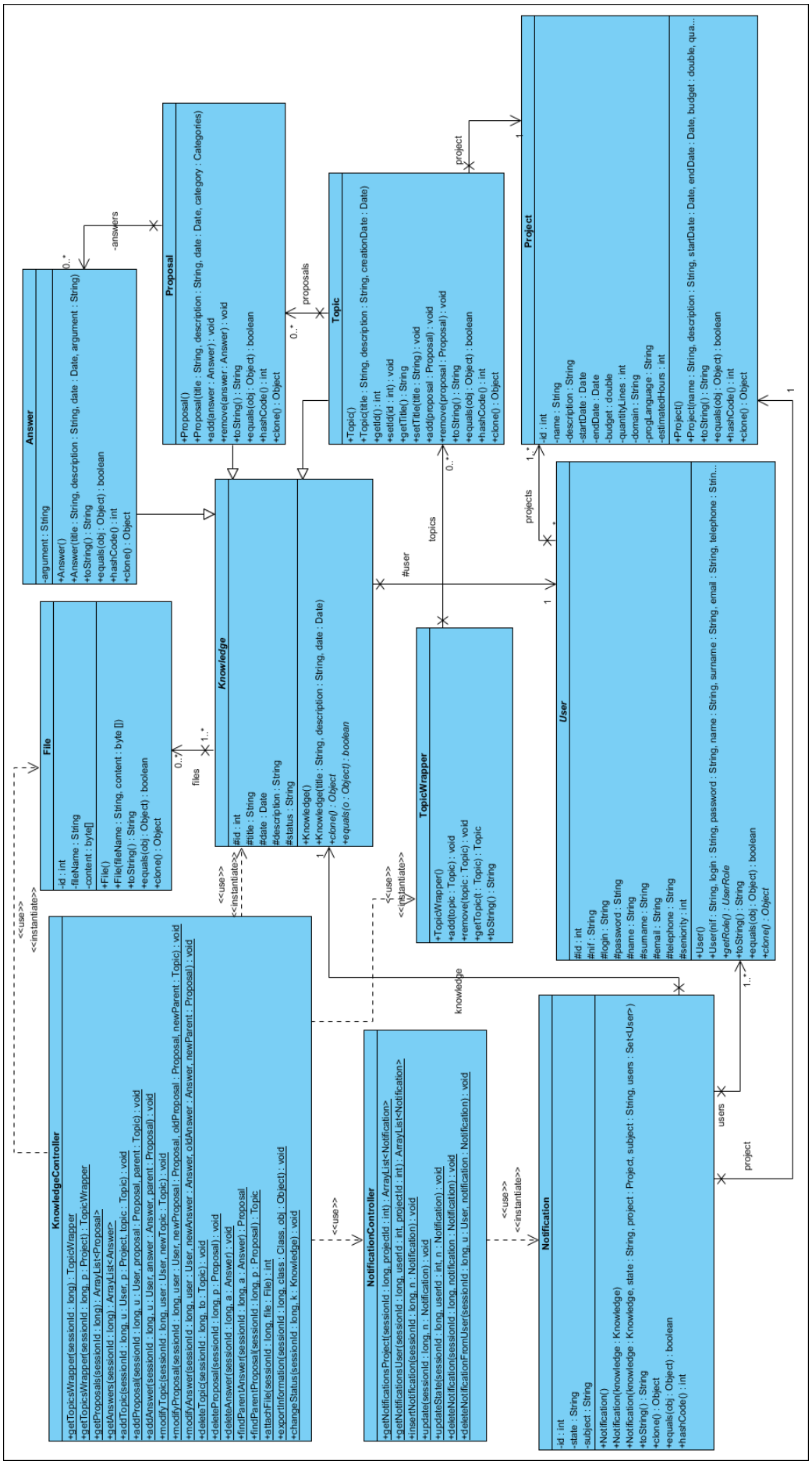

Figura 5.48: Diagrama de clases - Gestión de decisiones

actualizar su estado. En el diagrama de la Figura 5.49 se muestra este patrón y las clases que lo forman.

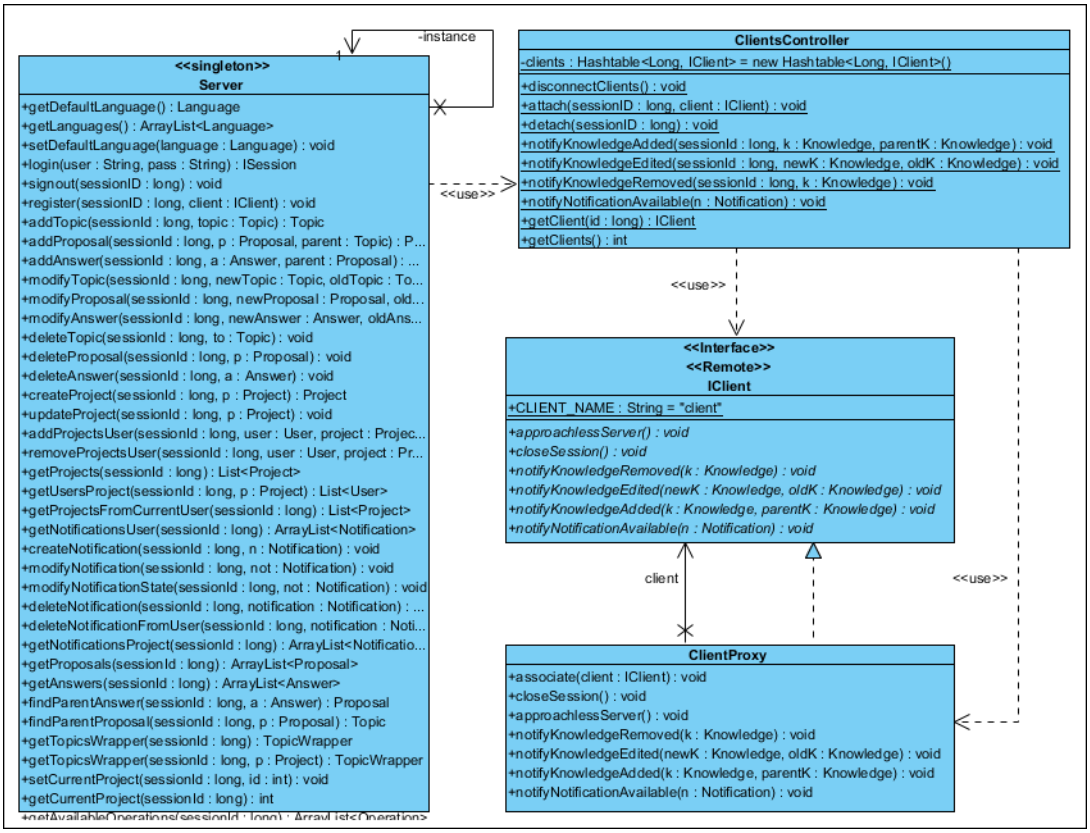

Figura 5.49: Diagrama de clases - Observador para actualizar clientes conectados

En el fragmento de código 5.7 se muestra parte de la implementación de la clase controlador de los clientes, y en el fragmento 5.8 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
      for(Long id : clients.keySet())
         if (id != sessionId)
            clients.get(id).notifyKnowledgeAdded(k, parentK);
   }

