UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA
EN INFORMÁTICA

PROYECTO FIN DE CARRERA

QuQuSI: Plataforma para la Gestión Integral de Concursos Televisivos

Eduardo Monroy Martínez

Septiembre, 2013

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

PROYECTO FIN DE CARRERA

QuQuSI: Plataforma para la Gestión Integral de Concursos Televisivos

Autor: Eduardo Monroy Martínez
Director: Carlos González Morcillo

Septiembre, 2013

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

RESUMEN

Los concursos son un fenómeno televisivo y social en alza que despierta un gran interés por parte de los espectadores. La variedad de concursos emitidos, sus elevadas audiencias y la alta rentabilidad que logran gracias a la publicidad son prueba de ello. La producción de este tipo de programas se realiza en la mayoría de las ocasiones por grandes productoras de forma artesanal y de forma específica para cada concurso concreto. En la actualidad no existen plataformas genéricas para la realización de concursos televisivos que puedan ser utilizadas por pequeños estudios de televisión y productoras locales o regionales.

El presente Proyecto Fin de Carrera surge como solución a la problemática anterior y plantea la construcción de una plataforma libre para la gestión integral de concursos televisivos. *QuQuSI* es un sistema distribuido con arquitectura cliente servidor que da soporte completo al ciclo de vida de un concurso televisivo, desde la fase de preproducción, en la que se generan las pruebas, pasando por la fase de producción, en la que se realiza y graba el concurso, hasta la fase de postproducción, donde se edita el vídeo final.

QuQuSI proporciona un conjunto de aplicaciones que cumplen con una serie de funcionalidades obtenidas a partir del análisis de requisitos de los concursos televisivos actuales; habilita el *workflow* para la generación de preguntas multimedia, permite la integración con los equipos de producción, realización y grabación genéricos habitualmente disponibles, es capaz de desplegar contenidos audiovisuales 2D y 3D, proporciona una interfaz de control distribuida y da soporte a la participación del público mediante dispositivos móviles, todo ello basándose en estándares y tecnologías libres.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE LISTADOS

AGRADECIMIENTOS

Son muchas las personas a las que quiero agradecer su apoyo y ayuda.

En primer lugar a mi familia, especialmente a mi madre, sin quien no habría podido llegar hasta aquí.

A todos mis amigos, en especial a Juan Andrada y Rafael Muela, compañeros durante la carrera.

A Raquel, por su infinita paciencia, su ayuda y su compañía.

Por último, pero no menos importante, a todos los profesores de la Escuela Superior de Informática, en especial a Carlos González Morcillo, buen profesor, director y mejor persona.

CAPÍTULO 1. INTRODUCCIÓN

En los últimos años, los concursos han proliferado en la parrilla televisiva, ocupando las franjas de emisión con mayor audiencia y rentabilidad.

La realización de un concurso televisivo requiere, además del equipamiento necesario, que se lleven a cabo un conjunto de tareas que van desde la gestión de preguntas y contenidos empleados en el concurso hasta su grabación y posterior edición. La producción de este tipo de espacios requiere, además de la preparación del escenario y "atrezzo", multitud de herramientas software y hardware para dar soporte a las tareas específicas de cada concurso. En la actualidad no existen plataformas genéricas abiertas que den soporte a la producción de este tipo de programas.

Con la finalidad de ocupar este vacío se plantea el presente Proyecto Fin de Carrera, *QuQuSI*, una plataforma para la gestión integral de concursos televisivos con soporte para la participación del público.

1.1. CONCURSOS TELEVISIVOS

Los concursos son un género televisivo en el que uno o varios participantes se enfrentan a diferentes pruebas para conseguir un premio.

Las mecánicas y temáticas de los concursos televisivos son muy variadas. Existen concursos de preguntas sobre cultura general, como *«Saber y ganar»*. En otros, entra en juego la capacidad de cálculo y la variedad de vocabulario, como *«Cifras y Letras»*. Están aquellos en los que prima la velocidad de respuesta, como *«¡Ahora caigo!»* o *«Atrapa un millón»*. También se encuentran a la orden del día los concursos a pie de calle, como *«Lo sabe, no lo sabe»* o *«Negocia como puedas»*.

Los concursos no solo están presentes en la televisión. Gracias a videojuegos como *«Buzz!»*, los usuarios de videoconsolas pueden reproducir la experiencia de un concurso televisivo en sus casas. Otros videojuegos que cuentan con una mecánica similar a la de los concursos televisivos, como *«Atriviate»*, permiten a los usuarios de teléfonos móviles enfrentarse con sus amigos en cualquier momento.

Los concursos, al igual que los juegos educativos, favorecen el aprendizaje. Juegos educativos como *«Brain Training»* permiten entrenar las habilidades mentales llevando un registro del progreso conseguido. Otros juegos, como *«Aprende con Pipo»*, son una valiosa herramienta pedagógica orientada al público infantil.

La realización de un concurso televisivo requiere que se lleven a cabo un gran número de tareas; concebir la mecánica, preparar las pruebas, decorar el escenario, seleccionar a los participantes, preparar el equipo de grabación, realizar el concurso y un largo etcétera.

Existen algunas plataformas para la gestión de concursos televisivos, como AdiutorQuizShow[1]. Sin embargo estas alternativas son privativas y cerradas, limitando su capacidad de personalización por parte del usuario final, y tienen un elevado coste, por lo que quedan fuera del alcance de multitud de instituciones y empresas que podrían sacarles partido.

La creación de una plataforma libre para la gestión integral de concursos televisivos permitirá a instituciones y pequeñas productoras la realización de este tipo de programas.

Un ejemplo de la utilidad de *QuQuSI* se dio en la fase final de las VII Olimpiadas Informáticas que se realizó en la Escuela Superior de Informática el día 24 de mayo de 2013. En este evento, los tres grupos de cada categoría con mejor calificación en las pruebas presenciales accedieron a la prueba final, la cual consistió en un concurso televisivo en el que se decieron los premios. El concurso, grabado y realizado en directo, contó con la asistencia de más de cien personas que contestaron a las mismas preguntas que los concursantes a través de sus dispositivos móviles. Posteriormente, se editó el vídeo del concurso y se publicó en el canal de YouTube de la escuela[2]. Trás la realización del concurso, varias instituciones y cadenas de televisión (Universidad Politécnica de San Luis Potosí, Televisión Ciudad Real e Imás TV) se pusieron en contacto para interesarse por la plataforma y utilizarla en un entorno de producción profesional (Véase Anexo F).

Queda así demostrada la utilidad de una plataforma libre para la gestión integral de un concurso con formato televisivo que permita la personalización de la mecánica de juego y la temática de las preguntas.

1.2. IMPACTO SOCIO-ECONÓMICO

El impacto social de los concursos televisivos queda reflejado en el gran número de diferentes concursos que se emiten actualmente y en las elevadas audiencias que logran [1].

Los concursos se rentabilizan generalmente gracias a la publicidad. En las franjas horarias en las que se emiten el precio por spot de 20 segundos alcanza los 19.000 €. Ejemplos

[1] http://www.aranova.es/
[2] http://www.youtube.com/watch?v=eGgvzBzjDQk

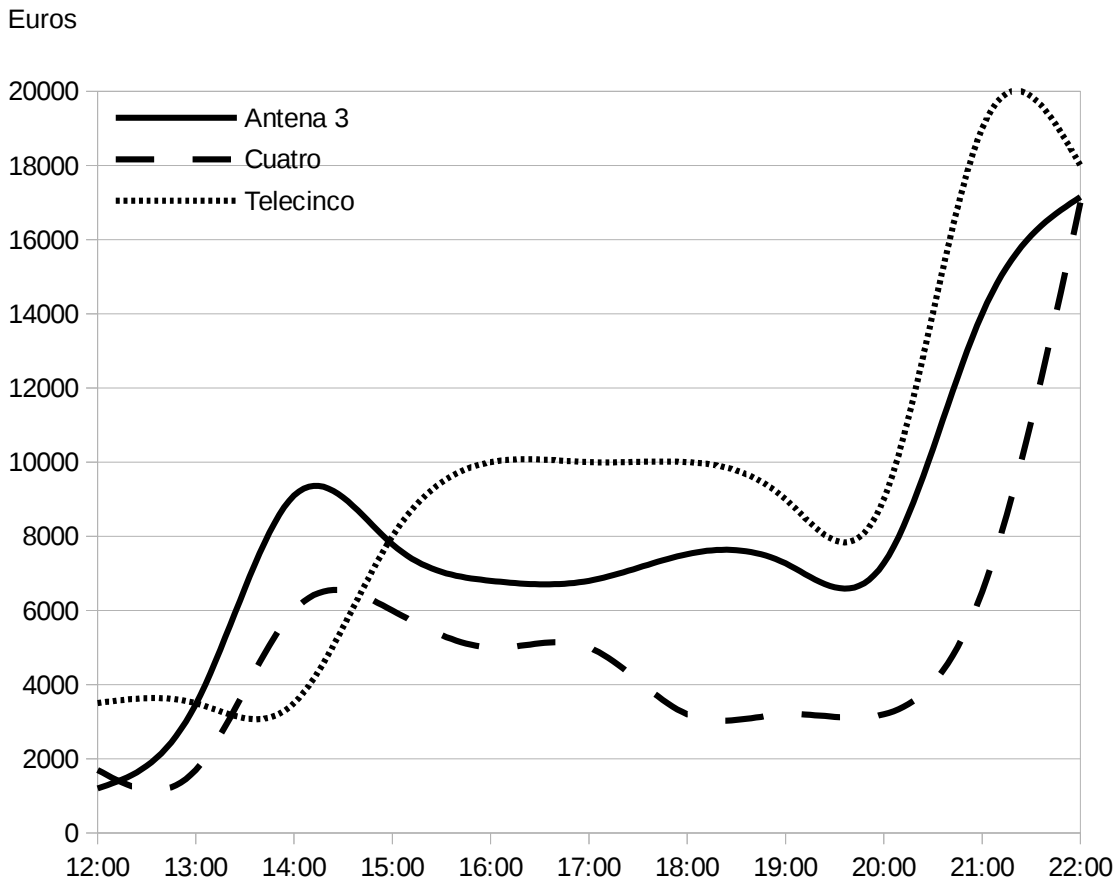

Figura 1.1: Precio por spot de 20 segundos en función de la franja horaria [2].

son *«Atrapa un millón»*, que se emite en Antena 3 de 20:00 a 21:00 con un precio por spot que va desde los 7.270 € hasta los 13.000 €, *«Pasapalabra»*, que se emite en Telecinco de 20:15 a 21:05 con un precio por spot que llega a los 19.000 €, o *«¡Ahora caigo!»*, el cual se emite de 18:45 a 20:00 con un precio por spot de 7.270 €.

1.3. PROBLEMÁTICA

La gestión integral de un concurso televisivo plantea una serie de problemas de diferente naturaleza. Debe dar soporte a todas las fase del ciclo de vida del concurso, desde la fase de preproducción, donde se generen y validen las preguntas, pasando por la fase de producción, cuando se realice y grabe el programa, hasta la fase de postproducción, donde se edite y componga el vídeo final.

En la fase de preproducción se requiere que se lleve a cabo un *workflow* o flujo de trabajo cuya salida sea un conjunto de preguntas validadas. Este conjunto deberá ser suficientemente grande como para que el concurso tenga la duración deseada. En el flujo de trabajo intervendrán personas con diferentes roles y responsabilidades, encargándose unos de la creación de las preguntas, otros de su validación y otros de la gestión de los usuarios y sus roles.

Durante la fase de producción tiene lugar el grueso del trabajo ya que se debe dar soporte a una gran variedad de funcionalidades. En primer lugar es necesario el despliegue de las preguntas generadas durante la fase de preproducción. Estas se deben presentar de un modo adecuado que permita tanto su proyección como su combinación con la imagen real. Además se han de reproducir los contenidos multimedia (imagen, vídeo y sonido) asociados a las preguntas. En segundo lugar se requiere un mecanismo de control distribuido que permita la realización del concurso, permitiendo enviar las señales de control y eventos que dirijan su desarrollo. En tercer lugar la plataforma ha de dar soporte a la participación del público a través de dispositivos móviles como teléfonos y *tablets* en tiempo real. Por último, se debe permitir el despliegue físico y la integración con el hardware disponible.

En la fase de postproducción es necesario que se pueda reproducir de nuevo la salida gráfica y sonora del concurso, repitiendo las señales de control y eventos acontecidos, a fin de facilitar la edición y composición del vídeo final y a la posible participación del público en diferido.

A partir de todos estos problemas se extraen las siguientes características y funcionalidades que debe satisfacer una plataforma para la gestión integral de concursos televisivos:

- Habilitar un flujo de trabajo en el que intervengan diferentes roles y cuyo resultado sea un conjunto de preguntas validadas.

- Permitir el despliegue de elementos gráficos 2D y 3D, soportando la reproducción de imagen y vídeo.
- Soportar diferentes modos de visualización que permitan la proyección o la combinación con la imagen real.
- Reproducir efectos sonoros y música ambiental.
- Integración con los dispositivos disponibles, soportando un conjunto heterogéneo de hardware.
- Ofrecer un mecanismo de control distribuido para facilitar la realización del concurso en directo.
- Permitir la participación del público a través de dispositivos móviles de diferente naturaleza.
- Guardar un *log* de señales de control y eventos que pueda reproducirse posteriormente.

En el capítulo 2 se describen con mayor detalle estas características y funcionalidades. *QuQuSI* se plantea como una plataforma para la gestión integral de concursos televisivos que satisface las anteriores características. Los conceptos y las alternativas tecnológicas analizadas para su desarrollo se describen en el capítulo 3.

1.4. ESTRUCTURA DEL DOCUMENTO

El presente documento se ha redactado conforme a la normativa del Proyecto Fin de Carrera de la Escuela Superior de Informática de la Universidad de Castilla-La Mancha, incluyendo los siguientes capítulos:

- **Capítulo 2. Objetivos:** En este capítulo se enumeran y describen los objetivos y subobjetivos planteados para el presente Proyecto Fin de Carrera.
- **Capítulo 3. Antecedentes:** En este capítulo se resumen los principales conceptos estudiados y se analizan algunas de las alternativas tecnológicas disponibles para la realización de *QuQuSI*.
- **Capítulo 4. Método de trabajo:** En este capítulo se explica y justifica la metodología elegida para llevar a cabo el presente Proyecto Fin de Carrera. Además se listan las herramientas, tanto software como hardware, empleadas en el desarrollo y despliegue de *QuQuSI*.

- **Capítulo 5. Arquitectura:** En este capítulo se describen las aplicaciones que forman la plataforma y los módulos que las componen siguiendo un enfoque *top-down*, comenzando con una descripción funcional, continuando con las decisiones de diseño y terminando con los detalles de implementación.
- **Capítulo 6. Evolución y costes:** En este capítulo se describe la evolución del sistema durante su desarrollo y se desglosan los costes asociados.
- **Capítulo 7. Conclusiones y propuestas:** En este capítulo se analizan el trabajo realizado y los objetivos alcanzados. También se plantean líneas de trabajo futuro para mejorar y ampliar la plataforma.

CAPÍTULO 2. OBJETIVOS

En este capítulo se enumeran y describen los objetivos y subobjetivos planteados para el presente Proyecto Fin de Carrera.

2.1. OBJETIVO GENERAL

El objetivo general del proyecto puede definirse como el **desarrollo de una plataforma para la gestión integral de un concurso con formato televisivo que permita la participación del público**.

La plataforma deberá abarcar el ciclo de vida de un concurso televisivo genérico al completo, partiendo desde la recogida de las preguntas, pasando por la grabación y emisión del programa y terminando con la postproducción del mismo.

2.2. OBJETIVOS ESPECÍFICOS

A partir del objetivo principal descrito en la sección anterior se determinan los siguientes subobjetivos definidos a través del análisis de requisitos funcionales descritos en la sección 3.1:

2.2.1. Despliegue gráfico multimedia 2D y 3D

Se desarrollarán diferentes *widgets* que den soporte al despliegue de gráficos 2D y 3D tales como vídeos, imágenes y textos. También se contará con soporte para la reproducción de sonido.

La plataforma deberá ser capaz de:

- Utilizar fuentes *TrueType* y ajustar el tamaño del texto en función del espacio disponible para mejorar su legibilidad.
- Representar cada tipo de pregunta del concurso de una forma comprensible y adecuada a la información.

- Reproducción de música y efectos de sonido con posibilidad de controlar el volumen de forma gradual en tiempo de ejecución.
- Ofrecer una interfaz que permita la combinación de la salida gráfica con la imagen real obtenida por las cámaras mediante la técnica del *chroma key*.
- Reproducción de archivos multimedia de cualquier tipo (imágenes, vídeos y sonido) de forma eficiente en tiempo real.

2.2.2. Integración con los sistemas existentes

Se analizarán las características genéricas de los sistemas de grabación y proyección existentes en el mercado para su integración en la plataforma. Se tendrán varias aplicaciones que permitan aprovechar los elementos disponibles, ya sea para el control de la plataforma o para la visualización del concurso. Se realizará un estudio exhaustivo específico con los medios disponibles en la Escuela Superior de Informática de la UCLM.

El sistema habrá de permitir:

- Contar con diferentes modos de visualización que permitan la proyección de la salida gráfica o su combinación con la imagen real.
- Posibilidad de configurar tanto el aspecto gráfico como el sonoro de la plataforma, permitiendo su conexión con los diferentes dispositivos de visualización y mesas de mezcla de vídeo y audio disponibles.

2.2.3. Sistema para la gestión de preguntas multimedia

Se creará un sistema que permita la gestión de diferentes tipos de preguntas multimedia. El sistema dará soporte a un flujo de trabajo en el que se definirán roles con diversas responsabilidades, desde profesores que crearán las preguntas a validadores que revisarán y verificarán los contenidos.

Este sistema deberá permitir:

- Autenticación de los usuarios en el sistema, garantizando la confidencialidad de las preguntas y previniendo filtraciones de las preguntas a los concursantes.
- La asignación de roles a los usuarios, de modo que cada uno cuente con permisos, accesos y responsabilidades diferentes. Así, el sistema deberá contar con al menos tres roles; administrador, profesor y validador. Así se definirá un *workflow* en los que cada rol cumplirá una función especifica.

- Un mecanismo que permita que las preguntas sean validadas antes de su explotación.
- Un diseño basado en el patrón MVC (Modelo Vista Controlador) con soporte para las operaciones CRUD (*Create, Read, Update and Delete*) que facilite la posterior adaptación de la plataforma a diferentes entornos de explotación.
- El uso de una interfaz Web que facilite la gestión de las preguntas y que sea accesible desde cualquier dispositivo.

2.2.4. Plataforma para la intervención del público

El sistema permitirá la participación del público a través de sus móviles y *tablets*, soportando los sistemas operativos más comunes en estos dispositivos (*Android*, *iOS* y *Windows Phone*).

En este grupo de requisitos funcionales, la plataforma deberá permitir:

- Sincronización con el estado de concurso en tiempo real, de modo que el público responda a las mismas preguntas que los concursantes en tiempo real.
- Mecanismo de autenticación que permita identificar de forma unívoca a cada participante.
- Retroalimentación del usuario de modo que le sea fácil seguir el desarrollo del concurso.
- Clasificación de los participantes por aciertos y tiempo de respuesta, evitando empates.

2.2.5. Múltiples interfaces de control

Se desarrollarán varias aplicaciones que permitan un control distribuido del sistema. Cada controlador enviará la señal a un servidor central que mantendrá el estado del concurso y actualizará los elementos del sistema, garantizando la coherencia.

La plataforma permitirá:

- Enviar señales de control que dirijan el desarrollo del concurso en función de los sucesos acontecidos.
- Ejecutar múltiples instancias de forma que se comparta la responsabilidad de control del concurso en diferentes localizaciones (presentador, mesa de control, etc.).

- Mostrar información suficiente para entender el estado del concurso y poder presentarlo de forma sencilla.
- Desplegar toda la información relacionada con la pregunta actual, de modo que el presentador pueda aclarar confusiones y dar explicaciones.