   ....
}
```

Listado 5.7: 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 5.8: 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 5.9 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 5.9: 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.

5.3.3.1.3 Pruebas

Para concluir el desarrollo de esta iteración, se diseñan e implementan los casos de prueba unitarios para este grupo funcional del sistema. De este modo, para el servidor, se crean casos de pruebas encargados de probar las diferentes operaciones relacionadas con la gestión de decisiones, como su creación, modificación, eliminación, notificación a los clientes, etc. En dichos casos de prueba se prueban escenarios correctos e incorrectos de los casos de uso, comprobando que se obtiene el resultado esperado en cada caso escenario. En la Figura **??** se observa una captura de pantalla de los diferentes casos de prueba para las operación de gestión de decisiones.

En el caso del cliente, como en el resto de casos, se crean *checklist* para asegurar el correcto funcionamiento de la interfaz gráfica.

5.3.3.2 Grupo funcional F4: *Gestión de notificaciones*

5.3.3.2.1 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 5.3.3.1.2, 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 5.48. 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 5.27.

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 5.10. 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 5.10: 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.

5.3.3.2.2 Pruebas

Al igual que en el caso anterior, se diseñan e implementan los casos de prueba unitarios, así como *checklists*.

5.3.4 Iteración 6

Siguiendo los casos de uso del grupo funcional **F5:** *Gestión de proyectos* (ver Figura 5.16), 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.

5.3.4.1 Grupo Funcional F5: *Gestión de proyectos*

5.3.4.1.1 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.

5.3.4.1.2 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 5.50, mientras que en la Figura 5.51 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.

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 5.52 se muestra el diagrama de secuencia para este caso de uso en el cliente, mientras que en la Figura 5.53 se observa el diagrama de secuencia para el servidor.

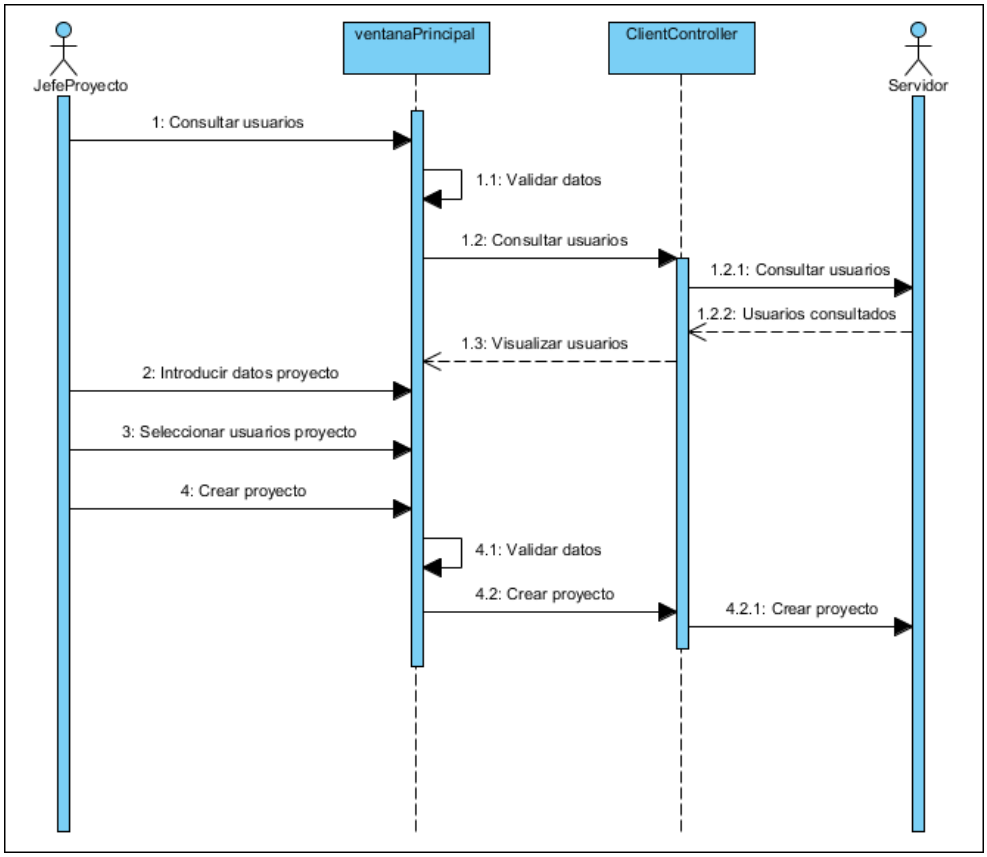

Figura 5.50: Diagrama de secuencia - Cliente - Crear proyecto

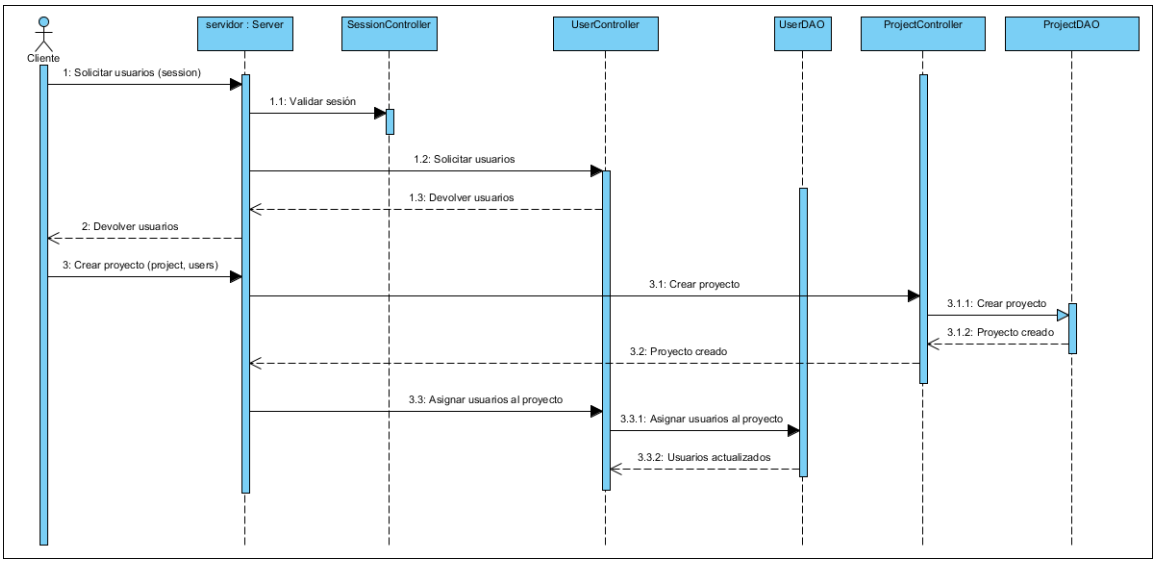

Figura 5.51: Diagrama de secuencia - Servidor - Crear proyecto

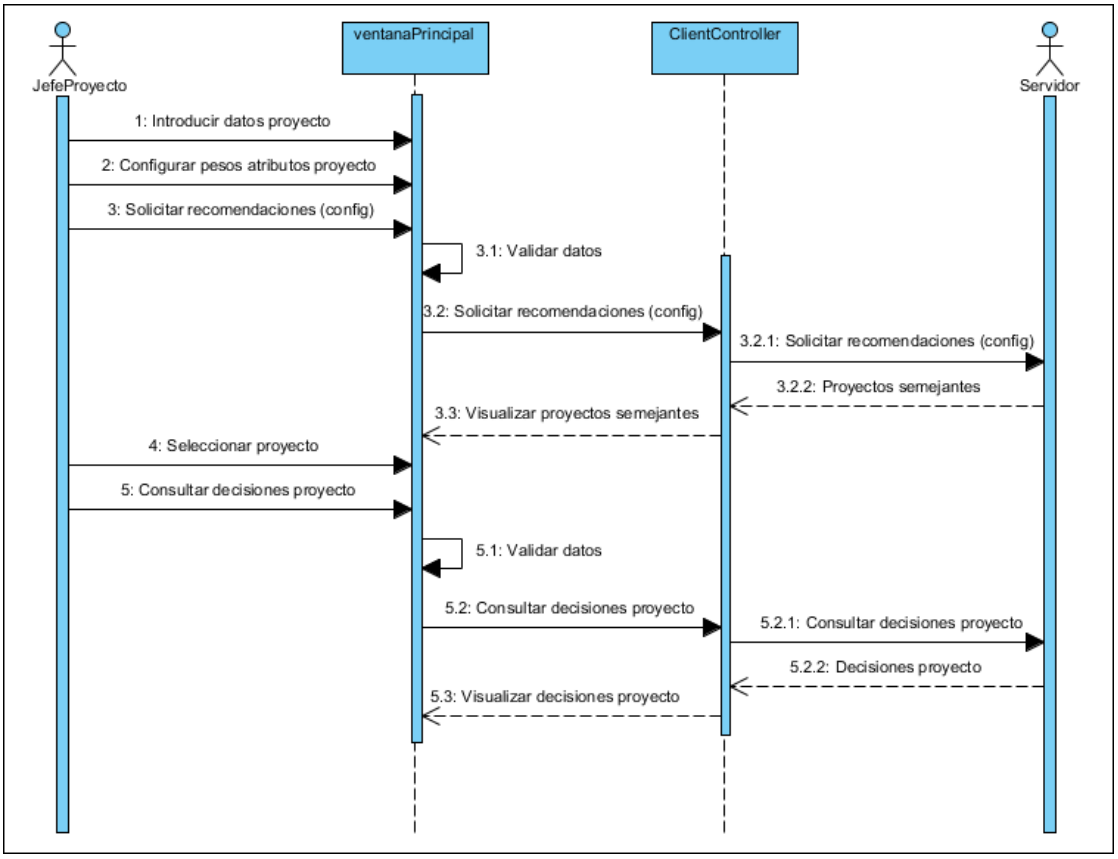

Figura 5.52: Diagrama de secuencia - Cliente - Aconsejar decisiones

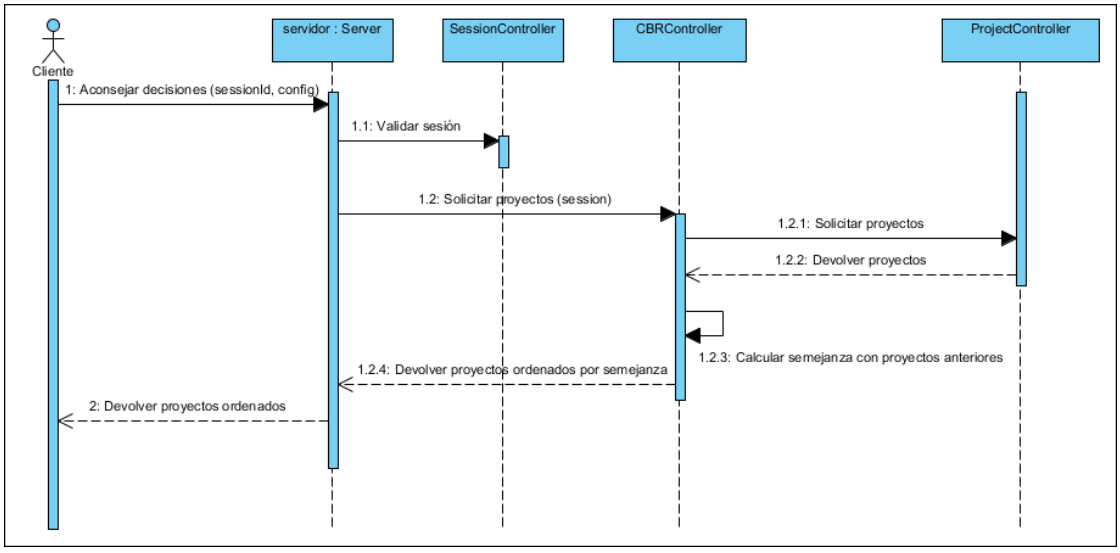

Figura 5.53: Diagrama de secuencia - Servidor - Aconsejar decisiones

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.

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 5.54 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.

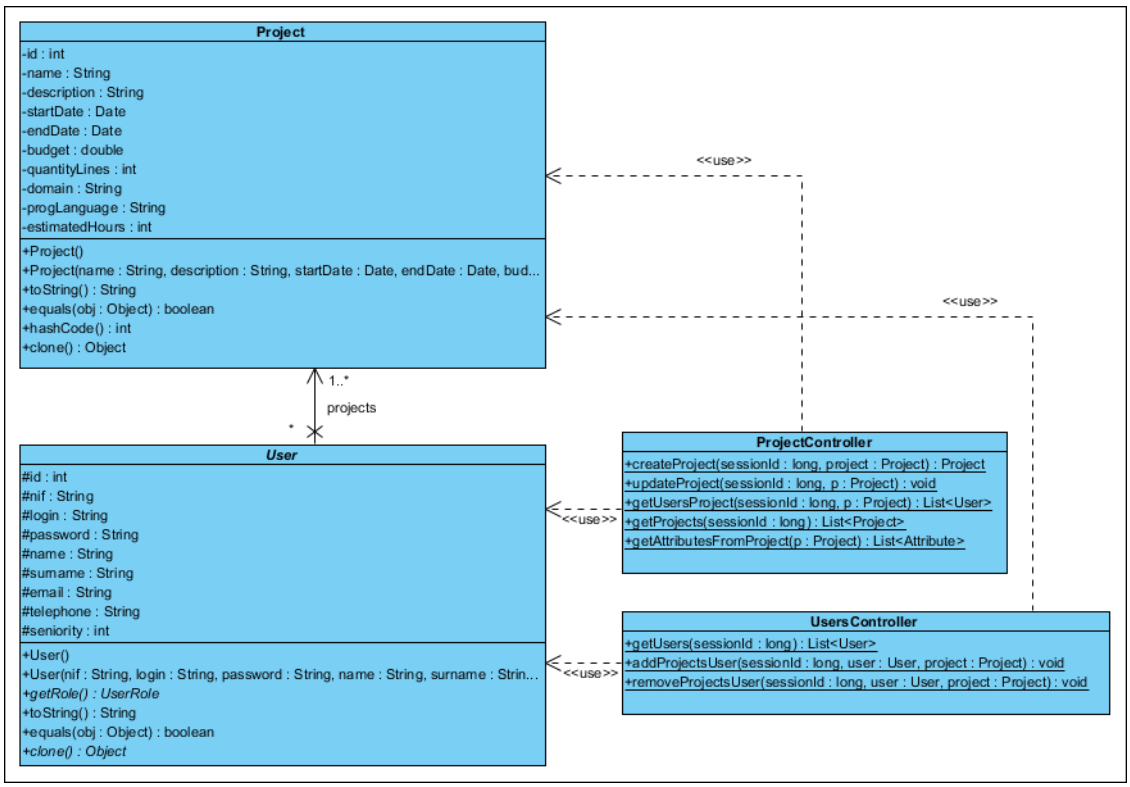

Figura 5.54: Diagrama de secuencia - Servidor - Controlador proyectos

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 3.3.

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 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 3.3. 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 5.1 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 5.2 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.

$$sem(c1, c2) = \sum_{i=1}^{n}(w_i * sem(att1_i, att2_i))/\sum_{i=1}^{n} w_i \tag{5.1}$$

$$sem(c1, c2) = \sum_{i=1}^{n}(w_i * sem(att1_i - att2_i)^2) \tag{5.2}$$

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 5.55 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 5.11 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
   {
      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)));
      }
```