2.2.6. Robustez del sistema y recuperación ante errores

La plataforma debe dar soporte para la producción de concursos en directo. Así, es importante que el sistema no falle de forma crítica y pueda recuperarse de cualquier fallo puntual sin afectar al desarrollo del concurso. Por lo tanto deberá ser tolerante a errores utilizando técnicas como la replicación y la recuperación automática.

Así, el sistema debe permitir:

- Recuperarse de forma automática ante un fallo en la red de comunicaciones, tratando de reconectarse periódicamente.
- Notificar cualquier suceso que pudiera alterar el correcto desarrollo del concurso mediante mensajes de advertencia y error.

2.2.7. Plataforma distribuida y heterogénea

La plataforma estará formada por varias aplicaciones software que se ejecutarán en diferentes máquinas (ordenadores, móviles, *tablets* y otros controladores hardware).

Por ello debe permitir:

- Utilizar una arquitectura cliente servidor que permita la comunicación entre las diferentes aplicaciones del sistema distribuido.
- El uso de interfaces Web que puedan usarse a través de un navegador, siendo independientes de la plataforma.
- Permitir la comunicación entre aplicaciones software escritas en diferentes lenguajes.

2.2.8. Arquitectura flexible y extensible

Para favorecer la reutilización de la plataforma su arquitectura será diseñada con vistas a la flexibilidad de despliegue y una implementación extensible, de modo que pueda utilizarse para la gestión integral de concursos con diferentes temáticas y mecánicas.

Por esto el sistema debe:

- Utilizar patrones de diseño que mejoren la calidad del código y solucionen problemas recurrentes.
- Favorecer la reutilización y mantenibilidad del código, disminuyendo el acoplamiento y aumentando la cohesión del diseño.
- Implementarse mediante buenas prácticas de programación, incluyendo los comentarios suficientes para su comprensión.
- Capacidad de configuración de todas las partes de la plataforma, permitiendo reutilizar la plataforma para concursos de diferente temática.

2.2.9. Basado en estándares y tecnologías libres

El proyecto será llevado a cabo utilizando software y hardware libre. Las herramientas y bibliotecas usadas tendrán licencias libres para facilitar su posterior comercialización y puesta en explotación en entornos profesionales.

Así, la plataforma debe:

- Construirse utilizando herramientas con licencias libres sin coste económico e independiente de los intereses económicos de terceros.
- Utilizar bibliotecas de código abierto y con licencias libres para la implementación.
- Desarrollarse empleando bibliotecas multiplataforma (al menos disponibles en *GNU/-Linux* y *Windows*).
- Emplear hardware libre para la construcción de los prototipos electrónicos.

CAPÍTULO 3. ANTECEDENTES

Alcanzar los objetivos propuestos en *QuQuSI* requiere un amplio abanico de conocimientos en numerosos campos de la informática, tales como son las redes de computadores, la representación de gráficos 3D por computador, la reproducción de contenidos multimedia, la programación de microcontroladores, la ingeniería del software o las bases de datos.

En este capítulo se resumen los principales conceptos estudiados para llevar a cabo el presente Proyecto Fin de Carrera (Fig. 3.1). Puesto que no existe ninguna plataforma libre para la gestión integral de concursos televisivos, el estudio del arte se ha centrado en las diferentes áreas de la informática y en las soluciones tecnológicas susceptibles de resultar de utilidad para la creación de una plataforma con las características descritas en el capítulo 2. En primer lugar se estudia el concepto de jugabilidad, haciendo una revisión de los concursos televisivos más populares en la actualidad, analizando algunas de las posibles causas de su éxito. A continuación se describen una serie de fundamentos de la ingeniería del software que resultan útiles para el desarrollo del proyecto. Después se estudian los servidores Web, necesarios para el despliegue de aplicaciones Web, y los *sockets*, los cuales permiten la implementación de protocolos de comunicación y sincronización. En siguiente lugar se examinan medios para el almacenamiento de la información, como son las bases de datos y los sistemas de archivos. Dado el despliegue gráfico requerido, se estudiarán los gráficos 3D por computador, el proceso de *rendering* y la técnica audiovisual de *chroma key*. Por último, se analizan algunos de los *frameworks* con licencias libres existentes que permitan implementar y dar soporte a lo anterior.

3.1. *GAMEPLAY*

El concurso al que da soporte *QuQuSI* es un sistema interactivo, y como tal cuenta con la característica de la usabilidad. Este concepto está relacionado con la capacidad de un software de ser comprendido, aprendido, usado y disfrutado por el usuario [22]. Según la ISO 9241-11 [25], la usabilidad puede definirse como *la medida en que un producto puede ser usado por usuarios específicos para lograr objetivos especificados con efectividad, eficiencia*

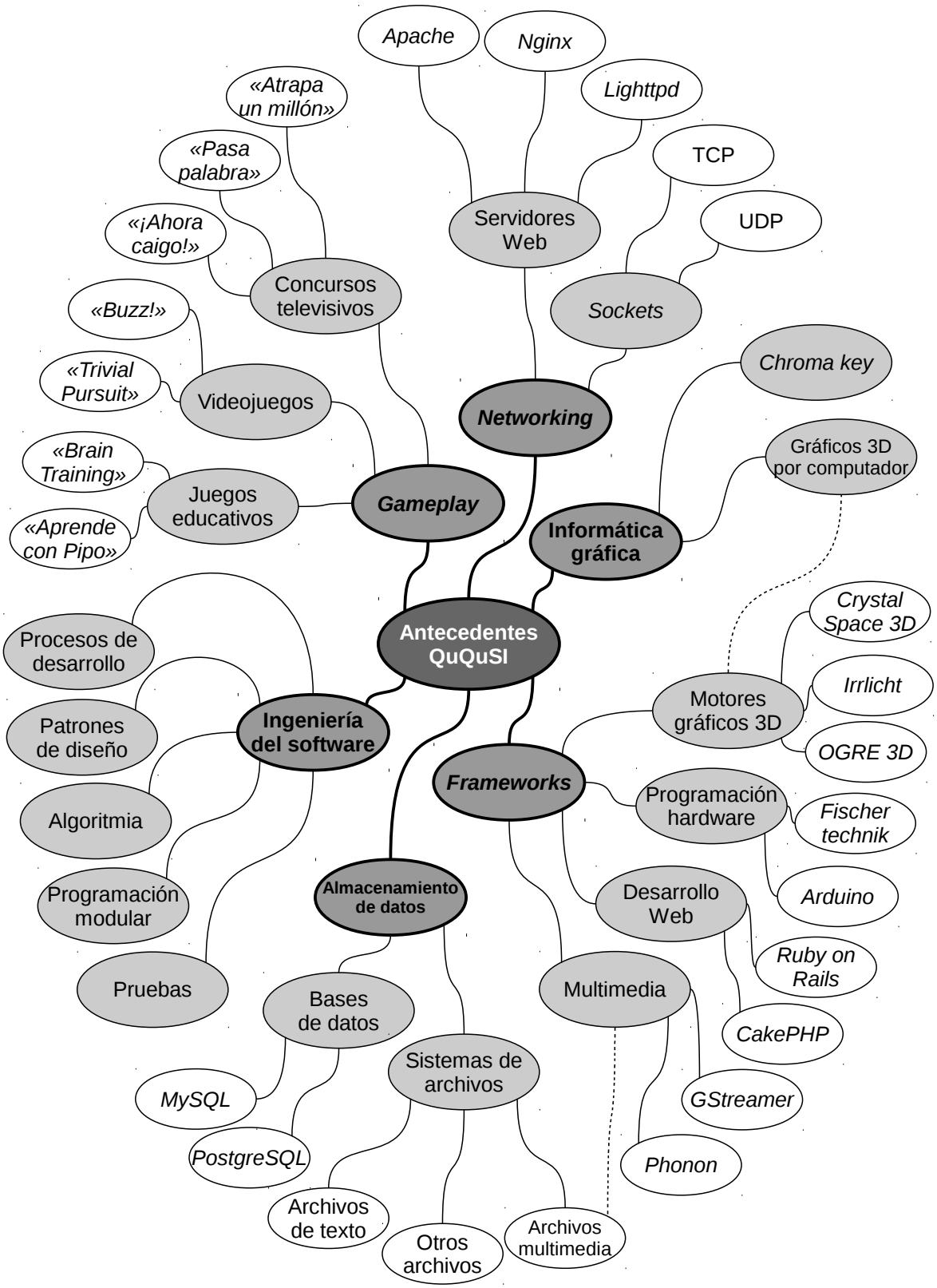

Figura 3.1: Mapa conceptual de antecedentes de *QuQuSI*.

y satisfacción en un contexto de uso. Así, la usabilidad de un sistema se mide en función de sus propiedades de efectividad, eficiencia, satisfacción, aprendizaje y seguridad.

Los concursos televisivos han de resultar entretenidos ya que, como se menciono anteriormente, se rentabilizan mediante los ingresos publicitarios. Una mayor audiencia implica un mayor interés por parte de los anunciantes para aparecer en los espacios publicitarios del programa. Así, el entretenimiento es un punto clave a la hora de concebir un concurso exitoso. Por ello, se ha estudiado el concepto de jugabilidad.

El *gameplay* o jugabilidad es un concepto que extiende al de usabilidad y que puede definirse como *el conjunto de propiedades que describen la experiencia del jugador ante un sistema de juego determinado, cuyo principal objetivo es divertir y entretener "de forma satisfactoria y creíble" ya sea solo o en compañía* [23].

Este término suele relacionarse con el software de entretenimiento (videojuegos). Sin embargo, no siempre ha de ser así, pudiendo utilizarse para medir las propiedades de satisfacción, aprendizaje, efectividad, inmersión, motivación, emoción y socialización de cualquier producto de entretenimiento, sea o no software [23]. Estas propiedades no solo extienden las descritas en la ISO 9241, sino que también suponen un diferente matiz. Por ejemplo, desde el punto de vista de la usabilidad, resulta deseable que un software cuente con una curva de aprendizaje rápida que permita al usuario familiarizarse con la herramienta lo antes posible. Sin embargo, desde el punto de vista de la jugabilidad, una curva de aprendizaje rápida puede suponer que un juego no plantee la dificultad suficiente al jugador como para mantener su interés.

En la jugabilidad de un concurso televisivo, además de la experiencia de los concursantes, se ha de tener en cuenta la experiencia del público, pues es hacia quién realmente se dirige la intención de entretener. Idealmente, se puede tratar de aproximar ambas experiencias permitiendo al público participar de algún modo en el concurso, ya sea prestando ayuda a los concursantes o respondiendo a las mismas preguntas, optando por un premio.

En las siguientes secciones se describen las características y la mecánica de algunos concursos televisivos, videojuegos y juegos educativos que servirán para definir las características generales que debe soportar una plataforma como *QuQuSI*.

3.1.1. Concursos televisivos

Los concursos son un género televisivo en el que uno o varios participantes se enfrentan a diferentes pruebas para conseguir un premio.

Este formato de programa de televisión está en alza gracias a algunos exitosos concursos como son *«Pasapalabra»* (Telecinco), *«La ruleta de la suerte»*, *«¡Ahora caigo!»* o *«Atrapa*

un millón» (Antena 3). La prueba de su éxito son las elevadas audiencias que logran [1].

Aunque los concursos atravesaron una crisis en la que su tiempo de emisión se vio reducido a las horas de menor audiencia, se han vuelto a popularizar hasta el punto de ocupar los huecos de máximo *sharing*.

Algunas de las claves de este resurgimiento son un formato dinámico con pausas breves, una realización y un aspecto visual atractivos, preguntas de una dificultad asequible para el espectador medio y el carisma de presentadores tales como Carlos Sobera o Arturo Valls.

Otro aspecto novedoso e importante de los concursos actuales es la integración del público. La audiencia puede participar en el concurso respondiendo a las mismas preguntas que los concursantes en tiempo real como es el caso de *«Atrapa un Millón»*. También contribuye a este aspecto la grabación del público asistente.

Este tipo de programas se rentabiliza gracias a la publicidad. En las franjas horarias en las que se emiten, el precio por un spot de publicidad de 20 segundos puede alcanzar los 19.000 € [2].

A continuación se describe la mecánica de alguno de estos concursos, a partir de las cuales se definieron las principales características que debía soportar *QuQuSI*.

«Atrapa un millón»

Figura 3.2: Imagen de *«Atrapa un millón»* obtenida de [3].

«Atrapa un millón» es un concurso actualmente emitido en Antena 3 de lunes a viernes de 20:00 a 21:00 y presentado por Carlos Sobera. En la franja horaria en la que se emite, el

spot de publicidad de 20 segundos va desde los 7.270 € hasta los 13.000 € [2]. Su versión original, *«The Money Drop»*, se emite en 13 países de todo el mundo [3].

La mecánica del juego es sencilla. Una pareja de concursantes recibe una suma de dinero al comienzo del concurso (200.000 € en la versión diaria y 1.000.000 € en la versión semanal). Para ganar el dinero, deben enfrentarse a ocho preguntas con un número de opciones que depende del número de pregunta, siendo cuatro en las cuatro primeras, tres en las tres siguientes y dos en la última. En cada pregunta, los concursantes colocan el dinero sobre unas trampillas, repartiéndolo entre las opciones disponibles en un tiempo no superior a un minuto. Al desvelarse la solución, conservan el dinero que colocaron en la respuesta correcta, mientras que pierden el dinero situado sobre las opciones incorrectas al caer este por una trampilla. La temática de las preguntas es variada, pudiendo elegir los concursantes entre dos temas antes de responder a la pregunta.

El formato de pregunta con opciones es uno de los más utilizados por los concursos (*«¿Quién quiere ser millonario?»*, *«El millonario»*) al entrañar un nivel de dificultad asequible para el público medio. Esto aumenta el interés de la audiencia al sentirse capaces de responder a las preguntas.

Otro aspecto interesante de la mecánica de *«Atrapa un millón»* es la dificultad de sus preguntas. Aunque no se indique en ningún lugar, la dificultad de las preguntas en las fases finales aumenta notablemente en función de la cantidad de dinero que conserven los participantes.

Adicionalmente, *«Atrapa un millón»* cuenta con una aplicación Web [5] implementada en Flash que permite al público participar desde sus casas respondiendo a las mismas preguntas que los participantes. Aquellos espectadores que consiguen los mejores resultados son recompensados con facilidades para acceder al *casting* de participación en el concurso, donde optarán por el premio en metálico. La aplicación muestra estadísticas de participación y aciertos en tiempo real en el concurso, aumentando así la inmersión de la audiencia.

«Pasapalabra»

El concurso *«Pasapalabra»*, anteriormente emitido por Antena 3, se emite actualmente en Telecinco de lunes a viernes de 20:15 a 21:05 y es presentado por Christian Gálvez [4]. El precio del spot de 20 segundos durante el espacio publicitario alcanza los 19.000 € [2].

En *«Pasapalabra»* dos concursantes, con la ayuda de una pareja de invitados famosos, acumulan segundos para la prueba final, en la que se enfrentan por la permanencia en el concurso y por un bote que se acumula tras cada programa (el máximo ganado actualmente es de 1.524.000 €) [4]. Tras jugar una serie de pruebas diferentes, los dos concursantes

Figura 3.3: Imagen de *«Pasapalabra»* obtenida de [4].

participan en una prueba final ("El rosco") en la que han de responder con rapidez a las definiciones dadas por el presentador con la única pista de una letra del alfabeto. El concursante con más aciertos en esta prueba final permanece en el concurso. Además, si acierta todas las definiciones puede ganar el bote acumulado.

«Pasapalabra» cuenta con multitud de pruebas diferentes, como reordenar las letras de las palabras para formar otras dada una definición, emparejar definiciones con términos poco conocidos, unir logros con nombres de personajes, etc. Este concurso presenta una dificultad algo superior al resto de programas estudiados. Aún así, gracias al reto que supone logra enganchar al espectador, especialmente en la popular prueba final.

«¡Ahora caigo!»

«¡Ahora caigo!» se emite en Antena 3 de lunes a viernes de 18:45 a 20:00. Su presentador es Arturo Valls [3] y el precio por spot publicitario de 20 segundos en su franja horaria es de 7.270 € [2].

En *«¡Ahora caigo!»* un concursante central se enfrenta uno a uno a otros diez en duelos de preguntas contrarreloj. Se pueden dar tantas respuestas como se quiera, pero si la cuenta atrás termina sin haber dado la solución correcta el concursante queda descalificado, abriéndose una trampilla bajo sus pies por la que se precipita al vacío. El concursante central cuenta con comodines gracias a los que puede pasar la pregunta a su rival. Cada concursante del circulo exterior cuenta con una "moneda" que equivale a una cantidad oculta de dinero (desde 1 € hasta 5000 €) la cual se muestra al terminar el duelo. Si un concursante del circulo derrota

Figura 3.4: Imagen de *«¡Ahora caigo!»* obtenida de [3].

al concursante central, gana la cantidad de dinero oculta en su "moneda", mientras que si el ganador del duelo es el concursante central, se suma la cantidad del rival derrotado al total. El concursante central puede retirarse una vez se han descubierto un número de monedas o optar al premio mayor de 200.000 € que consigue derrotando a todos los rivales.

El formato de las pruebas es el de una pregunta y una respuesta de la que solo se muestra su número de caracteres como espacios en blanco y algunas letras a modo de pista. Este tipo de prueba también favorece a la experiencia del espectador, ya que pueden conocer la respuesta directamente o deducirla a partir de las pistas.

«La ruleta de la suerte»

«La ruleta de la suerte» es emitido en Antena 3 de lunes a viernes al mediodía, de 12:50 a 14:00, y es presentado por Jorge Fernández [3]. El precio del spot de 20 segundos es de 3.465 € durante su franja horaria [2].

En este programa se enfrentan tres concursantes que participan por conseguir dinero, llegando a una prueba final aquel que haya acumulado más. Los concursantes se van turnando para hacer girar una ruleta en la que están marcados premios en metálico, regalos, comodines o efectos negativos como la quiebra o la pérdida de turno. Al caer en una marca de dinero o regalo, los concursantes deben dar una consonante para ir desvelando una frase oculta de la que se da una pista al inicio de la ronda. Para ganar el dinero acumulado en la ronda, el concursante debe dar la solución de la frase oculta durante su turno. Se suceden así rondas de

Figura 3.5: Imagen de *«La ruleta de la suerte»* obtenida de [3].

turnos hasta llegar a la final, en donde el concursante que acumuló más dinero puede ganar un coche o más dinero en metálico.

Las pruebas de *«La ruleta de la suerte»* cuentan con varias modalidades. El caso normal consiste en tratar de desvelar una frase oculta haciendo girar la ruleta, dando consonantes y comprando vocales hasta tener suficientes pistas como para dar la solución. Otro caso consiste en una prueba de velocidad en la que van apareciendo las letras de la frase hasta que un participante activa un pulsador y da la respuesta correcta.

La dificultad de este tipo de pruebas es asumible por el espectador medio, evitando que pueda sentirse frustrado por no saber responder. Además, el público puede participar desde casa llamando por teléfono durante la ronda del espectador.

Características deseables de los concursos televisivos

Del análisis de la mecánica y las características de estos programas se pueden extraer una serie de propiedades que debe soportar una plataforma que de soporte en general a concursos televisivos:

- Carga de preguntas organizadas en categorías y con diferente nivel de dificultad.
- Adaptación dinámica de la dificultad de la pregunta según el estado del juego.
- Facilidad para la composición de la imagen obtenida desde las cámaras por la generada por ordenador.

- Integración de efectos de sonido que se puedan mezclar en tiempo real.
- Soporte de diferentes tipos de pruebas y dinámicas de juego.
- Posibilidad de incluir "cortinillas" para la transición entre diferentes tipos de prueba.
- Soporte para la interacción con el mundo físico mediante pulsadores y actuadores mecánicos.
- Gestión de múltiples pantallas, diferentes resoluciones y modo de despliegue.
- Habilitar mecanismos para facilitar la participación del público en tiempo real.
- Evitar la frustración de los participantes adaptando la dificultad de las preguntas a su nivel.