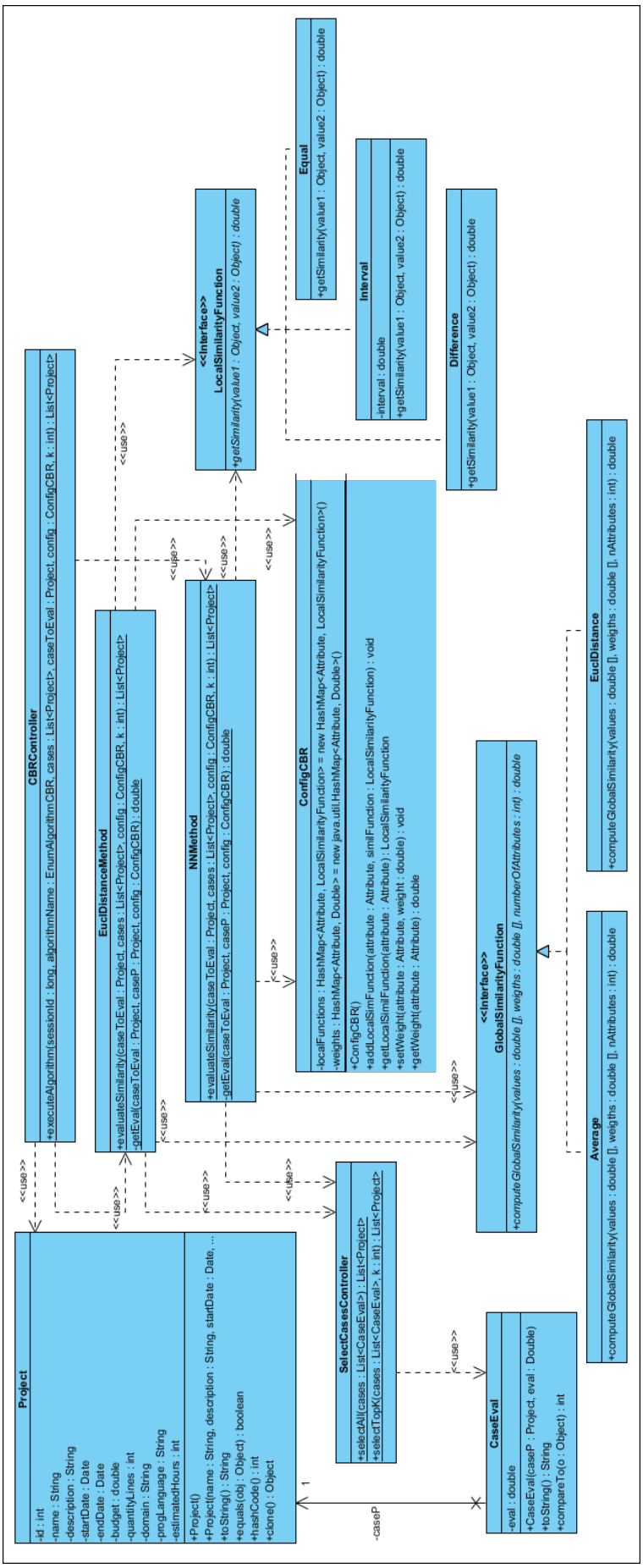

Figura 5.55: Diagrama de clases - Servidor - Razonamiento Basado en Casos

```
    // 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 5.11: 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.

5.3.4.1.3 Pruebas

5.3.5 Iteración 7

Siguiendo los casos de uso del grupo funcional **F6:** *Generación de informes* (ver Figura 5.19) y de grupo funcional **F7:** *Generación de estadísticas* (ver Figura 5.17), 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.

5.3.5.1 Grupo funcional F6: *Generación de informes*

5.3.5.1.1 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.

5.3.5.1.2 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 5.56 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 5.57 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.

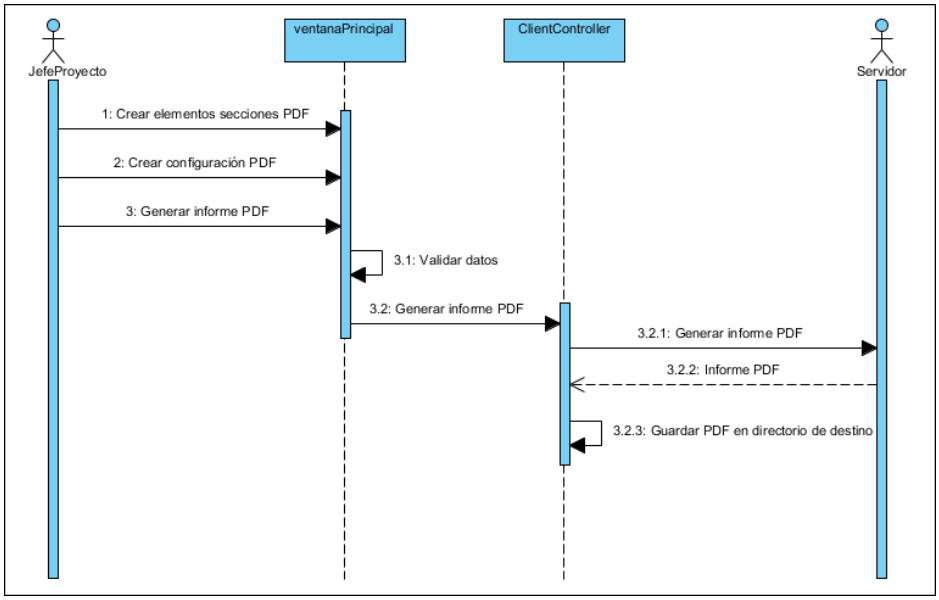

Figura 5.56: Diagrama de secuencia - Cliente - Generar informe

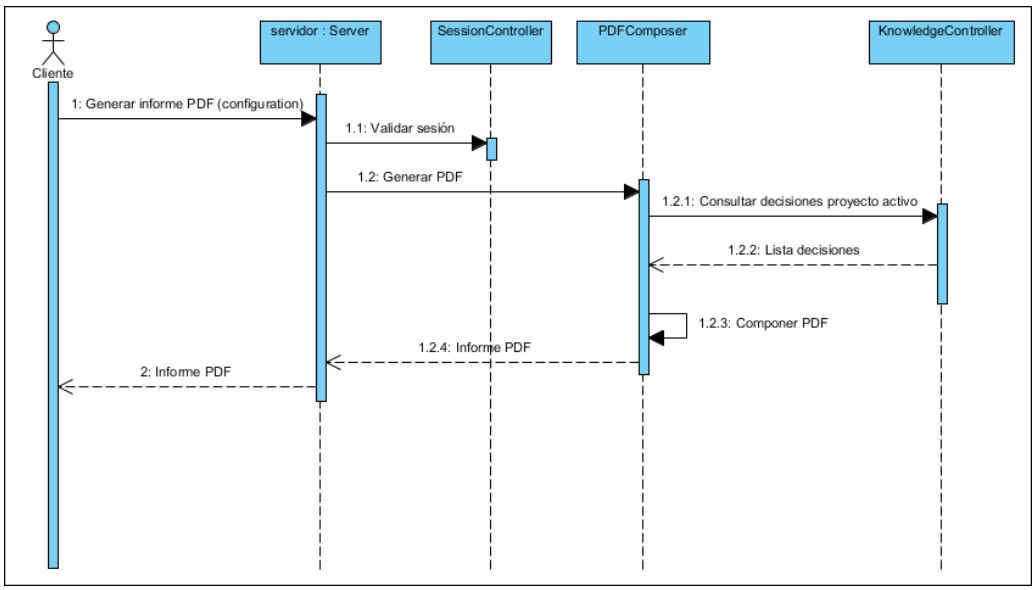

Figura 5.57: Diagrama de secuencia - Cliente - Generar informe

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 5.58 se muestra el diagrama de clases de diseño para la funcionalidad de generación de informes en PDF.

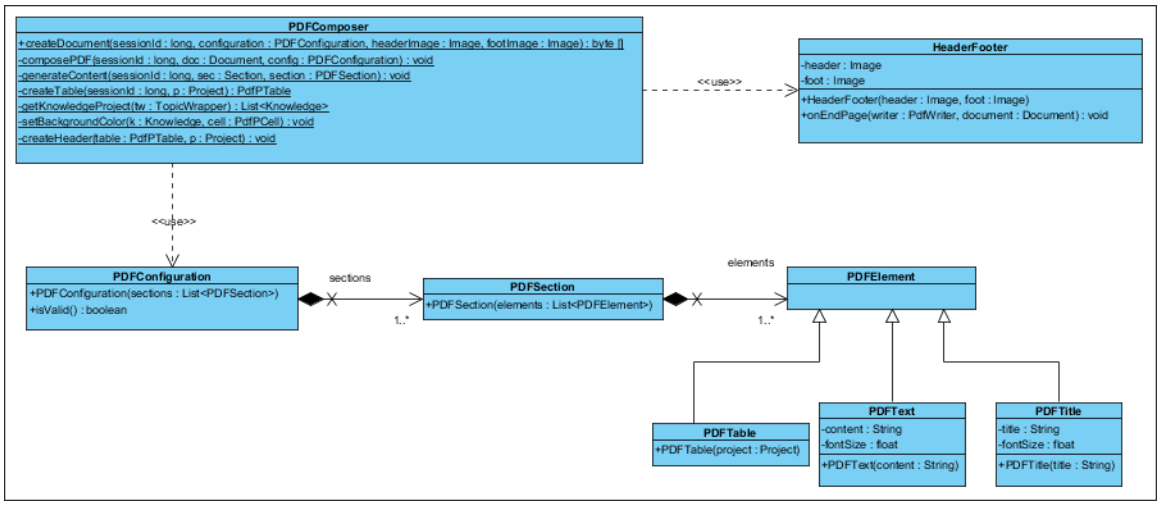

Figura 5.58: Diagrama de clases - Generación de documentos PDF - Servidor

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:

- 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 5.58, 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 4.2) 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 5.12 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 5.12: 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 5.59 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 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*.

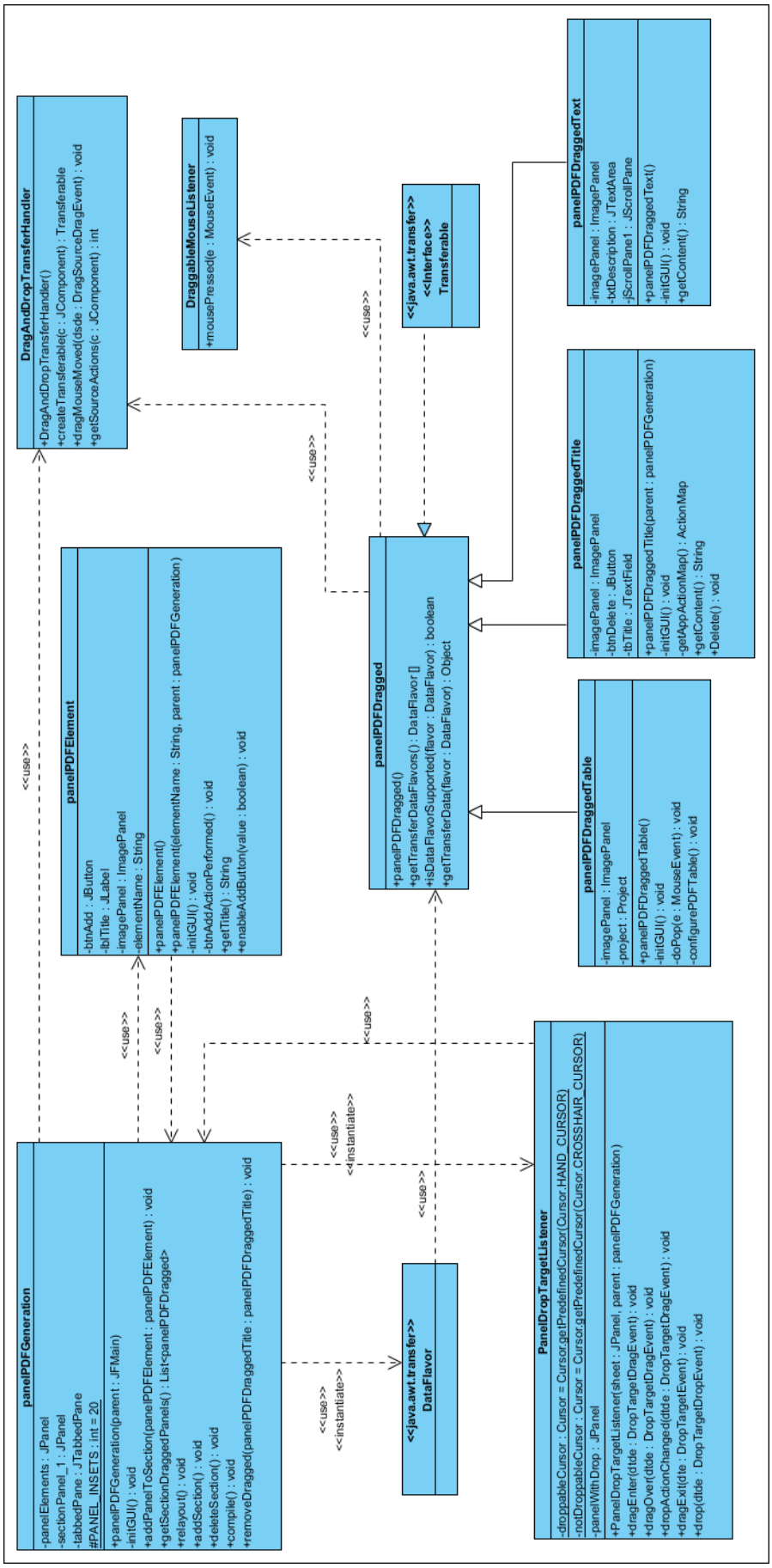

Figura 5.59: Diagrama de clases - Generación de documentos PDF - Cliente

En la Figura 5.60 se muestra visualmente el proceso de *Drag & Drop*, de manera simplificada.

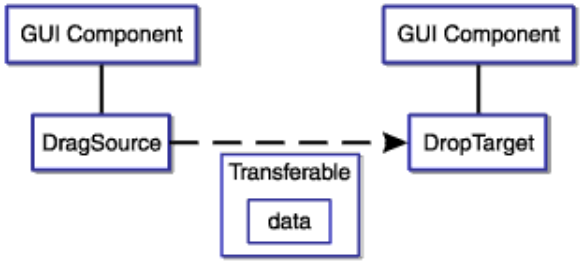

Figura 5.60: Proceso simplificado de *Drag & Drop*

5.3.5.1.3 Pruebas

5.3.5.2 Grupo funcional F7: *Generación de estadísticas*

5.3.5.2.1 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 5.14 se describe el caso de uso *Generación gráficos estadísticos*.

La Figura 5.61 representa el diagrama de clases de análisis para el subsistema cliente.

5.3.5.2.2 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.

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 5.14: Especificación del caso de uso *Generación gráfico estadístico*

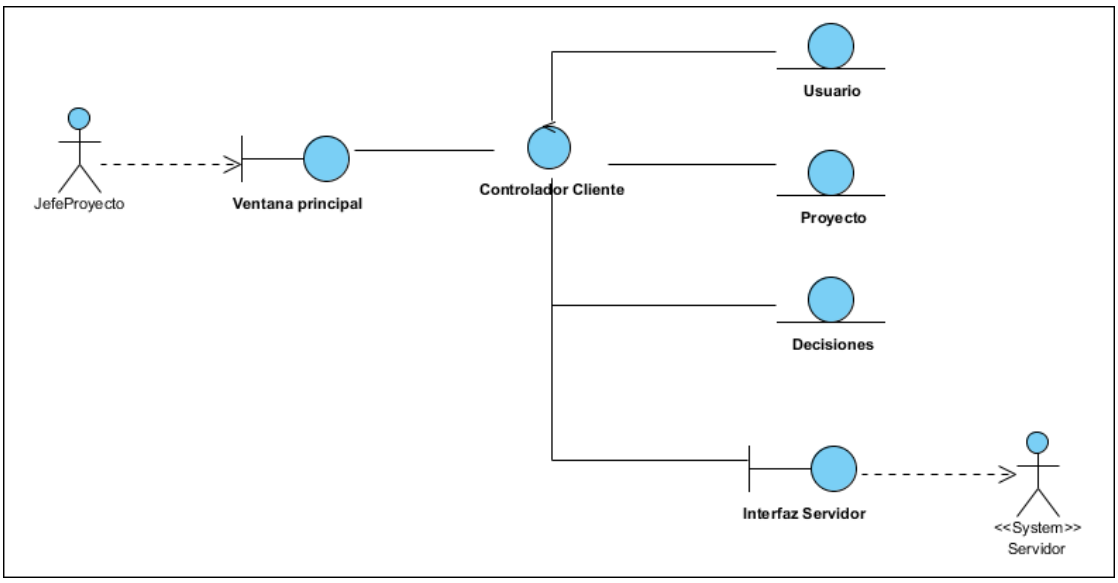

Figura 5.61: Diagrama de clases de análisis - Cliente - Generación gráficos estadísticos

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 4.2). 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.

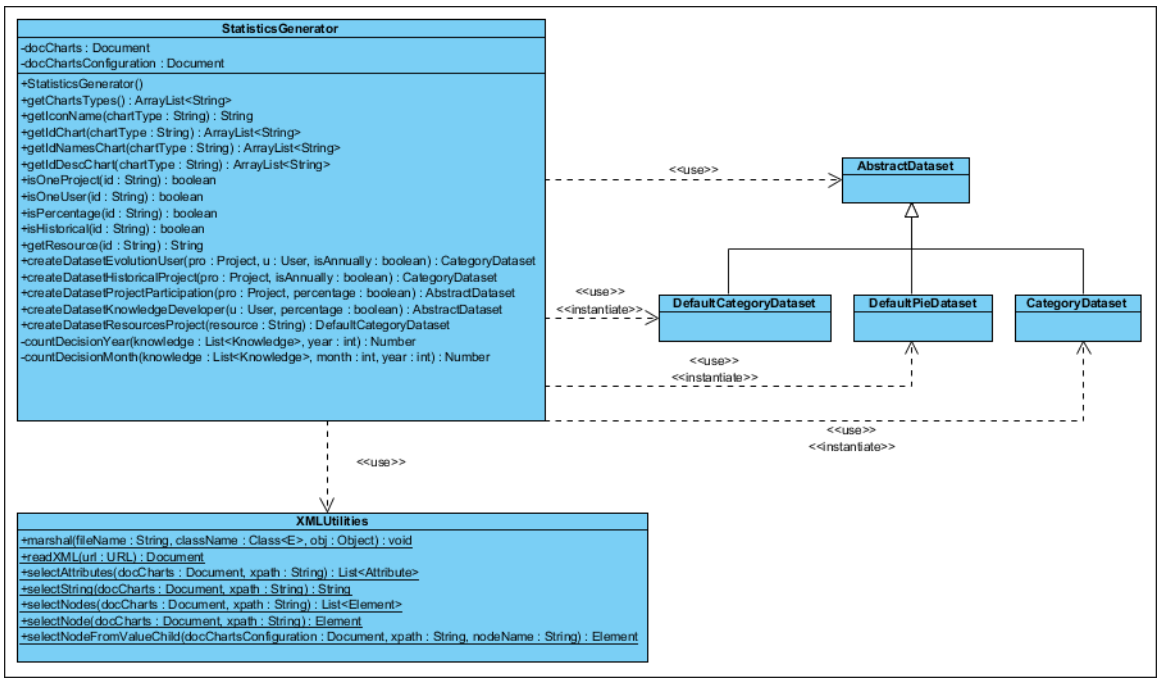

Figura 5.62: Diagrama de clases - Cliente - Generar estadísticas

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 5.13 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 5.13: Fragmento de código para la generación de *datasets*