En esta sección se han analizado los aspectos más destacables de los concursos televisivos. En la siguiente sección se analizan los elementos fundamentales que se han de tener en cuenta en el diseño de videojuegos con una mecánica similar a la de los concursos televisivos.

3.1.2. Videojuegos

El éxito de los concursos no solo se limita a la televisión. Gracias a videojuegos como *«Buzz!»*, con más de ocho millones y medio de unidades vendidas en todo el mundo [9], los usuarios de videoconsolas pueden disfrutar de la experiencia de participar en un programa televisivo en sus propias casas. Mediante periféricos como PlayStation Eye se puede aumentar la inmersión del usuario en el concurso al permitir que se tomen fotografías, se graben vídeos o se personalicen los avatares virtuales con el rostro de los usuarios.

También existen videojuegos que versionan juegos de mesa con una mecánica parecida a la de los concursos televisivos, como el popular *«Trivial Pursuit»*, el cual cuenta con versiones para todas las consolas actuales y PC.

A continuación se describen algunos ejemplos de videojuegos semejantes a concursos televisivos.

«Buzz!»

La saga *«Buzz!»*, desarrollada por Relentless Software y publicada por Sony Computer Entertainment, es una serie de videojuegos que simulan un concurso de tipo televisivo.

«Buzz!» cuenta con 18 entregas para los sistemas PlayStation 2, PlayStation 3 y PlayStation Portable, habiendo vendido en total más de ocho millones y medio de unidades. En

Figura 3.6: Captura de *«Buzz!»* obtenida de [15].

2006, el segundo juego de la serie, *«Buzz!: The Big Quiz»*, recibió el premio BAFTA al "Mejor Juego Casual y Social" [6].

Todos sus juegos tienen una mecánica similar. Cada jugador (hasta ocho) utiliza un controlador especial llamado "Buzzer", el cual cuenta con cuatro botones de colores y un pulsador rojo. Los jugadores utilizan el controlador para responder a preguntas cuyo tipo y temática varían en cada entrega de la saga. Las pruebas se agrupan en rondas, ganando al final el jugador con más puntos, igual que en los concursos televisivos.

Algunos de los tipos de pruebas con los que cuenta *«Buzz!»* son:

- **Pregunta test:** Consiste en una pregunta con cuatro opciones.
- **Prueba de velocidad:** Gana puntos el que responda primero.
- **Patata caliente:** Cada respuesta correcta pasa una bomba temporizada a un rival. Pierde el jugador que tiene la bomba cuando el tiempo se acaba.
- **Pregunta oculta:** La pregunta y la respuesta se van mostrando progresivamente. El jugador que crea conocer la solución activa su "Buzzer" y responde.
- **Ronda final:** Los puntos de los jugadores se convierten en un contador que va decrementándose. Los participantes siguen contestando preguntas hasta que solo queda tiempo en el marcador de un jugador, el cual es declarado como vencedor.

Las diferentes versiones de *«Buzz!»* permiten una buena inmersión de los jugadores al permitir la personalización de los avatares con fotos y vídeos obtenidos a través de PlayS-

tation Eye. Además, durante la partida, la cámara puede tomar fotos de los jugadores, mostrando sus reacciones en pantalla.

«Trivial Pursuit»

Figura 3.7: Captura de *«Trivial Pursuit»* obtenida de [17].

«Trivial Pursuit» es un juego de mesa en el que los jugadores demuestran sus conocimientos contestando a preguntas de diferentes temas.

El objetivo del juego es recorrer el tablero respondiendo preguntas. La temática de la pregunta depende de la casilla en la que se encuentre el jugador. Existen casillas especiales en las que respondiendo correctamente se gana una pieza triangular que encaja en la ficha del jugador. Gana la partida aquel que completa su ficha consiguiendo seis piezas triangulares contestando a una pregunta de cada tema en las casillas especiales.

«Trivial Pursuit» dio el salto a los ordenadores durante la década de los 80. Desde entonces, se han lanzado diferentes versiones para multitud de plataformas, habiendo vendido más de un millón y medio de unidades desde el año 2004 [9].

Por último, cabe mencionar el caso de *«Atriviate»*, un clon de *«Trivial Pursuit»* para teléfonos móviles desarrollado por la empresa española Pandereta Estudio. Este juego gratuito, que cuenta con más de cinco millones de descargas [7], permite a los usuarios de teléfonos móviles jugar partidas con sus amigos o rivales aleatorios.

Características deseables de los videojuegos

De las características y la mecánica de los videojuegos se obtienen algunas propiedades que una plataforma para la gestión integral de concursos televisivos debe soportar:

- Gestión de diversos tipos de preguntas multimedia (vídeo, imagen, sonido, etc.).
- Soporte para *networking*.
- Posibilidad de despliegue de gráficos en 3D.
- Capacidad para generar ejecutables multiplataforma.
- Separación entre capa de almacenamiento de preguntas y capa de despliegue gráfico, permitiendo que diferentes dispositivos accedan al mismo conjunto de preguntas.
- Gestión eficiente y robusta de fuentes *TrueType* con ajuste automático a la resolución de pantalla.
- Gestión de ambiente según el estado de juego (música de fondo, efectos de sonido, etc.).

En esta sección se han analizado los elementos fundamentales del diseño de los juegos con una mecánica similar a un concurso televisivo. En la siguiente sección se analiza el caso particular de los videojuegos educativos.

3.1.3. Juegos educativos

Los juegos educativos tienen como propósito el enseñar nuevos conocimientos o entrenar habilidades y capacidades.

Aunque habitualmente se los relaciona con la enseñanza infantil, los videojuegos educativos se han utilizado en muchos otros campos, destacando la enseñanza de idiomas y el tratamiento de personas con enfermedades de la memoria como el Alzheimer.

Un problema asociado a este tipo de videojuegos es la falta de motivación debida a su escasa capacidad de diversión. Esto ha dado origen a varios estudios [23] gracias a los cuales se han obtenido modelos y herramientas para medir la jugabilidad.

A continuación se describen algunos ejemplos de juegos con fines educativos.

Figura 3.8: Captura de *«Brain Training»* obtenida de [16].

«Brain Training»

«Brain Training» (*«Brain Age»* en regiones NTSC) es un videojuego creado por Nintendo para su consola portátil Nintendo DS. Cuenta con dos entregas, habiendo vendido entre las dos más de treinta y cinco millones de copias en todo el mundo [9].

Su mecánica consiste en superar varios puzles que entrenan diferentes habilidades de la mente (memoria, rapidez, cálculo, etc.). El juego lleva un diario en donde se registran los progresos del jugador en cada una de estas habilidades.

La dificultad dinámica de los puzles unida al registro de los avances del jugador hacen de *«Brain Training»* un juego apto para cualquier persona.

Hay una cierta controversia respecto a los supuestos efectos beneficiosos que se obtienen al jugar a *«Brain Training»* diariamente, existiendo estudios tanto a su favor como en su contra.

«Aprende con Pipo»

«Aprende con Pipo» es una colección de juegos educativos desarrollada a partir de 1995 por Cibal Multimedia. Esta compuesta actualmente por un total de 4893 juegos [8] que abarcan los contenidos y niveles educativos de Infantil y Primaria. Desde el año 2004 cuenta con presencia en Internet.

La mecánica varía en cada juego, existiendo multitud de pruebas diferentes en función de

Figura 3.9: Captura de *«Aprende con Pipo»* obtenida de [8].

la temática del juego. Algunos ejemplos son pruebas musicales, responder a preguntas tipo test, emparejar elementos o realizar cálculos.

Su consistente aspecto visual junto con la enorme variedad de temas que abarcan sus diferentes entregas hacen de *«Aprende con Pipo»* una valiosa herramienta didáctica.

Características deseables de los juegos educativos

Del estudio de los juegos educativos también se extraen algunas características que debe soportar una plataforma para el soporte de concursos televisivos:

- Posibilidad de jugar con otras capacidades, además del conocimiento específico, como la rapidez de respuesta, la memoria visual, la identificación de patrones, etc.
- Tratamiento de mecanismos de enfrentamiento metal: cálculo, capacidad visual, etc.
- Diferenciación de las pruebas en función del nivel de enseñanza además del tema.
- Nivel de dificultad adecuado para suponer un reto estimulante que no produzca frustración.

En las anteriores secciones se ha definido el concepto de *gameplay* y se han analizado las características y la mecánica de concursos televisivos, videojuegos y juegos educativos a partir de los cuales se han definido las características generales que debe soportar una plataforma como *QuQuSI*. En la siguiente sección se estudian conceptos de la ingeniería del software que se aplicarán en el desarrollo del presente Proyecto Fin de Carrera.

3.2. INGENIERÍA DEL SOFTWARE

Para desarrollar una plataforma como *QuQuSI* se requieren conocimientos sobre ingeniería del software.

La ingeniería del software es *la aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada requerida para desarrollar, operar (funcionar) y mantenerlos* [33].

Así, para emprender cualquier tipo de proyecto software de cierta envergadura, se han de tener unos conocimientos generales de ingeniería del software a fin de conseguir un producto con unos estándares de calidad satisfactorios en un tiempo limitado y con unos costes acotados.

La ingeniería del software incluye técnicas, metodologías y procedimientos desde el más alto nivel (ciclo de vida software, metodologías y procesos de desarrollo, etc.) hasta niveles más cercanos al código (diagramas UML, patrones de diseño, etc.).

Otro aspecto que cuida la ingeniería del software es el de las pruebas. Existen multitud de técnicas que permiten encontrar fallos en el código con un mayor o menor nivel de cobertura y profundidad.

Además, la ingeniería del software incluye en la definición de software la documentación que lo acompaña, haciendo así un especial hincapié en su importancia.

A continuación se estudian algunos de los principales conceptos de la ingeniería del software de utilidad para la elaboración del presente Proyecto Fin de Carrera.

3.2.1. Procesos de desarrollo

Un proceso consiste en una serie de pasos que involucran actividades, restricciones, recursos, herramientas y técnicas que producen una determinada salida esperada [26].

Cuando la salida de un proceso es un producto software se le denomina ciclo de vida del software. Así, un proceso de desarrollo software es *el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software* [29].

En general, los procesos cuentan con las siguientes características:

- Establece a todas sus actividades principales.
- Utiliza recursos, está sujeto a restricciones y genera productos tanto intermedios como finales.
- Puede estar formado por subprocesos encadenados, definiéndose así una jerarquía de procesos, teniendo cada uno su propio modelo.
- Las actividades que conforman el proceso cuentan con criterios de entrada y salida, de modo que se sabe cuando empieza y cuando acaba cada una de ellas.
- Las actividades se organizan secuencialmente, quedando claro el orden relativo de cada una respecto a las demás.
- Las metas de cada actividad se explican en un conjunto de principios orientadores.
- Los controles del proceso pueden aplicarse a las actividades, los recursos o los productos.

Los procesos de desarrollo software pueden agruparse en dos categorías, los procesos pesados y los procesos ligeros. Los procesos pesados se caracterizan por generar una mayor cantidad de documentación, mientras que los procesos ágiles favorecen la comunicación entre los miembros del equipo de desarrollo y los clientes frente a la elaboración de documentos.

A continuación se realiza un breve repaso de un ejemplo representativo de cada una de estas categorías:

- **Proceso Unificado:** El Proceso Unificado de Desarrollo de Software es un marco de trabajo genérico que puede especializarse, permitiendo su aplicación en multitud de sistemas software, diferentes áreas de aplicación, tipos de organizaciones y tamaños de proyecto [29].

  Al estar basado en componentes, el sistema software resultante del proceso está compuesto por componentes que se comunican mediante interfaces bien definidas.

  El Proceso Unificado utiliza el Lenguaje Unificado de Modelado (*Unified Modeling Language*, UML) para la realización de los diagramas.

  Los aspectos que mejor definen al Proceso Unificado son:

- **Dirigido por casos de uso:** El sistema software resultante del proceso se origina a partir de las necesidades de sus usuarios, por lo que se han de conocer las necesidades de estos. Un caso de uso representa un requisito funcional del sistema. El conjunto de todos los casos de uso conforma el modelo de casos de uso, el cual describe la funcionalidad total del sistema. Los casos de uso se desarrollan de forma simultánea a la arquitectura del sistema.
- **Centrado en la arquitectura:** La arquitectura de un software se describe mediante diferentes vistas en las que se incluyen las características más importantes del sistema. Es una visión completa del sistema en la que se omiten los detalles. La arquitectura debe permitir el desarrollo de todos los casos de uso, evolucionando ambos aspectos en paralelo.
- **Iterativo e incremental:** Las iteraciones son secuencias de pasos en el flujo de trabajo mientras que los incrementos se refieren al crecimiento del producto. El Proceso Unificado divide el proyecto en varios subproyectos. Cada uno de estos subproyectos es una iteración que resulta en un incremento. En cada iteración se identifican y especifican los casos de uso más relevantes siguiendo un orden lógico. Un desarrollo iterativo e incremental reduce los riesgos, aumenta la eficiencia y permite refinar los requisitos del usuario.

El Proceso Unificado cuenta con cinco flujos de trabajo (requisitos, análisis, diseño, implementación y pruebas) los cuales tienen lugar sobre cuatro fases (inicio, elaboración, construcción y transición). Cada iteración del proceso atraviesa los cinco flujos de trabajo, repartiéndose la carga de trabajo de distinta forma en función de la fase en la que se desarrolla la iteración (Fig. 3.10).

En la primera fase se buscan las principales funciones del sistema, se plantea la arquitectura del mismo, se desarrolla el plan del proyecto y se realiza un estudio de viabilidad que incluye el tiempo y el coste estimado.

En la fase de elaboración se especifican la mayor parte de los casos de uso y se diseña la arquitectura del sistema.

Durante la fase de construcción se lleva a cabo la implementación del producto. Esta fase finaliza cuando el producto resultante es adecuado para una primera entrega.

En la fase de transición se presenta el producto a un conjunto reducido de usuarios expertos que realizan pruebas para buscar errores y defectos. Los desarrolladores corrigen los fallos encontrados e implementan las mejoras finales sugeridas por los usuarios.

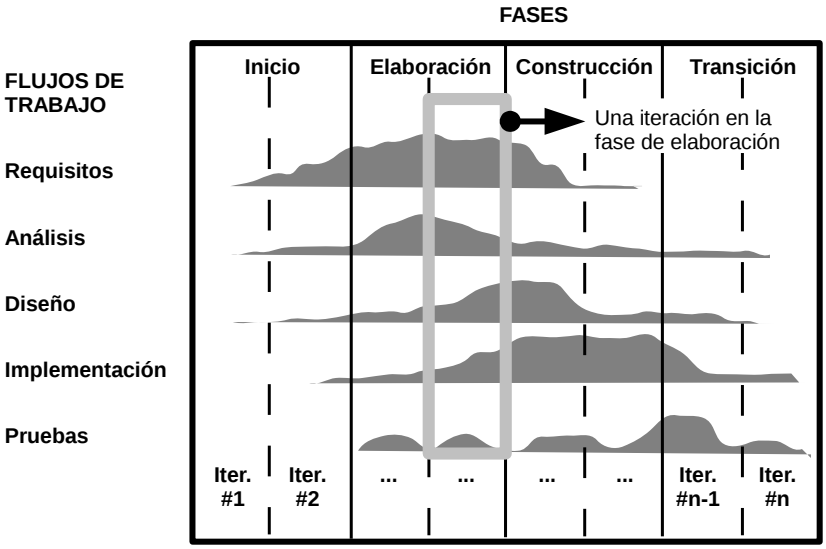

Figura 3.10: Fases, flujos de trabajo e iteraciones en el Proceso Unificado [29].

Por último, cabe mencionar que el Proceso Unificado puede adaptarse en función del tamaño y la complejidad del sistema, permitiendo su aplicación al desarrollo de cualquier tipo de software.

- **Programación Extrema:** La Programación Extrema es un proceso ágil que utiliza la retroalimentación frente a la documentación como mecanismo de control principal. Esta retroalimentación está dirigida por pruebas regulares y entregables o diferentes versiones del software. En la figura (Fig. 3.11) se puede comprobar el orden de eventos en la Programación Extrema como una serie de entregables.

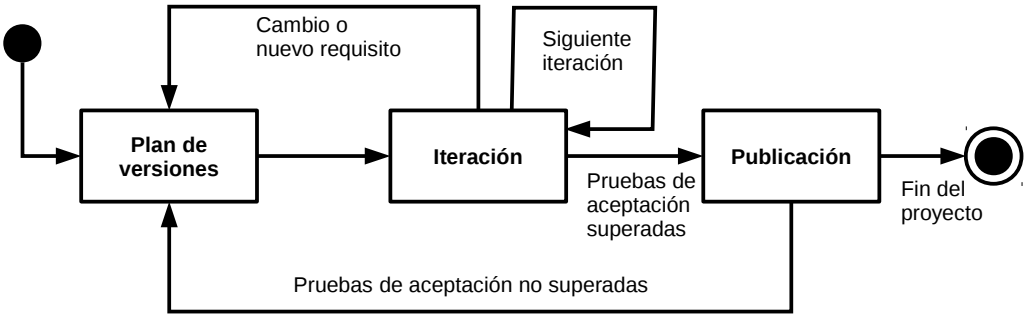

Figura 3.11: Orden de eventos en la Programación Extrema [30].

La Programación Extrema se rige por cuatro valores centrales (comunicación, simplicidad, retroalimentación y valor). De estos valores se derivan doce prácticas las cuales

son aspectos metodológicos del proceso. Gracias a estas prácticas el producto software que se desarrolla está en consonancia con los cuatro valores centrales [30].

A continuación se describe el significado de cada uno de los valores centrales de la Programación Extrema:

- **Comunicación:** En la Programación Extrema se valora la comunicación entre las personas implicadas en el proyecto (desarrolladores y usuarios) frente a la elaboración de documentos, como ocurre en los procesos pesados. El propósito de este valor es lograr que tanto desarrolladores como usuarios compartan la visión del sistema.
- **Simplicidad:** Se fomenta la búsqueda de soluciones simples que más tarde serán mejoradas mediante la refactorización. La Programación Extrema centra el diseño y la programación en las necesidades actuales frente a las futuras.
- **Retroalimentación:** Los proyectos de Programación Extrema se guían mediante la retroalimentación, la cual se relaciona con las diferentes dimensiones del desarrollo. Los desarrolladores necesitan retroalimentación directa acerca del estado del sistema tras realizar cambios en la implementación. Los usuarios necesitan retroalimentación acerca del estado de su sistema de modo que puedan supervisar el desarrollo.
- **Valor:** El último valor significa que desarrolladores, clientes y gestores deben estar dispuestos a probar nuevas ideas y aproximaciones, pudiendo dar pasos atrás en el desarrollo si se detectaran mejores caminos para la construcción del sistema. Gracias a los valores de comunicación y retroalimentación, cualquier problema que surgiese por arriesgarse a tomar un camino diferente se haría evidente a tiempo.

En los proyectos de Programación Extrema el flujo se divide en versiones que son creadas de forma sucesiva y rápida. Antes de implementar cada versión, se comienza especificando un plan de versiones. El desarrollo de cada versión se realiza de forma iterativa y termina con unas pruebas de aceptación que, de superarse, suponen una entrega (Fig. 3.12).

Cada iteración, que suele tener una duración de varias semanas, comienza con un plan de iteración.

Siguiendo el plan de iteraciones, se forman parejas de programadores que cooperaran durante la duración de la iteración, cambiando estas parejas entre iteraciones.