Para terminar, en el fragmento de código 5.14 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 5.14: Fragmento de código para la generación de gráficos

5.3.5.2.3 Pruebas

5.3.6 Iteración 8

Siguiendo los casos del grupo funcional **F8:** *Exportar información* (ver Figura 5.22) 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.

5.3.6.1 Grupo funcional F8: *Exportar conocimiento*

5.3.6.1.1 Análisis de casos de uso

En esta etapa se aborda el análisis de los casos de uso de esta funcionalidad, mostrados en la Figura 5.22. 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 5.15 se describe el caso de uso *Exportar información*.

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 5.15: Especificación del caso de uso *Exportar información*

La Figura 5.63 representa el diagrama de clases de análisis para el subsistema cliente. La Figura 5.64 refleja el diagrama de clases de análisis para el subsistema servidor.

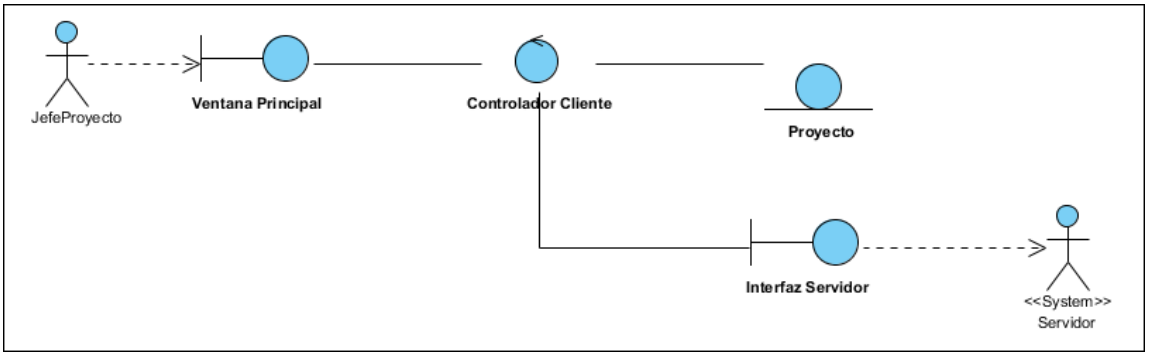

Figura 5.63: Diagrama de clases de análisis - Cliente - Exportar información

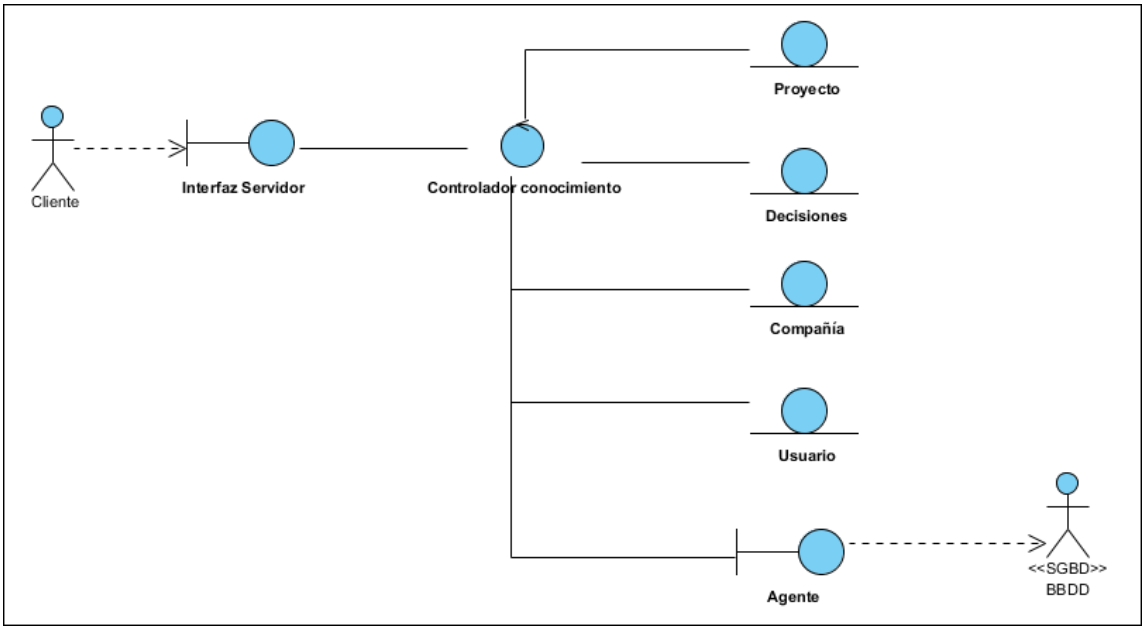

Figura 5.64: Diagrama de clases de análisis - Servidor - Exportar información

5.3.6.1.2 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.

Servidor

En el diseño e implementación de esta funcionalidad, cabe destacar la utilización de **JAXB** (ver sección 4.2) para serializar las clases que componen la jerarquía de decisiones y toda su información relacionada (ver Figura 5.42) 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 5.15 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 5.16 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 5.42.