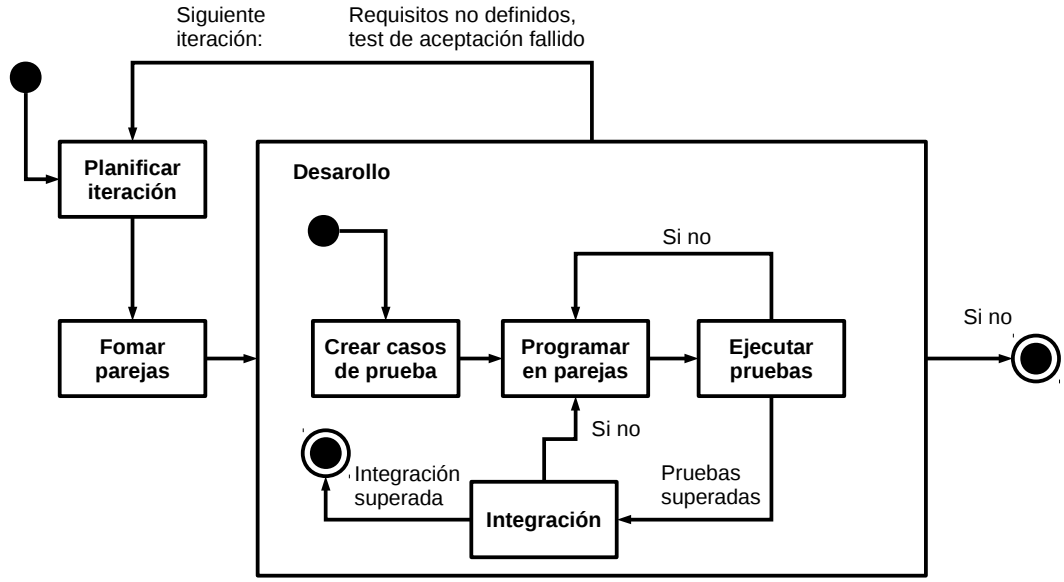

Figura 3.12: Eventos dentro de una iteración de la Programación Extrema [30].

Cada pareja comienza la implementación escribiendo casos de prueba, las cuales formalizan los requisitos.

Cuando las pruebas establecidas al comienzo de la iteración son superadas exitosamente, la pareja de programadores habrá terminado su labor y podrá integrar el resultado de su trabajo en el proyecto. Esta integración continua es otra de las características de la Programación Extrema gracias a la cual se asegura que el progreso del trabajo es visible a todo el equipo del proyecto.

Para la realización del presente Proyecto Fin de Carrera se ha seguido el Proceso Unificado de Desarrollo.

Los motivos para elegir este proceso son:

- Capacidad para adaptarse a proyectos y equipos de desarrollo de cualquier tamaño
- Guiarse por casos de uso, lo que permite que el software desarrollado cumpla con los requisitos.
- Estar orientado a la arquitectura, produciendo un código de mayor calidad.
- Carácter iterativo e incremental, de forma que se pueden implementar las diferentes partes del sistema en sucesivas iteraciones.

Los procesos ligeros se descartaron principalmente por estar optimizados para equipos de desarrollo de varios miembros.

En esta sección se han analizado procesos de desarrollo de diferentes tipos. En la siguiente sección se estudian los patrones de diseño.

3.2.2. Patrones de diseño

Un patrón describe un problema recurrente así como su solución, de modo que se pueda aplicar esta de forma sistemática cada vez que se presenta el problema [27].

Los patrones también se pueden aplicar al diseño de sistemas orientados a objetos. Su finalidad es la de conseguir que las soluciones utilizadas en sistemas específicos sean lo suficientemente generales como para poder ser reutilizadas. No solo benefician a la reutilización, sino que también mejoran la flexibilidad, elegancia y mantenibilidad del sistema.

Cada patrón de diseño cuenta con cuatro elementos esenciales [28]:

- **Nombre del patrón:** Permite referirse a un problema de diseño y su solución mediante una o dos palabras. Incorporar el nombre del patrón al vocabulario del diseñador le da la capacidad de entenderse con sus compañeros, ya sea hablando entre sí o mediante la documentación escrita.

- **Problema:** Se trata de una descripción del problema al que da solución el patrón. Puede expresarse tanto de modo concreto como con los síntomas de un diseño inflexible que podría mejorar mediante la aplicación del patrón. También incluye una lista de condiciones necesarias que justifican el uso del patrón.

- **Solución:** En ella se describen los elementos, las relaciones, las responsabilidades y las colaboraciones que constituyen el diseño. Se trata de una plantilla que puede aplicarse en muchas situaciones diferentes. Proporciona así una descripción abstracta del problema y una disposición general de elementos que lo resuelve.

- **Consecuencias:** Enumeran las ventajas e inconvenientes de la utilización del patrón. Gracias a ellas se pueden valorar las diferentes alternativas existentes a la hora de elegir un patrón determinado. Incluyen el impacto del patrón sobre la flexibilidad, la extensibilidad y la portabilidad de un sistema.

Los patrones de diseño se clasifican en función de dos criterios [28], su ámbito y su propósito. El ámbito se refiere a si el patrón se ocupa de las relaciones entre clases o entre objetos. El propósito se refiere al tipo de problema y solución con los que trata el patrón. Según su propósito, se clasifican en:

- **Creación:** Estos patrones proporcionan soluciones relacionadas con la construcción de clases, objetos y otras estructuras de datos. Ejemplos de patrones de creación son *Factory Method*, *Abstract Factory* y *Singleton*.

- **Estructurales:** Tratan sobre la jerarquía y organización de las clases y las relaciones composiciones entre objetos que proporcionan un buen diseño en un cierto contexto. Ejemplos de este tipo de patrón son *Adapter*, *Decorator*, *Facade* y *Proxy*.

- **Comportamiento:** Describen la comunicación establecida entre objetos mediante envío de mensajes y la organización en las llamadas a los diferentes métodos para realizar una tarea de una forma adecuada. Algunos patrones de comportamiento son *Interpreter*, *Command*, *Iterator*, *Observer*, *State* y *Visitor*.

En el desarrollo de *QuQuSI* se han tenido en cuenta los patrones de diseño a la hora de buscar solución a los problemas encontrados.

En particular, se han utilizado los siguientes patrones:

- ***Singleton:*** Este patrón garantiza que solo exista una instancia de una clase durante la ejecución. Puede implementarse de diferentes maneras; habitualmente a través de un miembro estático en la clase y declarando el constructor como privado. El patrón se ha utilizado en aquellas clases en las que la duplicidad de instancias resultase redundante o pudiera provocar estados inconsistentes, como en los gestores de recursos o en los estados del concurso.

- ***State:*** Este patrón permite diseñar una clase como si se tratara de una máquina de estados. Mediante la herencia y el uso de interfaces se implementan clases que representan los estados. Se ha utilizado para diseñar la lógica del concurso, siendo las señales que recibe la máquina de estados las señales de control y los eventos.

- ***Observer:*** Se utiliza para que un sujeto (*subject* notifique a una serie de observadores (*observers*) suscritos a el. Se ha empleado en las tareas de comunicación, para leer de forma síncrona los mensajes de los *sockets* y para que las aplicaciones cliente sean notificadas de los cambios de estado del concurso.

En esta sección se han estudiado los patrones de diseño. En la siguiente sección se estudia la algoritmia y las diferentes familias de algoritmos.

3.2.3. Algoritmia

Un algoritmo es un conjunto ordenado y finito de normas u órdenes que permiten efectuar un cálculo o encontrar la solución de un problema. La algoritmia consiste en el estudio de los algoritmos [34].

La algoritmia permite evaluar el efecto de factores externos sobre los algoritmos disponibles de modo que se pueda elegir el que resulte más adecuado. Algunos de estos factores externos son el modo de representación del problema, las estructuras de datos y la capacidad de cálculo y almacenamiento de un computador.

También índica la forma de diseñar nuevos algoritmos que lleven a cabo tareas concretas.

La algoritmia presenta numerosas ventajas. Algunas de las principales son:

- Permite medir la eficiencia de los algoritmos. Esto se hace en función de diferentes recursos, como pueden ser el tiempo o el espacio disponibles.
- Gracias a lo anterior se pueden comparar los diferentes algoritmos y aplicar el más adecuado en cada caso.
- Aporta una serie de indicaciones con las que diseñar nuevos algoritmos. De este modo se pueden resolver problemas a los que no dan solución los algoritmos disponibles.

Existen varias familias de algoritmos. Algunas de ellas son:

- **Algoritmos voraces:** Se trata de algoritmos que buscan una solución utilizando los datos disponibles de forma inmediata sin tener en cuenta efectos futuros. Son fáciles de inventar, implementar y eficientes. Suelen utilizarse en problemas de optimización. Su mayor inconveniente es que la mayoría de problemas no puede solucionarse con un enfoque tan simple.
- **Divide y vencerás:** Más que una familia de algoritmos se trata de una técnica de diseño que consiste en descomponer el problema en subproblemas más pequeños que puedan resolverse de forma sucesiva e independiente. Con ellos se obtiene una buena eficiencia, pero son difíciles de concebir e implementar.
- **Programación dinámica:** Consiste en mantener una tabla de resultados conocidos de forma que no se realicen cálculos redundantes durante la ejecución del algoritmo. De esta forma logran mejor eficiencia pero tienen un mayor consumo de espacio en memoria.

- **Exploración de grafos:** Se trata de algoritmos que permiten resolver problemas que se representan mediante grafos. Se aplican cuando el orden en el que se visitan los nodos es irrelevante. Su eficiencia depende del modo en que se recorra el grafo.
- **Algoritmos probabilistas:** Se basan en tomar un curso de acción aleatorio frente a invertir tiempo en determinar la siguiente acción. Su principal característica es que, dada su aleatoriedad, se comportan de distinto modo al aplicarlo varias veces para resolver un mismo problema.

En el desarrollo de *QuQuSI* se han utilizado los algoritmos voraces.

Particularmente, se han aplicado en el proceso de selección dinámica de preguntas, recuperándose la pregunta más adecuada disponible en función de una serie de parámetros que dependen del estado del concurso. Las ventajas de este algoritmo son:

- Se corresponde con el problema de obtener el elemento con la mejor valoración de una lista.
- Complejidad lineal, se ejecuta en un tiempo razonable que no produce retrasos en el concurso.
- En el caso del algoritmo diseñado la solución es óptima; la pregunta que se recupera es la más adecuada de todas las disponibles.

En esta sección se ha estudiado el concepto de algoritmia y se han analizado las diferentes familias de algoritmos. En la siguiente sección se analiza el paradigma de la programación modular.

3.2.4. Programación modular

Se trata de un paradigma de programación que consiste en dividir la funcionalidad de un sistema en módulos. Cada uno de estos módulos contiene el código fuente y las variables necesarias para llevar a cabo la función que le corresponde.

Gracias a la programación modular se resuelven los problemas de mantenimiento y detección de errores de programas grandes. Al dividir el código en módulos es más fácil aislar y detectar que parte de este produce los errores [10].

La programación modular ofrece numerosas ventajas:

- **Menor acoplamiento:** La utilización de módulos permite disminuir el acoplamiento, ya que al definirse de forma clara la funcionalidad de cada uno de ellos se aísla mejor del resto de módulos y solo se comunica con aquellos con los que es necesario.

- **Mayor cohesión:** Al contener todo el código fuente y las variables necesarias para satisfacer la funcionalidad de la que se encarga el módulo se aumenta la cohesión. El módulo incluirá únicamente las funciones, métodos y variables necesarias para llevar a cabo su cometido.
- **Reutilización:** Como consecuencia directa de las anteriores ventajas, el código resultante de una programación modular es más fácil de reutilizar, pudiendo además ser sustituido sin tener que modificar el resto de partes del programa.
- **Mantenibilidad:** Gracias a que es más sencillo aislar y detectar los errores en la programación modular se mejora la mantenibilidad del programa. Adicionalmente, al contener tan solo el código fuente y las variables estrictamente necesarias para cumplir con su cometido, los módulos son entendidos más fácilmente por otros programadores.

Durante el desarrollo de *QuQuSI* se ha aplicado el paradigma de la programación modular:

- Al tratarse *QuQuSI* de un sistema distribuido, permite la reutilización de componentes entre las aplicaciones que lo forman.
- La mantenibilidad y extensibilidad del código es mayor, lo que permite futuras ampliaciones de la plataforma.
- La cohesión aumenta, permitiendo una mayor facilidad en la depuración y detección de errores.

En esta sección se ha estudiado el paradigma de la programación modular. En la siguiente sección se analizan las pruebas de software.

3.2.5. Pruebas

Como no existe software carente de errores, las pruebas son una parte fundamental en el desarrollo de cualquier sistema. La intensidad de las pruebas a las que se someta el software dependerá principalmente de lo crítica que sea la funcionalidad de éste.

El objetivo de las pruebas no es demostrar que un sistema está exento de fallos, sino encontrar errores. Si tras la ejecución de las pruebas no se encuentra ningún error en el sistema es muy posible que estas estén mal diseñadas. Además, que todas las pruebas sean ejecutadas con éxito no supone una demostración de que el sistema sea correcto; solo demuestran que el sistema tiene el comportamiento deseado en todas las situaciones contempladas.

Las pruebas del software aportan numerosas ventajas, entre las que se encuentran:

- Permiten detectar errores en el código durante todas las fases del desarrollo.
- Es posible dirigir el desarrollo del software mediante las pruebas (*Test Driven Development*, TDD).
- Pueden volver a ejecutarse en cualquier momento, permitiendo detectar errores introducidos en nuevas versiones.
- Ayudan a descubrir nuevos requisitos y a corregir los existentes.

Las pruebas se realizan en diferentes niveles del desarrollo, con diferentes enfoques y utilizando criterios de cobertura del código y valores en las variables que favorezcan la detección de errores.

En función del nivel del desarrollo en el que se realizan las pruebas se tienen los siguientes tipos:

- **Pruebas unitarias:** Se buscan errores a nivel de unidad de prueba, siendo esta unidad la clase en el caso de la programación orientada a objetos. Es el nivel más bajo de prueba y se realiza para cada unidad previamente a su integración en el sistema.
- **Pruebas de integración:** El siguiente nivel de prueba se encarga de localizar fallos en la interacción entre dos o más unidades o componentes del sistema. Estas pruebas se realizan al integrar nuevos módulos al código existente previamente probado.
- **Pruebas de sistema:** Se trata de pruebas de más alto nivel en las que se involucran módulos complejos o sistemas completos entre sí.
- **Pruebas funcionales:** En el nivel más alto se ejecutan las pruebas funcionales, las cuales deben garantizar que el sistema cumple con los requisitos funcionales al tener el comportamiento deseado en las situaciones contempladas. Estás pruebas son idealmente realizadas por los usuarios finales del sistema.

En cuanto al enfoque de las pruebas, se tienen los siguientes tipos:

- **Caja negra:** Se prueba las entradas y las salidas del sistema, sin importar como lleva a cabo la funcionalidad. Para cada entrada se comprueba que la salida sea la deseada; de no ser así, se habrá detectado un error. Como normalmente el número de entradas diferentes es muy grande no puede garantizarse que el sistema se comporte de forma correcta siempre. Sin embargo, existen técnicas que permiten elegir los casos más susceptibles de encontrar errores:

  - **Clases de equivalencia:** Con este método se dividen los posibles valores de entrada en clases de datos a partir de los que se pueden derivar casos de prueba. Cada uno de estos casos debe cubrir el mayor número de entradas posible. Los valores agrupados en cada clase de datos deben producir una misma salida en el sistema.
  - **Valores límite:** Esta técnica, que complementa a la anterior, consiste en elegir los valores en el límite de cada clase de equivalencia. De este modo, se abarcan todas las clases y se prueban los limites, en donde es más habitual encontrar errores. No solo se prueban los valores límite en las entradas si no también en las salidas.
  - **Conjetura de errores:** Se trata de utilizar el sentido común y la intuición para diseñar casos de prueba que busquen errores típicos o predecibles que el programador pueda haber introducido en el código.

- **Caja blanca:** Se prueba la funcionalidad del sistema atendiendo a los diferentes caminos que puede llevar la ejecución del código. Surge así el concepto de cobertura o cantidad de caminos o bifurcaciones que han sido probadas. Gracias a la cobertura se puede determinar el porcentaje en el que se ha probado el código del sistema. Existen diferentes criterios de cobertura que permiten probar mayor o menor cantidad de código:

  - **Sentencias:** Se generan los casos de prueba suficientes como para que cada sentencia o instrucción del programa se ejecute al menos una vez.
  - **Decisiones:** Se tendrán los casos de prueba necesarios para que cada decisión del código tenga como mínimo una evaluación verdadera y otra falsa.
  - **Condiciones:** Se requieren casos de prueba que logren que cada condición dentro de cada decisión tome al menos una vez el valor verdadero y una vez el valor falso.
  - **Decisión/condición:** Consiste en tener casos de prueba que cumplan con los dos criterios anteriores al mismo tiempo.
  - **Condición múltiple:** Se diseñan los casos de prueba suficientes como para que se den todas las combinaciones de valores posibles en las condiciones de las decisiones.

El software de *QuQuSI* se ha probado siguiendo varias de estas técnicas:

- Con las pruebas unitarias se han probado los módulos implementados.

- Se han llevado a cabo pruebas de integración para combinar los módulos en cada aplicación.
- Las pruebas de sistema se han empleado para probar el funcionamiento de las aplicaciones entre sí.

Cabe destacar las pruebas funcionales que se realizaron con el sistema completo con ayuda del personal de la Escuela Superior de Informática de Ciudad Real.

En las anteriores secciones se han estudiado diferentes conceptos de la ingeniería del software aplicables en la elaboración del presente Proyecto Fin de Carrera. En la siguiente sección se estudian algunos elementos del *networking* útiles en el desarrollo de *QuQuSI*.

3.3. *NETWORKING*

Una plataforma distribuida como *QuQuSI* requiere una infraestructura de comunicación que permita controlar y sincronizar las partes que la componen.

Será necesario desplegar varias aplicaciones Web para gestionar las preguntas, los concursantes y sus respuestas. También se necesitará un mecanismo de sincronización que propague las señales de control, los eventos y el estado del concurso entre los elementos del sistema.

Para el despliegue de aplicaciones Web se han analizado los servidores Web. Para las tareas de comunicación y sincronización se han estudiado los *sockets*.

3.3.1. Servidores Web

Un servidor Web es un software para la publicación de contenidos y servicios a través de Internet. También puede referirse a la máquina donde se ejecuta dicho software.

Los servidores Web se utilizan principalmente para albergar sitios Web. El servidor se mantiene a la escucha en espera de las peticiones de los clientes. Una vez recibe una solicitud, la atiende mediante algún protocolo de comunicación, siendo el más habitual HTTP (*Hypertext Transfer Protocol*).

Algunas de las ventajas que ofrecen los servidores Web son:

- Pueden alojar varias páginas Web en una misma dirección IP (*Virtual Hosting*).
- Soportan archivos de gran tamaño, pudiendo atender peticiones de hasta 2GB.
- Gestionan el ancho de banda, impidiendo que las solicitudes grandes bloqueen a las pequeñas (*bandwidth throttling*).

- Son capaces de interpretar lenguajes de script del lado del servidor, generando así documentos dinámicos.

Existen multitud de servidores Web libres. Algunos de ellos son:

- ***Apache*:** Se trata del servidor HTTP más popular de Internet, siendo el más utilizado desde 1996 [11]. Es de código abierto y está disponible para multitud de sistemas operativos, incluyendo *GNU/Linux* y *Windows*. Una de sus principales características es que cuenta con un gran número de módulos que permiten ampliar su funcionalidad.

- ***Nginx*:** Es un servidor HTTP de código abierto y alto rendimiento. Es estable, fácil de configurar y consume pocos recursos. Su principal peculiaridad es que atiende las peticiones de forma asíncrona en lugar de crear hilos para cada solicitud como es común en la mayoría de servidores Web.

- ***Lighttpd*:** Es otro servidor de código abierto optimizado para entornos de alto rendimiento. Su bajo impacto en la memoria del sistema y su gestión del uso del procesador lo convierten en una buena alternativa para solucionar problemas de carga. Gracias a su infraestructura de entrada/salida de alta velocidad es más escalable que otros servidores.

Nginx y *Lighttpd* son más ligeros que *Apache*, pudiendo atender más peticiones ejecutándose sobre el mismo hardware. Sin embargo, *Apache* cuenta con una comunidad de desarrollo mayor y una gran cantidad de módulos que permiten extender su funcionalidad.

Para el desarrollo de *QuQuSI* se ha elegido el servidor Web *Apache*.

Los principales motivos que han llevado a esta decisión son el mejor soporte y la variedad de módulos disponibles.

En esta sección se han estudiado los servidores Web. En la siguiente sección se estudian los diferentes tipos de *sockets*.

3.3.2. *Sockets*

Los *sockets* son una abstracción a través de la cual una aplicación puede enviar y recibir datos, análogamente a como ocurre con la lectura y escritura de archivos [32].

Un *socket* permite que aplicaciones conectadas a una misma red se comuniquen entre sí. La información escrita en el *socket* por una aplicación en una máquina puede ser leída por otra.

Las principales ventajas de la utilización de *sockets* son:

- Ofrecen una interfaz de comunicación abstrayéndose de los detalles de las capas inferiores.

- Permiten el direccionamiento entre máquinas y aplicaciones. Cada *socket* se identifica de forma única mediante una dirección IP, un número de puerto y un protocolo.

- Pueden utilizarse como canales orientados a conexión o a datagramas, gracias a los protocolos TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*) respectivamente.

Respecto a este último punto, cabe mencionar las propiedades más importantes de cada uno de estos protocolos:

- **TCP:** Se trata de un protocolo que ofrece un canal orientado a conexión. Antes de utilizarse se ha de establecer una conexión. Se encarga de detectar errores y recuperarse de estos, garantizando que los datos llegan y se entregan en el orden en que fueron enviados. De este modo, las aplicaciones que utilizan *sockets* TCP no tienen que preocuparse de estos detalles.

- **UDP:** Es un protocolo que ofrece un canal orientado a datagramas, esto es, paquetes de datos independientes. El protocolo no requiere establecer una conexión previa para enviar los datos. No garantiza la entrega de los datos, de modo que las aplicaciones que utilizan *sockets* UDP deben estar preparadas para tratar con la pérdida y la corrupción de los datos.

El protocolo TCP garantiza la entrega de los datos en el destino a costa de un peor rendimiento provocado por el establecimiento de la conexión y la detección y recuperación de los errores. Por su parte el protocolo UDP no asegura la entrega de los datos pero es más eficiente, por lo que se utiliza en aquellos casos en los que la información no es crítica o pueden inferirse los datos perdidos.

Para la implementación de *QuQuSI* se han utilizado *sockets* TCP.

El principal motivo es evitar la pérdida de señales de control y sincronización, lo que podría llevar al concurso a un estado inconsistente. Además, los mensajes son pequeños y se envían con poca frecuencia, por lo que no hay que temer por una bajada de rendimiento debida a la comprobación y recuperación de errores del protocolo.

En las anteriores secciones se han analizado los servidores Web y los *sockets*. En la siguiente sección se estudian las diferentes alternativas para el almacenamiento de datos.

3.4. ALMACENAMIENTO DE DATOS

Para gestionar de forma integral un concurso televisivo se ha de almacenar una gran cantidad de datos, desde las preguntas que se realizan junto con los archivos multimedia relacionados hasta las respuestas introducidas por el público, así como otra información relevante relacionada con la seguridad y el *workflow* de la plataforma, como son las cuentas de usuario, las claves cifradas y los permisos.

Para almacenar esta información de forma adecuada se han estudiado las bases de datos y los sistemas de archivos.

Gracias a las bases de datos se mantiene la información necesaria para gestionar de forma integral el concurso televisivo. Los archivos multimedia y la información exportada desde la base de datos se guardarán en un sistema de archivos.

3.4.1. Bases de datos

Una base de datos es *una colección compartida de datos relacionados lógicamente y su descripción, designados para satisfacer las necesidades de información de una organización* [31].

Una base de datos puede considerarse como un repositorio de datos con capacidad para ser utilizado de forma simultánea por varios usuarios. La descripción de estos datos se conoce como diccionario de datos o metadatos. Son estos metadatos los que proveen a las aplicaciones de bases de datos de independencia de los datos, al proporcionar una definición externa que abstrae de cómo se defina la información internamente. Esto se conoce como abstracción de los datos. Dentro de una base de datos se distinguen entidades (representan los objetos del dominio de negocio de los que se quiere almacenar información) las cuales cuentan con atributos (la información que se quiere almacenar sobre cada objeto) y que se enlazan entre sí, formando relaciones. Esto es lo que se conoce como el modelo Entidad-Relación.

El sistema software que permite a los usuarios definir, crear, mantener y controlar el acceso a la base de datos [31] se denomina Sistema Gestor de Base de Datos (*Database Management System*, DBMS). Algunas de las funcionalidades habituales de estos sistemas son:

- **Definición de datos:** La definición de la base de datos se realiza a través de un lenguaje de definición de datos, con el cual se puede definir el tipo, la estructura y las restricciones de los datos.

- **Manipulación de datos:** Gracias al uso de un lenguaje de manipulación de datos, permiten la inserción, actualización, eliminación y recuperación de los datos mediante un lenguaje de consulta.

- **Control de acceso:** Proveen de un sistema de control de acceso que garantiza la privacidad de los datos.

Algunos ejemplos de bases de datos libres son [12]:

- ***MySQL*:** Es la base de datos de código abierto más popular. Soporta diferentes motores de almacenamiento, proporcionando capacidades como las tablas temporales, de ejemplo, de lectura rápida, etc. La documentación disponible es abundante, incluyendo manuales de referencia, libros y artículos en línea. Su núcleo está publicado bajo licencia GPL. Existen varias versiones, siendo algunas de pago.

- ***PostgreSQL*:** Se trata de una de las bases de datos de código abierto más avanzadas. Se distribuye una única versión completamente funcional, frente a las diferentes versiones que *MySQL* ofrece. Sus principales prioridades son la confiabilidad, la consistencia y la integridad. Cuenta con mecanismos de seguridad para controlar el acceso a la base de datos. Está bien documentada y ofrece soporte comercial para vendedores independientes.

Tanto *MySQL* como *PostgreSQL* están disponibles para múltiples plataformas, incluyendo GNU/Linux y Windows. Ambas son de código abierto, gratuitas, flexibles y escalables, pudiendo utilizarse para sistemas de cualquier tamaño, y soportan extensiones para ampliar su funcionalidad.

MySQL puede utilizarse en sistemas empotrados, y su rendimiento puede optimizarse mediante los diferentes motores de almacenamiento de los que dispone en función de las necesidades del sistema.

Por su parte, *PostgreSQL* no da soporte para sistemas empotrados, y su rendimiento se controla mediante su configuración ya que cuenta con un único motor de almacenamiento.

Para el presente Proyecto Fin de Carrera se ha elegido *MySQL*, principalmente por la documentación disponible y la experiencia con el sistema.

En esta sección se han comparado diferentes alternativas de bases de datos para almacenar la información de *QuQuSI*. En la siguiente sección se estudian los sistemas de archivos.

3.4.2. Sistemas de archivos

Un sistema de archivos consiste en una estructura de directorios o carpetas organizadas en la cual se almacenan archivos de diferentes tipos.

El sistema de archivos se encarga de gestionar la creación, borrado y modificación de los archivos, además de fijar el formato físico de la información guardada en los medios de almacenamiento (discos duros, memorias USB, etc.) y gestionar la memoria libre disponible en estos.

Los sistemas de archivos organizan los ficheros en directorios, los cuales forman una estructura jerárquica que facilita la búsqueda a los usuarios.

Existen multitud de tipos de sistemas de archivos. Estos suelen estar asociados a algún sistema operativo. Ejemplos son FAT (*File Allocation Table*), NTFS (*New Technology File System*) o ext (*Extended File System*).

Los archivos que se almacenan en los sistemas de archivos pueden clasificarse en los siguientes grupos:

- **Archivos de texto:** Se trata de archivos de texto plano que pueden ser leídos por las personas mediante algún editor o visualizador de texto. Tienen una codificación que determina el formato con el que se almacenan los caracteres del texto que contienen. Ejemplos de archivos de texto son archivos de código fuente (.cpp, .java), de etiquetas (.html, .xml) o de *script* (.sh, .batch).

- **Archivos multimedia:** Son archivos en los que se almacenan diferentes tipos de contenidos multimedia como imágenes, sonidos y vídeos. Suelen codificarse mediante algún algoritmo de compresión para disminuir su tamaño en disco. Ejemplos de tipos de archivos multimedia son los archivos de imagen (.png, .jpg, .gif), de sonido (.mp3, .wav, .ogg) o de vídeo (.mp4, .mov, .mkv, .ogv).

- **Otros archivos:** Existen otros tipos de archivo, muchos de ellos relacionados con los sistemas operativos, como son los ejecutables, las bibliotecas, los archivos de dispositivos, las tuberías, etc.

En el desarrollo de *QuQuSI* se ha utilizado el sistema de ficheros ext en su cuarta versión (ext4), al ser la última versión disponible para *GNU/Linux*, y se han utilizado tanto archivos de texto como multimedia para almacenar las preguntas del concurso y los contenidos asociados a estas respectivamente.

En las anteriores secciones se han analizado las soluciones disponibles para el almacenamiento de datos. En la siguiente sección se estudian técnicas y conceptos relacionados con la informática gráfica necesarios para llevar a cabo el despliegue visual requerido por *QuQuSI*.

3.5. INFORMÁTICA GRÁFICA

El apartado gráfico es una parte fundamental de la plataforma *QuQuSI*, ya que requiere el despliegue de elementos visuales, contenidos multimedia y composición de imagen.

Se ha realizado un estudio de algunos de los aspectos de la informática gráfica que permitan tal despliegue, como son la técnica de *chroma key* y los gráficos 3D por computador.

3.5.1. *Chroma key*

Se trata de una técnica audiovisual que permite componer, mediante la superposición de capas, dos imágenes o vídeos. Esta composición se logra extrayendo un color determinado de una de las imágenes de modo que se forman áreas "transparentes" a través de las cuales se puede ver la otra imagen (Fig. 3.13).

Figura 3.13: Ejemplo de la técnica *chroma key*.

Esta técnica tiene aplicación en multitud de áreas, pudiéndose destacar el cine, donde se graba a los actores sobre un fondo verde para luego añadir escenarios totalmente creados mediante imagen por computador, o los telediarios, en donde se superpone el presentador al mapa con la información meteorológica. También tiene cabida en los concursos televisivos, donde se superponen elementos como las preguntas, las respuestas u otros efectos con la imagen recogida por las cámaras de vídeo.

Los colores de fondo más utilizados para la utilización de la técnica son el verde y el azul. Esto se debe a que el pigmento de la piel humano es rojo en un mayor porcentaje. De este modo, no se eliminan de la imagen partes no deseadas. Sin embargo, existe el inconveniente de que no se puede usar el color elegido de fondo en ningún elemento de la imagen que se superpondrá, ya que sería eliminado durante el procesado del color.

La técnica se puede implementar tanto en hardware como en software, siendo habitual el uso de una función que compara los valores RGB de cada píxel de la imagen con un umbral.

La técnica de *chroma key* se ha utilizado en el presente Proyecto Fin de Carrera para llevar a cabo la composición de la imagen generada por computador con la obtenida por las cámaras de vídeo.

En esta sección se ha analizado la técnica del *chroma key* con la que se puede combinar la imagen por computador con la obtenida por las cámaras de vídeo. En la siguiente sección se estudian los gráficos 3D por computador y el *pipeline* de representación gráfica.

3.5.2. Gráficos 3D por computador

Los gráficos 3D por computador consisten en la transformación de escenas tridimensionales a imágenes bidimensionales. La escena 3D está formada por un conjunto de mallas (vértices unidos por aristas formando planos). Los objetos que forman la escena atraviesan un *pipeline* o cauce gráfico cuyo resultado es una imagen con una cierta resolución. Este proceso se conoce como *rendering*.

Los gráficos 3D son utilizados en multitud de aplicaciones, especialmente en aquellas interactivas como los videojuegos o la realidad aumentada.

El *pipeline* gráfico que transforma la escena 3D en una imagen bidimensional consta de tres etapas [36]:

- **Etapa de aplicación:** Durante esta etapa se llevan a cabo tareas como la detección de teclas pulsadas o el reposicionamiento de los objetos de la escena en función de los resultados de la simulación física.
- **Etapa de geometría:** En esta etapa los objetos de la escena atraviesan una serie de sistemas de coordenadas y transformaciones que facilitan la conversión de la escena tridimensional en la imagen bidimensional.
- **Etapa de *rasterización*:** En la última etapa se toma la salida de la etapa de geometría y se calcula el valor final de cada píxel de la imagen.

Existen dos tipos de *rendering*:

- ***Pre-rendering*:** Es el que se usa en las películas de animación por computador. En lugar de generar las imágenes en tiempo real, el proceso de *rendering* se lleva a cabo previamente en un equipo de altas prestaciones y se almacena el resultado en un archivo de vídeo.

- ***Real-time rendering*** Es el utilizado en aplicaciones interactivas. Se genera un número de imágenes por segundo suficiente como para dar sensación de movimiento. El resultado visual es de peor calidad pero mucho más rápido.

El *real-time rendering* se utiliza en las aplicaciones interactivas donde la imagen resultante del proceso es determinada por el usuario. Por su parte, el *pre-rendering* es utilizado en situaciones en las que el usuario no tiene el control de la imagen, como en películas o escenas de introducción en los videojuegos.

En *QuQuSI* se ha utilizado el *real-time rendering*.

El motivo es que el concurso al que da soporte la plataforma es una aplicación interactiva.

En las anteriores secciones se han analizado conceptos de la informática gráfica necesarios para satisfacer el despliegue visual requerido por *QuQuSI*. En la siguiente sección se estudian los diferentes *frameworks* que den soporte al resto de antecedentes analizados.

3.6. *FRAMEWORKS*

Puesto que no existe ninguna plataforma libre para la gestión integral de concursos televisivos con las características descritas en el capítulo 2, los *frameworks* que se han analizado están relacionados con los antecedentes descritos anteriormente.

Así, se han analizado *frameworks* que permitan la creación de gráficos 3D por computador, la programación de hardware, el desarrollo de aplicaciones Web y el despliegue de contenidos multimedia.

3.6.1. Motores gráficos 3D

Los motores gráficos son un software que se encargan de transformar una escena tridimensional en una imagen bidimensional.

Se utilizan para desarrollar aplicaciones gráficas como videojuegos, de realidad virtual, de realidad aumentada, etc. Cabe diferenciar un motor gráfico de un motor de juego, encargándose los primeros únicamente del despliegue gráfico mientras que los segundos llevan a cabo otras tareas como la detección de colisiones, la gestión de entrada salida y la reproducción de contenidos multimedia.

Los motores gráficos suelen presentar una serie de características comunes:

- Ofrecen formas de gestionar la escena tridimensional, pudiendo aplicar varias transformaciones a los elementos que la componen.
- Cuentan con mecanismos para calcular la iluminación y diferentes tipos de fuentes de luz.
- Proveen métodos para el cálculo dinámico de sombras.
- Permiten importar modelos de mallas desde multitud de programas de diseño 3D y con diferentes formatos.
- Pueden desplegar paneles 2D básicos y texto mediante *overlays*.

Algunos motores gráficos 3D libres son:

- ***Crystal Space 3D*:** Es un motor de juego completo para el desarrollo de videojuegos multiplataforma. Incluye gestión de sonido (OpenAL), animaciones con esqueletos, iluminación, sombras, efectos de partículas, interfaces de usuario (*CEGUI*), entrada/-salida e importación de modelos de mallas. Cuenta con un motor de física integrado (*Bullet* y *ODE*). Esta escrito en C++ y publicado bajo licencia LGPL.
- ***Irrlitch*:** Se trata de un motor gráfico 3D libre multiplataforma. Soporta animación con esqueletos, iluminación, sombras, efectos de partículas, interfaces de usuario e importación de modelos de mallas. Cuenta con un mecanismo simple de detección de colisiones. Ofrece interfaces en los lenguajes de programación C++ y .NET, aunque soporta otros lenguajes gracias al uso de *wrappers*. Se publica bajo licencia zlib.
- **OGRE 3D:** Es un motor gráfico libre orientado a objetos (*Object-Oriented Graphics Rendering Engine*) multiplataforma [14]. Se centra únicamente en el despliegue gráfico, dependiendo de bibliotecas externas para otros aspectos como la simulación física, la entrada/salida y la reproducción de contenidos multimedia. Así, ofrece soporte para iluminación, sombras, animación basada en esqueletos, sistemas de partículas, importación de modelos de mallas y despliegue básico de *overlays*. Esta publicado bajo licencia LGPL.

Crystal Space 3D es un motor de juego completo al incluir todos los aspectos necesarios para el desarrollo de videojuegos (entrada/salida, sonido, simulación física, etc.). *Irrlitch* y *OGRE 3D* son motores gráficos que se encargan del despliegue visual, dependiendo de bibliotecas externas para el resto de aspectos.

El motor gráfico utilizado en el presente Proyecto Fin de Carrera es *OGRE 3D*.

Se ha elegido este *framework* por su buen diseño, su fácil integración con otras bibliotecas y *frameworks* y la documentación y ejemplos disponibles.

3.6.2. Programación hardware

Estos *frameworks* permiten la programación de microprocesadores, microcontroladores y otro tipo de hardware.

Los microcontroladores se utilizan en multitud de áreas como la robótica o la ingeniería industrial. Permiten que un sistema interaccione con el mundo físico a través de sensores y actuadores.

Algunas características comunes de los *frameworks* de programación hardware son:

- Entorno integrado de desarrollo para la creación de programas mediante diagramas de flujo o lenguajes de programación.
- Carga de programas en la memoria del microcontrolador a través de algún puerto de comunicación (serie, USB).
- Comunicación con el microcontrolador a través de algún puerto de comunicación.
- Visualización del estado de las entradas/salidas de la placa del microcontrolador.
- Simulador del sistema en el que se pueden configurar los distintos dispositivos conectados a la placa.

Algunos de los *frameworks* de programación hardware son:

- ***Fischertechnik*:** Se trata de una marca de juguetes de construcción. Cuenta con una tarjeta controladora programable con diferentes especificaciones en función del modelo. La tarjeta cuenta con entradas y salidas tanto digitales como analógicas. Puede programarse mediante diagramas de flujo utilizando el entorno de desarrollo gráfico que proporciona (LLWin) o con un lenguaje de programación utilizando bibliotecas externas [24]. Esta licenciado bajo Creative Commons.
- ***Arduino*:** *Arduino* es una plataforma para crear prototipos electrónico de código abierto basada en hardware y software flexible y fácil de usar. Consiste en una placa con un microcontrolador y un número de puertos de entrada, salida, alimentación, etc. que dependen del modelo. Provee un lenguaje de programación y un entorno de desarrollo

propios. *Arduino* puede ampliarse mediante placas oficiales que aportan nuevas funcionalidades (*shields*) o con cualquier componente o circuito electrónico compatible con el voltaje de los puertos. Los diseños de *Arduino* se publican bajo licencia Creative Commons.

Con *Fischertechnik* la creación de prototipos es fácil y rápida gracias a la gran variedad de piezas, sensores y actuadores que ofrece. Por contra, es más difícil de integrar con sistemas que no provengan del fabricante.

Realizar prototipos con *Arduino* es menos sencillo al requerir conocimientos básicos de electrónica, pero la libertad de creación es mayor al poder utilizar cualquier componente electrónico compatible con sus entradas y salidas.

Para el presente Proyecto Fin de Carrera se ha utilizado la plataforma *Arduino*.

Se ha tomado esta decisión por su mayor libertad tanto a la hora de desarrollar prototipos hardware como al programar el microcontrolador.

3.6.3. Desarrollo Web

Estos *frameworks* facilitan la creación de sitios Web dinámicos.