```
@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 5.15: 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 5.16: 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 5.17 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 5.17: 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.

5.3.6.1.3 Pruebas

5.3.6.2 Grupo funcional F9: *Gestión de idiomas*

5.3.6.2.1 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.

5.3.6.2.2 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).

5.3.6.2.3 Pruebas

5.4 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.

Capítulo 6

Consecución de Objetivos

El objetivo principal del PFC era desarrollar una aplicación que facilite y permita la gestión de decisiones tomadas en proyectos software, así como la reutilización de las mismas para aprovechar la experiencia obtenida en proyectos previos, aplicada al ámbito del Desarrollo Global de Software.

Consideramos que este objetivo ha sido conseguido gracias al diseño y construcción de la aplicación, basada en Java, que ha sido detallada y comentada a lo largo del documento, especialmente en el capítulo 5.

A continuación, se explica la consecución del resto de sub-objetivos planteados en el capítulo 2, y resumidos en la Tabla2.1.

6.1 O1: Acceso desde diferentes localizaciones

La aplicación desarrollada ha sido diseñada siguiendo una arquitectura cliente-servidor, donde el servidor centraliza la lógica de dominio y control del sistema, y el cliente presenta la interfaz gráfica de usuario, que realiza peticiones al servidor y muestra los resultados relevantes para el usuario del sistema. Por ello, y gracias a esta arquitectura definida, se puede utilizar la aplicación cliente de manera distribuida y, por tanto, desde diferente ubicaciones.

Además, la aplicación cuenta con una funcionalidad de *login*, permitiendo el acceso a los usuarios de la misma. Esto corresponde al grupo funcional **F1** (ver Tabla 5.5 en capítulo 5).

6.2 O2: Facilitar y favorecer la gestión de decisiones

Para facilitar la gestión de decisiones tomadas en proyectos software, se han diseñado e implementado funcionalidades para la creación de decisiones, para su modificación, eliminación, etc. Por tanto, el sistema provee un mecanismo que facilita y favorece dicha gestión, además de la comunicación entre equipos de desarrollos, ya que al utilizar formularios y estructuras comunes a todos ellos, se minimizan los malentendidos y ambigüedades.

Por tanto, este objetivo queda cubierto con los requisitos funcionales que componen el grupo funcional **F2** (ver Tabla 5.5 en capítulo 5).

6.3 O3: Favorecer la representación y visualización de la información almacenada

La aplicación obtenida como resultado del desarrollo del PFC permite la representación de las decisiones tomadas en proyectos software, además de otra información asociada, de una manera gráfica, visual e intuitiva, lo que facilita también la comunicación, ya que dicha información puede ser entendida de una manera rápida, visual y sin ambigüedades.

Por tanto, este objetivo queda cubierto con los requisitos funcionales que componen el grupo funcional **F3** (ver Tabla 5.5 en capítulo 5).

6.4 O4: Facilitar la comunicación

Para facilitar la comunicación entre los usuarios de los equipos de desarrollo distribuidos, el sistema implementa mecanismos de comunicación síncrona y asíncrona.

En lo referente a la comunicación asíncrona, la aplicación genera automáticamente alertas cuando se ha producido cualquier cambio sobre las decisiones de un proyecto. Dichas alertas están disponibles para su lectura, en cualquier momento, para todos aquellos empleados que en ese proyecto trabajan. De este modo, este objetivo queda cubierto con los requisitos funcionales que componen el grupo funcional **F4** (ver Tabla 5.5 en capítulo 5).

En cuanto a la comunicación síncrona, gracias a la arquitectura cliente-servidor y a que los clientes deben acceder al sistema, registrándose en el servidor, cuando el cliente realiza cualquier petición relacionada con la gestión de decisiones, el servidor se encarga de notificar al resto de clientes conectados este cambio, para que puedan reflejarlo visualmente en ese mismo instante, en tiempo real. Por tanto, este objetivo también queda cubierto.

6.5 O5: Adaptación al idioma local

Gracias a la creación de ficheros de recursos, escritos en varios idiomas, la aplicación soporta la internacionalización y puede adaptarse al idioma elegido por el usuario. Ésto corresponde al requisito funcional **F8** (ver Tabla 5.5 en capítulo 5), cubriendo este objetivo.

6.6 O6: Facilitar la gestión de proyectos software

Para facilitar la gestión de proyectos software, se han diseñado e implementado funcionalidades para la creación y modificación de dichos proyectos, proveyendo formularios para realizar estas tareas, de modo que se siga una estructura común y se eviten errores. Además, la aplicación también permite asignar o modificar los usuarios que en dichos proyectos trabajan, favoreciendo el control de éstos.

Por tanto, este objetivo queda cubierto con los requisitos funcionales que componen el grupo **F5** (ver Tabla 5.5 en capítulo 5).

6.7 O7: Favorecer aspectos de control de proyectos

En lo que respecta al control de proyectos, la aplicación desarrollada implementa funcionalidades que favorecen dicho control, como es la exportación a archivos XML de la información de los proyectos y sus decisiones; la generación de informes en formato PDF, y la generación de gráficos estadísticos. De este modo, los jefes de proyectos pueden llevar un control acerca de los proyectos y toda su información asociada, generando diferentes informes y gráficos de una manera sencilla y visual.

Por tanto, este objetivo queda cubierto con los requisitos funcionales que componen los grupos funcionales **F6**, **F7** y **F9** (ver Tabla 5.5 en capítulo 5).

6.8 O8: Facilitar la reutilización de información

Para satisfacer este objetivo, propuesto por la necesidad de recuperar y reutilizar decisiones (y toda su información relacionada) de proyectos finalizados, en nuevos proyectos semejantes, se ha diseñado e implementado en la aplicación un mecanismo basado en técnicas de inteligencia artificial (CBR, en este caso) para poder comparar proyectos, recuperar y reutilizar decisiones de dichos proyectos similares.

Por tanto, este objetivo queda cubierto con el requisito funcional **F5.5** (ver Tabla 5.5 en capítulo 5).

Capítulo 7

Conclusiones y Propuestas

En las secciones posteriores de este capítulo se desarrollan las conclusiones obtenidas tras el desarrollo del PFC, así como algunas propuestas para realizar en un futuro.

7.1 Conclusiones

Tras haber realizado y finalizado el desarrollo del PFC, se ha podido comprobar como gracias a utilizar una metodología de desarrollo iterativa e incremental, como es el PUD, el sistema ha ido aumentando en funcionalidad de manera progresiva, facilitando las tareas de análisis, diseño, implementación y pruebas de cada iteración, que daban como resultado un nuevo incremento en la funcionalidad de la aplicación.

Además, gracias a este carácter iterativo del PUD y a su división en fases, se ha podido resolver de una manera sencilla y rápida la incorporación de los nuevos requisitos que fueron detectados al comienzo de la fase de Elaboración, puesto que fueron identificados en iteraciones muy tempranas del ciclo de vida y, por tanto, no causaron un retraso ni impacto importante en el desarrollo del sistema.

En cuanto a las tecnologías y *frameworks* utilizados, Hibernate ha permitido gestionar las operaciones relacionadas con las bases de datos de una manera transparente, permitiendo cambiar el SGBD utilizado sin afectar de ningún modo al sistema desarrollado.