Se utilizan para el desarrollo de aplicaciones Web con acceso a bases de datos. Suelen contar con extensiones para cubrir las necesidades habituales de este tipo de aplicaciones (gestión de usuarios, seguridad, interacción con la base de datos, sesiones, etc.).

Algunas características comunes de los *frameworks* de desarrollo Web son:

- Integración con las bases de datos, ofreciendo el conjunto de operaciones CRUD (*Create, Read, Update, Delete*).
- Generación automática de código a partir de la estructura de la base de datos.
- *Helpers* o clases que facilitan la generación de código.
- Soporte para diferentes lenguajes de *script*, tanto del lado del servidor (PHP) como del cliente (AJAX, JavaScript).
- Configuración de las direcciones de las partes del sitio Web (*routing*).

Ejemplos de *frameworks* libres para el desarrollo de aplicaciones Web son:

- ***Ruby on Rails*:** Es un *framework* libre para el desarrollo de aplicaciones Web con acceso a bases de datos basado en el patrón Modelo Vista Controlador (*Model View Controller), MVC*. Es de código abierto y utiliza el lenguaje interpretado Ruby. Puede ampliarse mediante la instalación de *plugins* (*gems*).

- ***CakePHP*:** Es otro *framework* libre para el desarrollo de aplicaciones Web con una arquitectura Modelo Vista Controlador. Está escrito en el lenguaje de *script* PHP y es de código abierto. Proporciona operaciones para acceder a la base de datos. Para instalarlo solo se necesita un servidor Web con capacidad para interpretar PHP.

Ambos *frameworks* son libres y de código abierto, siguen el paradigma Modelo Vista Controlador, proporcionan operaciones de interacción con bases de datos, permiten la generación automática de código y cuentan con una comunidad de desarrollo activa. *Ruby on Rails* requiere la instalación de *gems* mientras que *CakePHP* puede desplegarse en cualquier servidor Web con PHP.

Para desarrollar las aplicaciones Web de la plataforma *QuQuSI* se ha elegido *CakePHP*.

Se ha elegido este *framework* por utilizar el lenguaje PHP, por contar con módulos que facilitan la creación de un *workflow* con diferentes roles y por ser más fácil de instalar, al requerir tan solo un servidor Web con PHP.

3.6.4. Multimedia

Los *frameworks* multimedia se encargan de reproducir archivos multimedia.

Este tipo de archivos suelen estar codificados mediante algún algoritmo de compresión, por lo que es necesario que el *framework* sea capaz de decodificar e interpretar el contenido.

En general, los *frameworks* multimedia cuentan con las siguientes características:

- Decodificación de multitud de formatos de archivos multimedia.
- Lectura y escritura de *streams* de sonido y vídeo.
- Alteración de algunas propiedades como la velocidad de reproducción, volumen y espacios de color.
- Capacidad de mezclar y fundir diferentes fuentes.
- Volcado del resultado de la reproducción en archivos o memoria principal.

Algunos de los *frameworks* multimedia libres disponibles son:

- ***Phonon*:** Se trata de una interfaz de programación de aplicaciones multimedia (*Application Programming Interface*, API). Tiene capacidad para comunicarse con otros *frameworks* multimedia, lo que le da la capacidad de reproducir multitud de formatos en diferentes plataformas. Está escrito en C++ y publicado bajo licencia LGPL.

- ***GStreamer*:** Es un *framework* de desarrollo de aplicaciones multimedia [13]. Permite la creación de aplicaciones multiplataforma gracias a su interconexión con otros *frameworks* multimedia. Su arquitectura basada en la carga de *plugins* permite la creación de *pipelines* capaces de reproducir una gran variedad de formatos. Está escrito en C y publicado bajo licencia LGPL.

Phonon se encuentra un nivel por encima de *GStreamer*, ofreciendo una interfaz básica para la reproducción de contenidos multimedia a través del uso de otros *frameworks* (GStreamer incluido). Por su parte, *GStreamer* permite un tratamiento más detallado de los *streams* y un mayor número de opciones, pudiendo crear cauces personalizados gracias a su arquitectura basada en *plugins*.

Para el despliegue de contenidos multimedia en *QuQuSI* se ha utilizado *GStreamer*.

La decisión se ha tomado por su arquitectura basada en *plugins* y su capacidad para reproducir diferentes formatos.

CAPÍTULO 4. MÉTODO DE TRABAJO

El desarrollo de *QuQuSI* se ha llevado a cabo siguiendo una metodología y utilizando una serie de herramientas. En el capítulo actual se indican tanto la metodología empleada como las diferentes herramientas utilizadas durante la realización del presente Proyecto Fin de Carrera.

4.1. METODOLOGÍA

El proyecto se ha llevado a cabo siguiendo una metodología iterativa e incremental.

En concreto, se ha empleado el Proceso Unificado de Desarrollo [29], por las razones descritas en el capítulo 3. Así, el desarrollo ha contado con las siguientes características tomadas de este proceso:

- Se ha dirigido por casos de uso, estudiando las funcionalidades requeridas por los diferentes usuarios del sistema.
- Se ha centrado en la arquitectura, dirigiéndose los primeros esfuerzos hacia el desarrollo del "esqueleto" de la aplicación y ampliando su funcionalidad después.
- Ha sido iterativo e incremental, implementándose las diferentes partes del sistema en sucesivas iteraciones. Las iteraciones se describen en el capítulo 6.

En general, se han seguido las cuatro fases del Proceso Unificado y se ha repartido el trabajo en los cinco flujos de trabajo (Fig. 3.10). Los diagramas se han realizado con UML [37].

4.2. HERRAMIENTAS

Se han utilizado diversas herramientas durante el desarrollo de la plataforma. A continuación se enumeran los lenguajes, el software y el hardware empleados.

4.2.1. Lenguajes

Para la elaboración de *QuQuSI* se han utilizado lenguajes de diferente nivel y propósito. A continuación se listan los lenguajes empleados:

- **C++:** Es el lenguaje utilizado en el grueso de la implementación, la parte lógica y gráfica del concurso [38]. Se ha utilizado el estándar C++11, aprovechando varias de sus nuevas características.
- **APL (*Arduino Programming Language*):** Se trata del lenguaje de programación propio de la plataforma Arduino [35]. Es similar a C y se ha usado para programar el Arduino encargado de recibir y enviar las señales de los pulsadores.
- **HTML (*HyperText Markup Language*):** Este lenguaje de marcas se ha empleado en las aplicaciones Web de la plataforma.
- **CSS (*Cascading Style Sheets*):** Es el lenguaje con el que se ha especificado la apariencia de las aplicaciones Web.
- **JavaScript:** Junto con AJAX, se ha utilizado para implementar una consulta periódica (*polling*) del estado del concurso.
- **AJAX (*Asynchronous JavaScript And XML*):** Se trata de una técnica más que de un lenguaje propio. Se ha usado para actualizar la interfaz de las aplicaciones Web sin tener que recargar la página completa.
- **PHP (*PHP Hypertext Pre-processor*):** Es el lenguaje principal empleado en la implementación de las aplicaciones Web.
- **SQL (*Structured Query Language*):** Se ha utilizado para realizar consultas a la base de datos.
- **Latex:** Es un lenguaje para la edición de documentos de alta calidad. Se ha usado para preparar la documentación del proyecto.

4.2.2. Software

En la realización del presente Proyecto Fin de Carrera se ha utilizado software libre de desarrollo, bases de datos, Web, edición gráfica y bibliotecas.

Sistema operativo

Tanto el desarrollo como en el despliegue y la ejecución del proyecto se han llevado a cabo bajo el sistema operativo GNU/Linux.

La distribución utilizada ha sido Debian Wheezy (7.0) en su versión de 64 bits (amd64).

La única excepción es el servidor Web en el que se instalaron la Web de recogida de preguntas y la aplicación Web para el público, el cual utilizaba la distribución Fedora.

Software de desarrollo

El código de la plataforma se ha escrito con diferentes editores libres y se ha compilado y depurado con herramientas de GNU. La lista del software utilizado para el desarrollo es la siguiente:

- **GNU Emacs 23.4.1:** Es un entorno de desarrollo extensible y personalizable. Se ha usado para escribir el código fuente en C++ y los archivos en Latex de la documentación del proyecto.

- **GNU Make 3.81:** Es una herramienta que facilita la compilación de archivos fuente para la obtención de ejecutables. Puede lanzar varios hilos simultáneamente, acelerando la compilación al aprovechar los núcleos disponibles en el equipo. Se ha utilizado para compilar los archivos fuente de la aplicación y la documentación del proyecto.

- **GCC 4.7.2:** Es la colección de compiladores de GNU. Cuenta con soporte para multitud de lenguajes de programación. En el proyecto se ha utilizado un *wrapper* de GCC, colorgcc, que dota de color a la salida de texto del compilador, facilitando la lectura.

- **GDB 7.4.1:** Es el depurador de los sistemas GNU. Se ha empleado para depurar el código de las aplicaciones escritas en C++.

- **Geany 1.22:** Se trata de un editor de texto con las capacidades básicas de un entorno de desarrollo integrado. Cuenta con multitud de *plugins* con los que se puede ampliar su funcionalidad. Se ha utilizado para escribir el código de las aplicaciones Web.

- **Arduino IDE 1.0.1:** Es el entorno de desarrollo integrado de la plataforma Arduino. Permite compilar y cargar el código en la memoria del microcontrolador, además de comunicarse con la placa a través del puerto serie. Con él se ha escrito y probado el código que se ocupa de la comunicación con los pulsadores del concurso.

- **Mercurial 2.2.2:** Es el software de control de versiones utilizado durante el desarrollo del proyecto. Con él se ha mantenido el código en un repositorio en línea y se ha mantenido un diario de desarrollo.

Bases de datos

Las herramientas que se utilizaron para gestionar la base de datos de la plataforma son:

- **MySQL 5.5.31:** Es el sistema gestor de base de datos elegido para almacenar los usuarios, las preguntas, los jugadores y las respuestas. Los motivos de su elección están descritos en el capítulo 3.
- **PHPMyAdmin 3.4.11.1:** Se trata de una interfaz Web que facilita el uso de MySQL. Con él se han gestionado los permisos y las tablas de la base de datos.

Web

En cuanto a las aplicaciones instaladas para el despliegue y la ejecución de las aplicaciones Web, se han usado las siguientes:

- **Apache 2.2.22:** Es el servidor Web elegido para la instalación de las aplicaciones Web. Los motivos por los que se prefirió ante otros servidores Web están descritos en el capítulo 3. Se instaló junto con el módulo *mod_rewrite* requerido por CakePHP.
- **PHP 5.4.4:** Es el interprete de PHP que se instaló junto al servidor Web. Accede a la base de datos y genera los documentos HTML dinámicos de las aplicaciones Web de la plataforma.
- **APC 3.1.13(*Alternative PHP Cache*):** Es una caché tanto para el código interpretado de PHP como para variables de usuario. Se ha utilizado para mejorar el rendimiento de las lecturas en la base de datos y las consultas al servidor del estado del concurso.

Edición gráfica

Para la edición de recursos gráficos y vídeos se han usado las siguientes herramientas:

- ***GIMP* 2.8 (*GNU Image Manipulation Program*):** Es un editor de imágenes libre. Se ha utilizado para crear la interfaz del concurso y para editar algunos de los recursos multimedia.

- ***Blender* 2.49a:** Es un programa para la creación y animación de modelos tridimensionales, aunque también cuenta con otras capacidades como la edición de vídeo. Con él se modeló y exportó el plano sobre el que se reproducen los archivos multimedia del concurso.
- ***LibreOffice Draw* 3.5.4.2:** Es un editor de gráficos vectoriales y diagramas. Se ha usado para crear las figuras de la documentación.
- ***DIA* 0.97.2:** Es el un editor gráfico con soporte para multitud de tipos de diagramas. Con este programa se han elaborado los diagramas UML.
- ***Inkscape* 0.48:** Es un potente editor de gráficos vectoriales. Se ha utilizado para añadir detalles en los diagramas elaborados con el resto de aplicaciones.

Bibliotecas

En el desarrollo de la plataforma se han utilizado diversas bibliotecas y *frameworks*. Estos son:

- ***STL* C++11 (*Standar Template Library*):** Es una biblioteca de C++. Contiene multitud de estructuras de datos y algoritmos. Se ha utilizado el estándar C++11.
- ***OGRE 3D* 1.8.1:** Es el motor gráfico que se encarga del despliegue de los elementos visuales del concurso [14]. Los motivos para su elección se describen en el capítulo 3.
- ***OIS* 1.3.0 (*Object Oriented Input System*):** Es una biblioteca para la gestión de dispositivos de entrada. Se ha utilizado junto con OGRE 3D en las aplicaciones principales del concurso.
- ***GStreamer* 0.10.36:** Es el *framework* elegido para la reproducción de contenidos multimedia [39]. Se eligió por las causas expuestas en el capítulo 3.
- ***CakePHP* 2.2.3:** Es el *framework* con el que se desarrolló la aplicación Web para la introducción de las preguntas. Se ha utilizado para generar parte del código PHP a partir de la descripción de la base de datos.

4.2.3. Hardware

El hardware utilizado durante el presente Proyecto Fin de Carrera ha sido variado y numeroso. En los anexos se incluye el inventario completo con las especificaciones del hardware empleado (Véase Anexo C).

Computadores

Para el desarrollo y el despliegue de *QuQuSI* se usaron varios computadores:

- Cuatro equipos de sobremesa.
- Dos equipos portátiles.
- Un equipo portátil con pantalla táctil.
- Un servidor.

Vídeo

Durante las pruebas de despliegue y la grabación del concurso se utilizaron los siguientes equipos de vídeo:

- Tres cámaras con control remoto.
- Un mando de control remoto.
- Una mesa conmutadora de vídeo.
- Un panel de cuatro pantallas.
- Seis monitores.
- Tres proyectores.
- Un grabador DVD.
- Un grabador HDD.

Sonido

Para la grabación y reproducción del sonido del concurso se usaron los siguientes dispositivos:

- Tres micrófonos con soporte.
- Un micrófono de diadema.
- Una mesa de mezcla de sonido.
- Un amplificador.
- Dos altavoces.

Otros

Además de lo anterior, se utilizó el siguiente material para comunicación, conexión y conversión:

- Un microcontrolador Arduino.
- Tres pulsadores artesanales.
- Dos puntos de acceso Wifi.
- Cuatro conversores de señal de vídeo.

CAPÍTULO 5. ARQUITECTURA

En el capítulo actual se describe la arquitectura de la plataforma *QuQuSI* a través de un enfoque *top-down*, comenzando con una descripción general, continuando con las decisiones de diseño y terminando con los detalles de implementación. La plataforma *QuQuSI* es un sistema distribuido formado por una serie de aplicaciones (Fig. 5.1):

- **Aplicación Web para las preguntas:** Es la aplicación que permite el *workflow* que genera las preguntas del concurso. Permite la gestión de usuarios, roles y preguntas.
- **Servidor *QuQuSI*:** Es la aplicación principal del sistema. Carga las preguntas, mantiene el estado del sistema, recibe señales y eventos y comunica el estado al resto de aplicaciones del sistema.
- **Aplicación para el *chroma*:** Esta aplicación transforma el estado del concurso en una salida visual que puede proyectarse o combinarse con la imagen real obtenida por las cámaras.
- **Aplicación para los marcadores:** Es una pequeña aplicación para los marcadores de los equipos. Muestra el nombre del equipo, su puntuación y cambia de apariencia en función del turno.
- **Aplicación para los pulsadores:** Es la aplicación que hace de puente entre el servidor del concurso y la placa *Arduino*.
- **Aplicación Web de control:** Se trata de una interfaz Web para el control del concurso. Muestra el estado actual del concurso junto con información como el tiempo restante de cada fase o el número de preguntas disponibles.
- **Aplicación Web para el público:** Es la aplicación que permite la participación del público en directo.
- **Reproductor de eventos:** Es un *parser* para el histórico de señales de control y eventos generado durante el concurso. Se puede utilizar en postproducción para simular el desarrollo del concurso junto con la aplicación para el *chroma*.

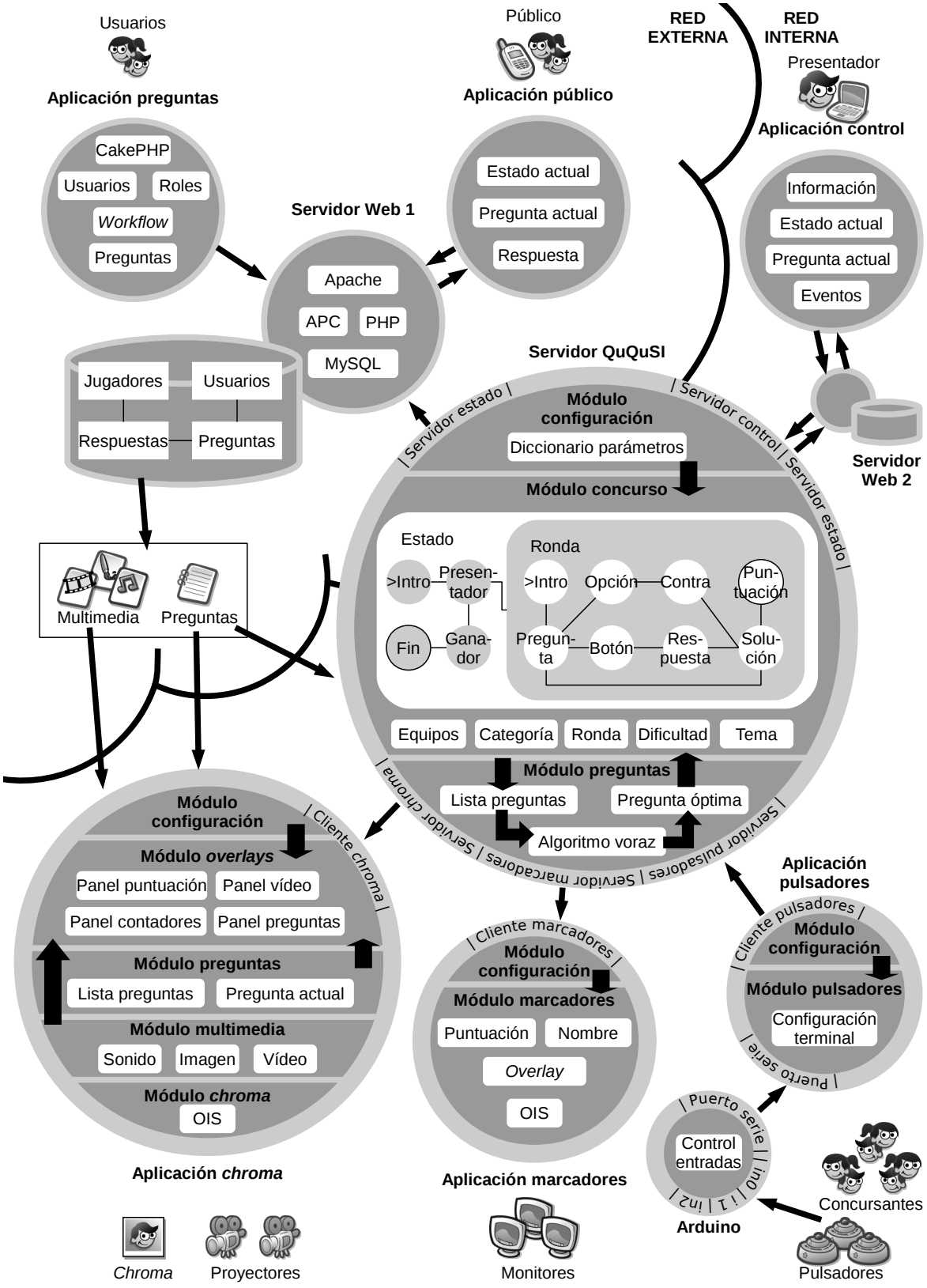

Figura 5.1: Esquema de la plataforma *QuQuSI*.

5.1. DESCRIPCIÓN GENERAL

QuQuSI se ha diseñado como un sistema distribuido con una arquitectura cliente servidor. El sistema está formado por varias aplicaciones cliente que se comunican con el servidor a través de *sockets* (capítulo 3). Esta arquitectura confiere a *QuQuSI* las siguientes características:

- **Coherencia:** El servidor mantiene el estado del concurso, previniendo estados incoherentes. Las comunicaciones son tratadas de forma síncrona, de modo que el estado del sistema es el mismo para todas las aplicaciones cliente.
- **Flexibilidad:** Las aplicaciones que forman *QuQuSI* pueden ejecutarse en redes de diversa topología. Esto permite una gran flexibilidad a la hora de desplegar la plataforma para su explotación.
- **Personalización:** Se pueden incorporar nuevas aplicaciones cliente al sistema realizando unos cambios mínimos en el servidor. Esto dota al sistema de una gran capacidad de personalización.
- **Control distribuido:** La aplicación de control puede ejecutarse desde diferentes computadores. De este modo, se pueden repartir las labores de control entre diferentes personas.
- **Configuración:** Cada aplicación cuenta con una serie de parámetros configurables desde archivo. Así, el sistema no requiere que se compile el código de nuevo para realizar cambios de configuración.

Cada una de las aplicaciones que forman el sistema se ha diseñado siguiendo el paradigma de la programación modular (capítulo 3). Gracias a este diseño, la arquitectura de cada aplicación de *QuQuSI* cuenta con las siguientes características:

- **Reusabilidad:** El código modular es fácilmente reutilizable (Fig. 5.1). Gracias a esto se redujo el esfuerzo durante la implementación. Algunos módulos reutilizados entre las diferentes aplicaciones de *QuQuSI* son el módulo de configuración y la clase *socket*.
- **Mantenibilidad:** Al contar con una mayor cohesión y un menor acoplamiento, el código de *QuQuSI* resulta más fácil de comprender y mantener. Gracias a ello se pudo gestionar la complejidad y el tamaño del código y se acotaron los errores detectados durante las pruebas de despliegue sin mucho esfuerzo.

- **Extensibilidad:** La modularidad del código permite incorporar nuevas clases sin realizar grandes cambios en el resto del código. El diseño cuenta con clases base con las que, por ejemplo, se pueden crear nuevos tipos de pruebas o mecánicas de concurso. También se han implementado plantillas que permiten aplicar patrones de diseño a las clases de una forma sencilla y sistemática. Las plantillas implementadas se corresponden con los patrones *singleton*, *observer* y *state*.

A continuación se describen, desde un mayor nivel de abstracción hasta una descripción más precisa, las diferentes aplicaciones del sistema y los módulos que las forman.

5.2. APLICACIÓN WEB PARA LAS PREGUNTAS

Un concurso televisivo requiere un *workflow* o flujo de trabajo previo en el que se genere una batería de preguntas o pruebas válidas durante la fase de preproducción.

En la elaboración de ésta colección de preguntas deben intervenir personas con diferentes responsabilidades. Unos crearán los contenidos y otros los validarán. Además, debe haber un administrador que gestione los usuarios y sus roles. En cualquier caso es deseable que el número de personas con acceso a todas las preguntas sea mínimo, a fin de evitar filtraciones.

El resultado del flujo de trabajo es una colección de preguntas validadas. La colección deberá poder ser exportada en algún formato que la aplicación del concurso pueda cargar.

Así, el *workflow* debe tener en cuenta una serie de características deseables:

- Debe permitir la introducción, visualización, edición y eliminación de preguntas.
- Las preguntas deben poder agruparse en función de una serie de atributos como su categoría, su tema, su dificultad, etc.
- Las preguntas deben atravesar algún tipo de validación de modo que no lleguen al concurso aquellas incorrectas o que se consideren inadecuadas.
- Debe poder realizar un seguimiento del número de preguntas introducidas y validadas de cada tipo, categoría, tema, etc.
- Debe soportar usuarios con diferentes roles, en función del cual se tengan más o menos privilegios.
- El número de usuarios con acceso a todas las preguntas debe ser mínimo a fin de evitar filtraciones.

- Debe ser accesible y fácil de usar.
- Ha de permitir exportar las preguntas en un formato adecuado.

La aplicación Web de gestión de preguntas se encarga de dar soporte al *workflow* necesario para satisfacer todas estas características.

Esta aplicación, desarrollada con el *framework CakePHP* (capítulo 3), se encarga de la gestión de usuarios y preguntas.

CakePHP cuenta con una arquitectura modelo vista controlador y permite la generación automática de código a partir de una base de datos. Para aprovechar esta funcionalidad, el primer paso es diseñar la base de datos del concurso.

El patrón modelo vista controlador se utiliza para aislar la lógica de la aplicación de su representación gráfica (interfaz de usuario):

- **Vista:** Es la parte de presentación de la aplicación con la que el usuario interactúa.
- **Controlador:** Recibe las acciones realizadas por el usuario en la vista y los traduce al dominio del modelo de la aplicación. Esto se suele llevar a cabo mediante funciones asociadas a cada tipo de evento (*callbacks*) que llaman a los métodos necesarios del modelo.
- **Modelo:** Contiene la lógica de la aplicación y recibe desde el controlador las llamadas a los métodos adecuados en función de las acciones del usuario. Puede comunicarse con la vista, transmitiéndole información de forma asíncrona.

La principal ventaja del patrón modelo vista controlador es que permite utilizar diferentes vistas para representar un mismo modelo sin tener que realizar grandes cambios en este.

La base de datos del concurso almacena la información de usuarios, preguntas, jugadores y respuestas (Fig. 5.2).

La tabla usuarios cuenta con los atributos de identificador, nombre, contraseña y rol. En función del rol del usuario se tienen diferentes permisos en la aplicación:

- **Profesor:** Es el rol por defecto. El usuario puede ver, crear, editar y eliminar sus propias preguntas. Una vez validada una pregunta, el usuario ya no puede modificarla. También puede cambiar su contraseña.
- **Validador:** Es el rol asignado a los encargados de validar las preguntas introducidas. Pueden ver, editar y eliminar cualquier pregunta, además de las propias.

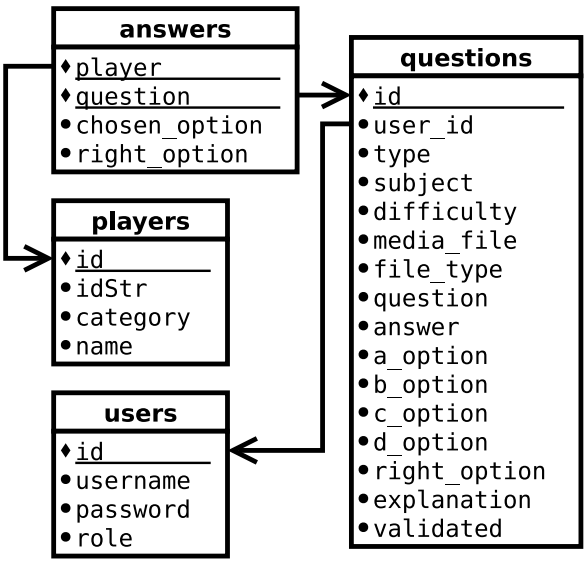

Figura 5.2: Esquema relacional de la base de datos.

- **Administrador:** Es el rol reservado al administrador del sistema. Además de tener todos los permisos sobre las preguntas, puede crear, editar y eliminar usuarios de diferentes roles. También tiene la capacidad de volver a asignar la contraseña por defecto a un usuario.

Respecto a la tabla de preguntas, puesto que el concurso cuenta con diferentes tipos de ellas, una primera aproximación de diseño sugiere el uso de la herencia. Sin embargo, dado que la generación de código automática de los *frameworks* de desarrollo Web no dan un buen soporte a las claves ajenas necesarias para la herencia, se diseñó una única tabla con los campos suficientes para codificar diferentes tipos de pruebas sin quedar muchos vacíos. La tabla de preguntas cuenta con los siguientes campos:

- **Identificador:** Es un campo numérico que identifica de forma univoca a la pregunta.
- **Autor:** Es una clave ajena al identificador del usuario que creó la pregunta.
- **Categoría:** Es la categoría a la que pertenece la pregunta.
- **Tipo:** El tipo de pregunta. En función de este campo se codifica de una forma u otra la pregunta en los campos de la tabla.
- **Tema:** El tema de la pregunta.

- **Dificultad:** La dificultad de la pregunta. En un principio se plantearon cinco niveles de dificultad, pero se redujo a tres después de una reunión con los usuarios de la aplicación.
- **Archivo:** El nombre del archivo multimedia asociado a la pregunta.
- **Tipo de archivo:** El tipo del archivo, pudiendo este ser vídeo, sonido, imagen o ninguno.
- **Pregunta:** El texto de la pregunta, en caso de proceder.
- **Respuesta:** La respuesta a la pregunta, en caso de proceder. En este campo se codifican algunos tipos de pregunta, separando los campos necesarios mediante un carácter auxiliar.
- **Opciones:** Un campo para cada opción de una pregunta tipo test. En el resto de preguntas estos campos están vacíos.
- **Opción correcta:** La opción correcta de una pregunta tipo test. En el resto de preguntas este campo está vacío.
- **Explicación:** La explicación de la solución, de modo que pueda mostrarse al presentador al concurso en caso de necesitarse una aclaración.
- **Validada:** Campo que indica si la pregunta ha sido validada o está pendiente de validación.

La codificación de ciertos tipos de preguntas se realiza mediante el uso de un carácter auxiliar que divide un campo de texto en varios.

Las tablas de jugadores y respuestas son ajenas a la gestión de usuarios y preguntas, por lo que serán descritas en la aplicación Web para el público, en la sección 5.8.

Una vez creadas las tablas necesarias, se generó el código base de la aplicación, el cual fue modificado en función de los requisitos de la aplicación.

La creación, visualización, modificación y borrado de preguntas se consigue con el código generado. Sin embargo, al tener diferentes tipos de preguntas, fue necesario reescribir parte del código del controlador y de las vistas. Así, se creo una nueva acción en el controlador, denominada `select` y una vista asociada. Esta vista aparece cuando se quiere crear una nueva pregunta y permite elegir su categoría y su tipo. El tipo se pasa como variable a la vista de creación de la pregunta, mostrando así el formulario correspondiente.

Las preguntas se pueden ordenar y agrupar gracias al método `paginate` proporcionado por el *framework*.

La validación de las preguntas es llevada a cabo por los usuarios con rol validador. Se guarda en el campo correspondiente si la pregunta está o no validada

El seguimiento del número de preguntas introducidas y validadas se realiza en la acción `select` mediante una tabla. Se recuperan desde el modelo el número de preguntas validadas y sin validar en función de su tipo y su categoría mediante el uso de las funciones de acceso proporcionadas por el *framework*

El soporte de usuarios con diferentes roles se logra mediante el uso del módulo `Auth` proporcionado por el *framework*. Este módulo proporciona los mecanismos de autenticación de usuarios (*login*), cifrado de contraseñas y control de acceso.

Las preguntas validadas se exportan en un archivo de texto en un directorio (media) junto con los archivos multimedia cargados por los usuarios, de modo que todo el contenido necesario para llevar a cabo el concurso se encuentra en el mismo lugar.

En las siguientes secciones se describe con mayor detalle las clases pertenecientes a cada capa del patrón modelo vista controlador.

5.2.1. Modelo

En la capa de modelo se cuenta con una clase por cada relación de la base de datos. Gracias a *CakePHP*, estas clases cuentan con las operaciones CRUD. Las clases que se han implementado en esta capa son:

- **`User`:** Es la clase que representa a un usuario del sistema. Establece una serie de restricciones a los valores que pueden tomar los atributos de un usuario mediante las reglas de validación proporcionadas por *CakePHP*.

- **`Question`:** Es la clase que representa una pregunta. Cuenta con una serie de constantes (categorías, temas, dificultades, etc.) y unas normas de validación.

5.2.2. Controlador

En la capa de control se asocian los eventos ocurridos en la vista con las funciones de la capa de modelo. Las clases que se implementaron en esta capa son:

- **`UsersController`:** Es el controlador para los usuarios. Implementa el mecanismo de *login* y los permisos de acceso mediante el módulo *Auth* proporcionado por

CakePHP. Cuenta con métodos que traducen las acciones del usuario en la vista a operaciones CRUD en el modelo.

- **`QuestionsController`:** Es el controlador para las preguntas. Proporciona funciones para la gestión de las preguntas en el modelo mediante operaciones CRUD, gestiona la subida y bajada de archivos, la exportación de las preguntas y la codificación y decodificación de los campos de las preguntas.

5.2.3. Vista

La vista provee de una interfaz de usuario a la aplicación, en la que se muestran los datos del modelo y se permite realizar acciones que los afectan. En esta capa se implementaron las siguientes vistas:

- **`Questions`:** Para las preguntas se cuenta con las siguientes vistas:
  - **`add`:** Es la vista para añadir una nueva pregunta. Se formatea un cuestionario en función del tipo de pregunta que se eligió en la vista `select`.
  - **`edit`:** Se trata de la vista para editar una pregunta existente. Los campos del formulario de edición se formatean en función del tipo de pregunta.
  - **`export`:** Es la vista para exportar las preguntas validadas desde la base de datos a un archivo de texto.
  - **`index`:** Es la vista en la que se listan las preguntas existentes. En función del rol del usuario se visualizaran todas las preguntas (validador y administrador) o tan solo las propias (profesor).
  - **`select`:** Es la vista previa a la creación de una nueva pregunta, en la que se determina la categoría y el tipo de pregunta. Muestra además una tabla con información sobre el número de preguntas validadas y sin validar agrupadas por tema y tipo.
  - **`view`:** Es la vista empleada para mostrar una pregunta.
- **`Users`:** Para los usuarios se tienen las siguientes vistas:
  - **`add`:** Es la vista utilizada para insertar un nuevo usuario en la base de datos.
  - **`change_password`:** Es una vista que permite al usuario cambiar su contraseña.

- **`edit`:** Se trata de la vista para la edición de un usuario existente.
- **`index`:** Es la vista en la que se muestra el listado de usuarios.
- **`login`:** Es la vista en la que el usuario introduce su nombre y contraseña para acceder al sistema.
- **`view`:** Se trata de la vista que permite mostrar la información de un usuario.

5.3. SERVIDOR *QUQUSI*

Un sistema distribuido como *QuQuSI* necesita mecanismos de comunicación y sincronización de modo que las partes que lo forman reciban la información pertinente en el momento correcto. La importancia de esta condición es aún mayor por tratarse de un concurso en directo. La comunicación ha de ser rápida y confiable, a fin de evitar pausas y perdidas de información respectivamente.

El sistema también necesita gestionar los recursos disponibles (preguntas y archivos multimedia). Debe permitir importar las preguntas y llevar una contabilidad de las mismas durante el concurso, de modo que no se repitan y no se acaben sin previo aviso.

También requiere mantener el estado del concurso de forma que este sea visto por todas las partes del sistema del mismo modo, manteniendo la coherencia. El sistema debe conocer la información del estado que es relevante a cada parte del mismo, de modo que se transmita la menor cantidad de datos posible tras cada actualización.

Otra funcionalidad relacionada con el concurso es la gestión dinámica del nivel de dificultad y la selección de la pregunta adecuada en función del estado del concurso.

Surgen así las siguientes características y funcionalidades deseables:

- Debe recibir señales de control y eventos de forma síncrona, previniendo así que se llegue a estados incoherentes o se sobrescriban los cambios de estado.
- Comunicar al resto de partes del sistema la información que requieran tras producirse un cambio en el estado del concurso.
- Debe mantener la coherencia, de modo que el estado del concurso percibido por cada parte del sistema sea el mismo.
- La comunicación debe ser rápida, enviándose la información estrictamente necesaria en cada mensaje.
- Prevenir la pérdida de mensajes, utilizando mecanismos con detección de errores y pérdida de datos.

- Gestionar la importación y el uso de las preguntas del concurso, evitando que se repitan e informando en caso de que queden pocas disponibles.
- Seleccionar de forma dinámica la pregunta más adecuada en función del estado del concurso.
- Gestionar el nivel de dificultad del concurso teniendo en cuenta los fallos y aciertos de los concursantes.
- Debe contar con capacidad de configuración, de forma que no sea necesario compilar el código cada vez que se quiera cambiar algún parámetro.

Para satisfacer estas características se diseño el sistema con una arquitectura cliente servidor.

El servidor *QuQuSI*, desarrollado en C++, es la parte más importante del sistema ya que se encarga de tareas críticas como mantener el estado del concurso y la sincronización del resto de aplicaciones.

En su diseño se tuvieron en cuenta los siguientes patrones:

- ***Observer:*** Este patrón se ha utilizado para dos tareas. La primera es la lectura síncrona de los *sockets* mediante la función `select`. La segunda es para notificar a los clientes de los cambios en el estado del concurso.
- ***State:*** Se presenta como una solución adecuada para modelar el comportamiento del concurso. El concurso en sí es una máquina de estados que recibe señales de control y eventos.
- ***Singleton:*** El patrón es utilizado en aquellas clases en las que la existencia de varias instancias podría provocar problemas de coherencia.

Se siguió el paradigma de la programación modular para su diseño. La aplicación cuenta con los siguientes módulos:

- **Gestor de configuración:** Es el módulo encargado de gestionar los parámetros de configuración de la aplicación. Estos parámetros son leídos desde un archivo, en donde cada línea contiene un par clave valor. Las claves leídas son almacenadas en un diccionario junto con su valor. El módulo ofrece mecanismos para recuperar, modificar y eliminar los parámetros.

- **Gestor de comunicaciones:** Es el módulo encargado de gestionar la comunicación con el resto de aplicaciones del sistema. Cuenta con varios servidores a modo de *plugin*. Cada servidor se encarga de la comunicación con un tipo de aplicación cliente. Los diferentes servidores y clientes se suscriben al gestor de concurso (patrón *observer*), siendo notificados de los cambios. También conocen la información estrictamente necesaria requerida por cada aplicación, de modo que no se transmiten datos irrelevantes.

- **Gestor de preguntas:** Este módulo se encarga de importar el archivo de texto que contiene las preguntas exportadas desde la aplicación Web. Las preguntas leídas se almacenan en una lista. El módulo lleva la contabilidad de las preguntas restantes de cada categoría, tipo, tema, etc. Además, se encarga de la selección dinámica de la pregunta más adecuada en cada momento mediante un algoritmo voraz que evalúa la lista de preguntas en base a los parámetros del concurso.

- **Gestor de concurso:** Este módulo gestiona la lógica del concurso. Incluye la máquina de estados que controla el desarrollo del concurso y la información sobre los equipos, las puntuaciones, las fases y las rondas. Gestiona la dificultad de las preguntas de forma dinámica, en función de los eventos de fallo y acierto recibidos. A partir de sus variables se crea la petición para la selección dinámica de preguntas.También guarda un historial de las señales de control y los eventos recibidos en un archivo para su posterior reproducción con el reproductor de historial descrito en la sección 5.9.

Gracias al uso del patrón *observer* y a la función `select` en el módulo de comunicaciones se consigue que la comunicación se realice de forma síncrona, evitando que señales simultáneas provoquen fallos en el desarrollo del concurso.

Con el patrón *observer* implementado entre el módulo de comunicación y el de concurso (*subject*) también se logra que las aplicaciones clientes se mantengan actualizadas, de modo que el estado del concurso se conozca en todo el sistema. Además, el servidor conoce la información requerida por cada cliente, de modo que se envía un mensaje con la información estrictamente necesaria.

La coherencia del sistema se mantiene gracias al uso de los patrones *state* y *singleton* en el módulo de concurso. Existe una única instancia tanto de la máquina de estados del concurso como de cada uno de los estados particulares.

Los problemas de pérdida de mensajes y detección de errores son abstraídos gracias al uso de *sockets* TCP en el módulo de comunicación.

El nivel de dificultad del concurso se establece en la máquina de estados del módulo de comunicación en función de las señales recibidas (respuesta correcta o incorrecta).