Por otra parte, la elección de RMI ha supuesto una serie de ventajas a la hora de realizar el sistema distribuido:

- Los objetos remotos pueden ser manejados como si fueran locales.

- Gracias a la utilización de interfaces para comunicar los subsistemas, éstos son totalmente independientes de su implementación, y se pueden extender con nuevo métodos de una manera sencilla.

- El servicio de registro de RMI, *rmiregistry*, permite fácilmente localizar e invocar los objetos remotos por su nombre.

- Al estar basado e integrado en Java, resulta sencillo y transparente al programador implementar el modelo de objetos distribuidos.

Como principal inconveniente de utilizar RMI, se puede destacar el problema de utilizarlo junto a Hibernate, ya que al serializar y transferir los objetos remotos, estas referencias no son las mismas que las referencias que mantiene Hibernate para gestionar su modelo de objetos, por lo que se encontraron problemas a la hora de, por ejemplo, modificar objetos en la base de datos cuando el cliente había realizado cambios en ellos. Sin embargo, este problema quedó solucionado clonando los objetos que fueron serializados por RMI y que deben modificarse en la base de datos, usando Hibernate.

Por otro lado, gracias a la arquitectura definida e implementada, y al uso de RMI, se ha conseguido independencia y extensibilidad en el sistema cliente-servidor. De este modo, la implementación del cliente podría cambiar sin verse afectado el servidor, y se podrían añadir nuevas funcionalidades al servidor sin verse afectado el funcionamiento actual de los clientes. Además, se podrían implementar nuevos subsistemas clientes y comunicarse con el servidor de manera sencilla, gracias a las interfaces y RMI, por lo que el sistema puede extenderse sin dificultades.

Para concluir, con el desarrollo de este PFC se han adquirido conocimientos acerca de los temas que en él se abordan, gracias a la revisión de la literatura existente, especialmente en la gestión de decisiones según *Rationale*, y a la comprensión de un método de Inteligencia Artificial, como es el CBR, pudiendo diseñar e implementar un algoritmo de CBR en el sistema. Además, también se ha adquirido o se ha profundizado en el conocimiento y manejo de las herramientas y tecnologías utilizadas para la construcción de la aplicación, destacando el uso de RMI, Hibernate, iText, JUNG, servicios Web de geoposicionamient y cómo diseñar y generar interfaces gráficas de usuario extensibles, flexibles y adaptables, utilizando para ello, además de en otros casos, el API de reflexión de Java.

7.2 Trabajo Futuro

Como trabajo futuro, se propone crear una aplicación Web aprovechando las funcionalidades del servidor actual, y sus interfaces. De este modo, el sistema contaría con una aplicación de escritorio, compuesta por los clientes actualmente implementados, y por una aplicación Web, todo ello gestionado por el servidor. Así, se puede extender su utilización por parte de los usuarios, ya que podría utilizarse desde la aplicación de escritorio y desde la Web.

Por otra parte, se propone extender la funcionalidad del sistema, incorporando un control de los proyectos software y sus decisiones más amplio, generando diferentes informes, tablas, gráficos, etc. Es por ello que se podría incorporar **BIRT**, que es un sistema de informes basado en Java (*Reporting System*, en inglés) con capacidad de generación de tablas dinámicas a partir de fuentes de datos, así como de gráficos e informes avanzados.

Otra consideración a tener en cuenta como trabajo futuro sería incorporar a la aplicación un soporte colaborativo, especialmente a la gestión de decisiones, para que los usuarios puedan interactuar en tiempo real, tengan conciencia de lo que otros usuarios están realizando en ese momento y puedan comunicarse en ese instante, añadiendo, por ejemplo, un chat.

Para terminar, se propone incorporar también una funcionalidad que permita exportar los datos de los proyectos y los empleados que en ellos trabajan desde definiciones de Microsoft Project, pudiendo importar esos datos en la aplicación, creando y almacenando nuevos proyectos con dichos datos.

Bibliografía

[1]

[2] Ågerfalk, P. *et al.*: *Benefits of global software development: the known and unknown*. Making Globally Distributed Software Development a Success Story, pages 1–9, 2008.

[3] Aha, D.W., C. Marling, and I. Watson: *Case-based reasoning commentaries: introduction*. The Knowledge Engineering Review, 20(03):201–202, 2005, ISSN 0269-8889.

[4] Alavi, M. and D.E. Leidner: *Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues*. MIS quarterly, pages 107–136, 2001.

[5] Aspray, W. *et al.*: *Globalization and offshoring of software*. Report of the ACM Job Migration Task Force, Association for Computing Machinery, 2006.

[6] Bornemann, M. *et al.*: *An Illustrated Guide to Knowledge Management*, 2003.

[7] 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.

[8] Chen, A. *et al.*: *Design history knowledge representation and its basic computer implementation*. In *Proceedings of the Design Theory and Methodology Conference, ASME*, pages 175–184, 1990.

[9] Cho, J.: *Globalization and global software development*. Issues in information systems, 8(2):287–290, 2007.

[10] Conchúir, E.Ó. *et al.*: *Global software development: where are the benefits?* Communications of the ACM, 52(8):127–131, 2009.

[11] Damian, D. and D. Moitra: *Guest Editors' Introduction: Global Software Development: How Far Have We Come?* Software, IEEE, 23(5):17–19, 2006.

[12] Dutoit, A.H. *et al.*: *Rationale management in software engineering*. Springer, 2006.

[13] Evaristo, J.R. and R. Scudder: *Geographically distributed project teams: a dimensional analysis*. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*, pages 11–pp. IEEE, 2000.

[14] Evaristo, R. and P.C. van Fenema: *A typology of project management: emergence and evolution of new forms*. International Journal of Project Management, 17(5):275–281, 1999.

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

[16] Freeman, R.E.: *Strategic Management: A Stakeholder Approach*. Cambridge University Press, 2010, ISBN 9780521151740.

[17] Heijstek, Werner: *Global software development*. Leiden University, 2011.

[18] Herbsleb, J.D. and D. Moitra: *Global Software Development*. Software, IEEE, 18(2):16–20, 2002.

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

[20] Hovland, I.: *Knowledge management and organisational learning: An international development perspective*. Overseas Development Institute Working Paper No. 224, Overseas Development Institute, London, UK.

[21] 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.

[22] Jacobson, I., G. Booch y J. Rumbaugh: *El proceso unificado de desarrollo de software*. Addison Wesley, 2000.

[23] Kanagasabapathy, K.A., R. Radhakrishnan, and S. Balasubramanian: *Empirical investigation of critical success factor and knowledge management structure for successful implementation of knowledge management system: A case study in process industry*. 2006.

[24] Kolodner, J.L.: *An introduction to case-based reasoning*. Artificial Intelligence Review, 6(1):3–34, 1992, ISSN 0269-2821.

[25] Kruchten, P.: *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[26] Lee, J.: *Design rationale systems: understanding the issues*. IEEE Expert, 12(3):78–85, 1997.

[27] Lopez De Mantaras, R. *et al.*: *Retrieval, reuse, revision and retention in case-based reasoning*. The Knowledge Engineering Review, 20(03):215–240, 2005, ISSN 0269-8889.

[28] 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.

[29] Sahay, Sundeep, Brian Nicholson, and Srinivas Krishna: *Global IT Outsourcing: Software Development across Borders*. Cambridge University Press, November 2003.

[30] Sangwan, R. *et al.*: *Global software development handbook (auerbach series on applied software engineering series)*. 2006.

[31] Torossi, G.: *El proceso Unificado de Desarrollo de Software*.

[32] Watson, I.: *CBR is a methodology not a technology*. Research & Development in Expert Systems XV, pages 213–223, 1998.

[33] Wenger, E.: *Communities of practice: Learning, meaning, and identity*. Cambridge Univ Pr, 1998.