La selección dinámica de preguntas se realiza mediante un algoritmo voraz (capítulo 3) implementado en el gestor de preguntas, el cual tiene en cuenta una serie de variables (categoría actual, tema, dificultad, tipo de ronda, etc.) que obtiene del módulo de concurso.

En las siguientes secciones se describe con más detalle la implementación de cada módulo de la aplicación.

5.3.1. Módulo de configuración

Este módulo consta de una única clase (`QConfig`) la cual se encarga de la gestión de los parámetros de configuración (Fig. 5.3).

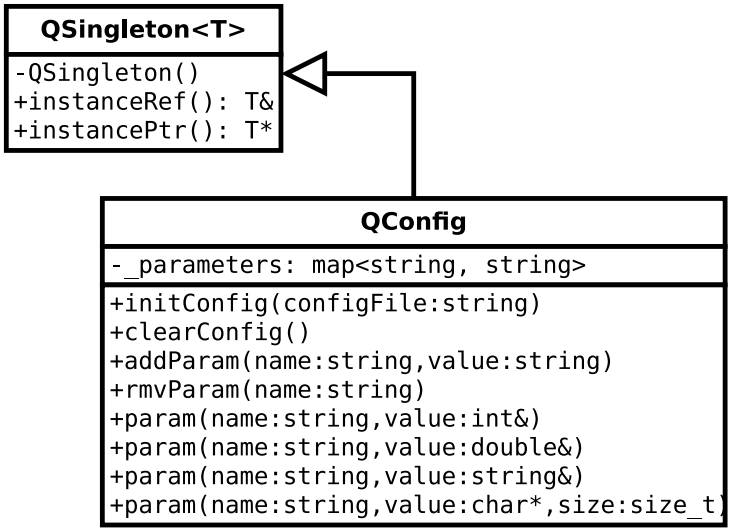

Figura 5.3: Diagrama de clases del módulo de configuración.

La clase `QConfig` hereda de la clase plantilla `QSingleton`, la cual permite el uso directo de este patrón. El patrón está implementado con instancias estáticas y la declaración de constructores privados y es totalmente reutilizable (Véase Anexo H). Gracias al uso de este patrón, cualquier clase puede acceder al gestor y recuperar los parámetros de forma coherente (no existirán varias instancias del gestor con diferentes valores para un mismo parámetro).

Los parámetros son almacenados en un mapa indexado por su nombre (el atributo miembro `parameters`). El valor se almacena en una cadena de texto que posteriormente es convertida al tipo deseado. Esta solución permite serializar cualquier tipo de valor en una cadena de texto que luego sea procesada y convertida al tipo deseado.

La función `initConfig` toma como entrada la ruta al archivo de configuración. Tras abrir el archivo, lo procesa línea a línea, notificando de nombres repetidos o valores mal

formateados. La función `clearConfig` permite vaciar el mapa de parámetros, reiniciando el módulo a su estado inicial.

Las funciones `addParam` y `rmvParam` permiten añadir y quitar parámetros del diccionario. Al tener una visibilidad pública se permite introducir parámetros desde otras partes del código además de desde archivo. De este modo se pueden fijar parámetros en tiempo de ejecución por estar ausente en el archivo de configuración o por tener un valor inadecuado.

La recuperación de los valores se logra mediante la sobrecarga de funciones. Las diferentes versiones de la función `param` tratan de convertir la cadena de texto en la que se almacena el valor de un parámetro al tipo de variable pasado como argumento. Comprueba que el parámetro existe y notifica en caso de no poder convertir en valor al tipo deseado. Esta solución ofrece una interfaz minimalista y sencilla que permite usar el módulo con diferentes tipos de variable (entero, punto flotante y dos tipos de cadena de caracteres).

5.3.2. Módulo de comunicación

El módulo se encarga de la comunicación síncrona a través de *sockets* TCP. El diseño se realizó siguiendo una filosofía *GNU/Linux*; partes pequeñas que realizan tareas concretas. Por ello, en lugar de un *socket* con un protocolo complejo, se prefirió utilizar varios *sockets* con protocolos sencillos (Fig. 5.4).

En general, su funcionamiento se basa en la función `select` y el patrón *observer*.

La función `select` sirve para esperar cambios en descriptores de archivo de cualquier tipo, como la entrada estándar o los *sockets*. Antes de llamar a `select` se han de incluir los descriptores de archivo en alguno de los tres conjuntos disponibles (lectura, escritura y excepción); `select` tratará un tipo de cambio u otro dependiendo del conjunto. Otra característica de `select` puede bloquear la ejecución durante un tiempo determinado mientras espera a que ocurra algún cambio en los descriptores de archivo. Este tiempo puede configurarse como cero, otro valor o infinito.

Para dar soporte a la función `select` se implementan la clases `QSelectMgr` y `QSelect`. `QSelect` es una clase abstracta que proporciona una interfaz para que una clase pueda suscribirse a `QSelectMgr`. La clase que hereda de `QSelect` ha de implementar la función `qSelFd`, la cual devuelve un descriptor de archivo. En función de los cambios que interese controlar, la clase implementará una o varias de las funciones `qSelRead`, `qSelWrite` y `qSelExcept` y se suscribirá en `QSelectMgr` a través de las funciones `addReadSel`, `addWriteSel` y `addExceptSel`. Una vez suscritas las clases en los conjuntos deseados, se utiliza `qSelect`. Esta función se encarga de incluir los descriptores de fichero de las clases suscritas en los conjuntos correspondientes y llama a `select`. En caso de producirse

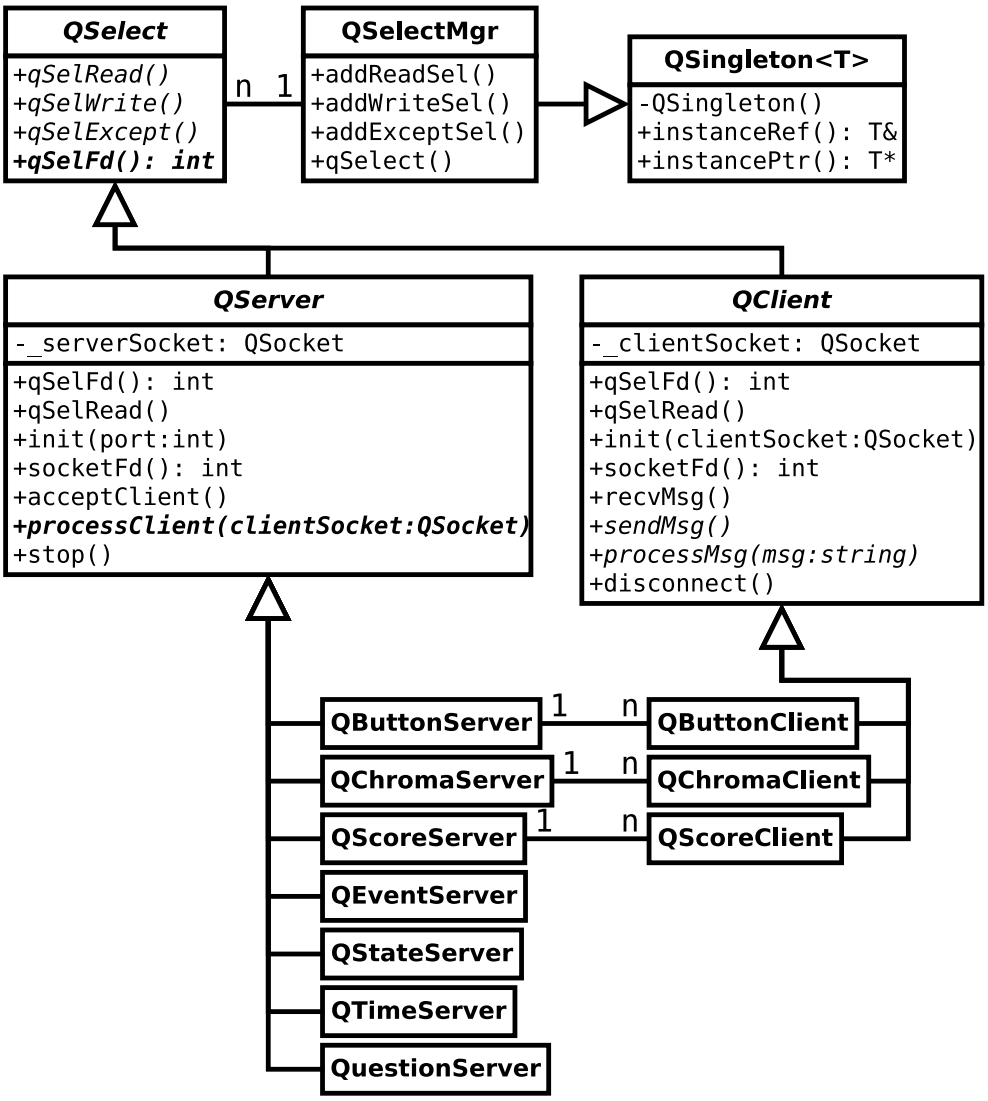

Figura 5.4: Diagrama de clases del módulo de comunicación del servidor *QuQuSI*.

algún cambio en alguno de los descriptores de archivo, llama a la función correspondiente de la clase que implementa `QSelect`, la cual lleva a cabo una tarea específica (recibir mensajes, aceptar clientes, enviar datos, etc.). La clase `QSelectMgr` permite fijar el tiempo de espera de `select` en segundos o milisegundos. Esta solución permite que se reutilice el código para cualquier aplicación que haga uso de la función `select`, ya sea con *sockets* u otros descriptores de archivo.

A partir de estas clases, se implementan las clases que permiten que el servidor *QuQuSI* se comunique con el resto de aplicaciones del sistema.

Las clases `QServer` y `QClient` implementan la interfaz `QSelect`. El descriptor de archivo que utilizan para suscribirse a `select` es el proporcionado desde la clase `QSocket`, que es un *wrapper* en C++ para las funciones C relacionadas con los *sockets*. `QSocket` es totalmente reutilizable y proporciona todas las funciones necesarias para enviar y recibir datos a través de *sockets* TCP, tanto en aplicaciones servidor como cliente.

`QServer` es una clase abstracta para la implementación de partes del servidor. Cada servidor se encarga de la comunicación con un tipo de aplicación o de una tarea específica. La función `init` crea el *socket*, lo enlaza y queda a la espera de conexiones de clientes. Cuando la recibe, desde `QSelectMgr` se llama a la función `acceptClient`, la cual crea el `QSocket` de servicio de la aplicación cliente. Este `QSocket` se pasa como argumento a la función virtual pura `processClient`, la cual se encarga o bien de crear un objeto `QClient` o de enviar directamente la información necesaria. Por último, la función `stop` termina la conexión con los clientes, elimina los objetos `QClient` y cierra el *socket* servidor. Este planteamiento permite que se pueda ampliar la funcionalidad del servidor sin tener que modificar el código existente; tan solo se debe implementar la nueva parte del servidor e inicializarlo desde el código llamando a `init`.

`QClient` representa a aquellas aplicaciones con *sockets* persistentes (al contrario de los *sockets* PHP, los cuales se cierran una vez se ejecuta el *script*). La función `init` se limita a copiar el *socket* que se pasa como argumento en la variable miembro de la clase. La función `recvMsg` se encarga de recibir los mensajes a través del *socket* y pasarlos a la función `processMsg`, la cual realiza el tratamiento adecuado. La función `sendMsg` puede utilizarse para enviar un mensaje a la aplicación cliente. Gracias a esta clase, se logra un mecanismo genérico que permite gestionar la comunicación con aquellos clientes que establecen conexiones persistente con el servidor.

Ambas clases implementan también el patrón *observer* para recibir notificaciones sobre el estado del concurso. Se omite este detalle ya que se describe en profundidad en la siguiente sección.

A partir de estas dos clases se implementan las partes del servidor *QuQuSI*. La función de cada parte es:

- **`QEventServer`:** El servidor recibe las señales de control y los eventos a través de esta clase. Cuando la clase recibe una conexión, lee un mensaje de un tamaño determinado y lo envía al módulo de concurso. La relación con el módulo de concurso está descrita en la sección 5.3.3 (Fig. 5.5).
- **`QStateServer`:** Al recibir una conexión, envía a través del *socket* de servicio de la aplicación cliente el nombre del estado en el que se encuentra el concurso y termina la comunicación.
- **`QTimeServer`:** Cuando recibe una conexión, genera un fragmento de código HTML en el que se incluye información sobre el estado del concurso y lo envía a través del *socket* de servicio.
- **`QuestionServer`:** Responde a las conexiones enviando el identificador de la pregunta actual. En caso de no haber pregunta, envía un valor negativo.
- **`QButtonServer`:** Acepta la conexión de la aplicación para los pulsadores, creando un objeto de la clase `QButtonClient` y destruyéndolo cuando se cierra la conexión.
- **`QButtonClient`:** Recibe el identificador del pulsador presionado desde la aplicación para los pulsadores y lo envía al módulo de concurso.
- **`QChromaServer`:** Es la parte del servidor que gestiona la conexión de la aplicación para el *chroma*. Crea objetos de la clase `QChromaClient`, encargándose también de su eliminación.
- **`QChromaClient`:** Se encarga de enviar los eventos del concurso a las aplicaciones para el *chroma* conectadas.
- **`QScoreServer`:** Es la parte responsable de la conexión de la aplicación para los marcadores. Crea objetos de la clase `QScoreClient` y es responsable de su existencia.

5.3.3. Módulo de concurso

El módulo de concurso es el que se encarga de la lógica del juego y el estado del concurso. Recibe los eventos que modifican su estado a través del módulo de comunicación (Fig. 5.5).

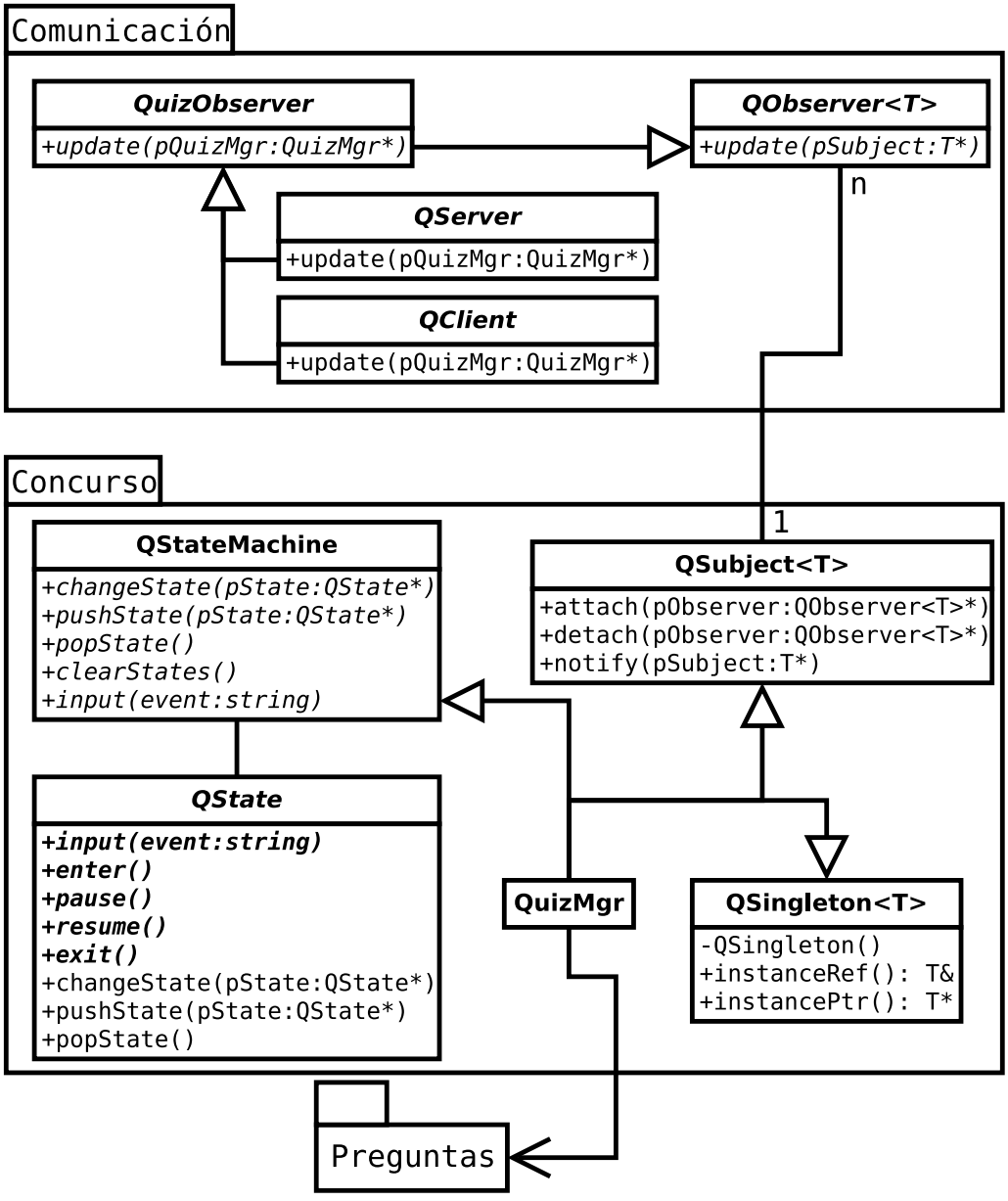

Figura 5.5: Diagrama de clases de la interacción entre el módulo de comunicación y el módulo de concurso.

La comunicación con las partes del módulo de comunicación se lleva a cabo mediante el patrón *observer*. Para la aplicación directa de este patrón existen las clases plantilla `QObserver` y `QSubject`, totalmente reutilizables y de aplicación simple (Véase Anexo H). Gracias a esta solución se logra que las diferentes partes del módulo de comunicación conozcan el estado del concurso y puedan acceder a sus funciones.

`QObserver` es una interfaz que representa a los objetos (*observers*) que se interesan por el estado de otro (*subject*). El observado puede notificar en cualquier momento a los observadores mediante la función `update`. El motivo por el que esta función tiene como argumento al observado es para que un mismo observador pueda suscribirse a diferentes sujetos, cada cual llama a su versión de la función `update`. Así, un objeto puede observar a diferentes tipos de sujetos y un sujeto puede ser observado por diferentes tipos de objetos.

Por su parte, `QSubject` cuenta con una lista de observadores a los que notificar. Se añaden y quitan observadores mediante las funciones `attach` y `detach`, respectivamente. Con la función `notify` el sujeto recorre la lista de observadores llamando a su función `update`.

La clase `QuizObserver` es una especialización de la clase QObserver para crear observadores de la clase `QuizMgr`. Esta interfaz es implementada por las clases `QServer` y `QClient` descritas en la sección 5.3.2. De este modo, las aplicaciones conectadas al servidor son notificadas de los cambios en el concurso.

La clase `QuizMgr` es el núcleo del módulo de concurso. Implementa, además de los patrones *observer* y *singleton*, el patrón *state*. La implementación del patrón *state* se lleva a cabo en las clases `QStateMachine` y `QState`, las cuales proporcionan una interfaz sencilla y totalmente reutilizable para su aplicación.

La clase `QStateMachine` mantiene una pila de estados y reenvía las llamadas a funciones al estado que se encuentra en la cima. Al cambiar de estado llama en el orden adecuado a las funciones que gestionan dicho cambio. La clase `QState` proporciona una interfaz para la creación de estados específicos. Sus funciones son llamadas desde la clase `QStateMachine` cuando se cambia de estado. Esto permite obtener un código más claro y mantenible, ya que no se acumula todo el código de la gestión de estados en una misma clase.

La clase `QuizState` supone una especialización de la clase `QState` para su uso con `QuizMgr`. Cada estado del concurso hereda de esta clase, a través de la cual conocen a la clase `QuizMgr`. Cuando se recibe un evento en `QuizMgr` a través de la función `input` heredada de `QStateMachine`, se pasa el evento a la función `input` del estado activo, el cual se encarga de gestionar las transiciones (Fig. 5.6).

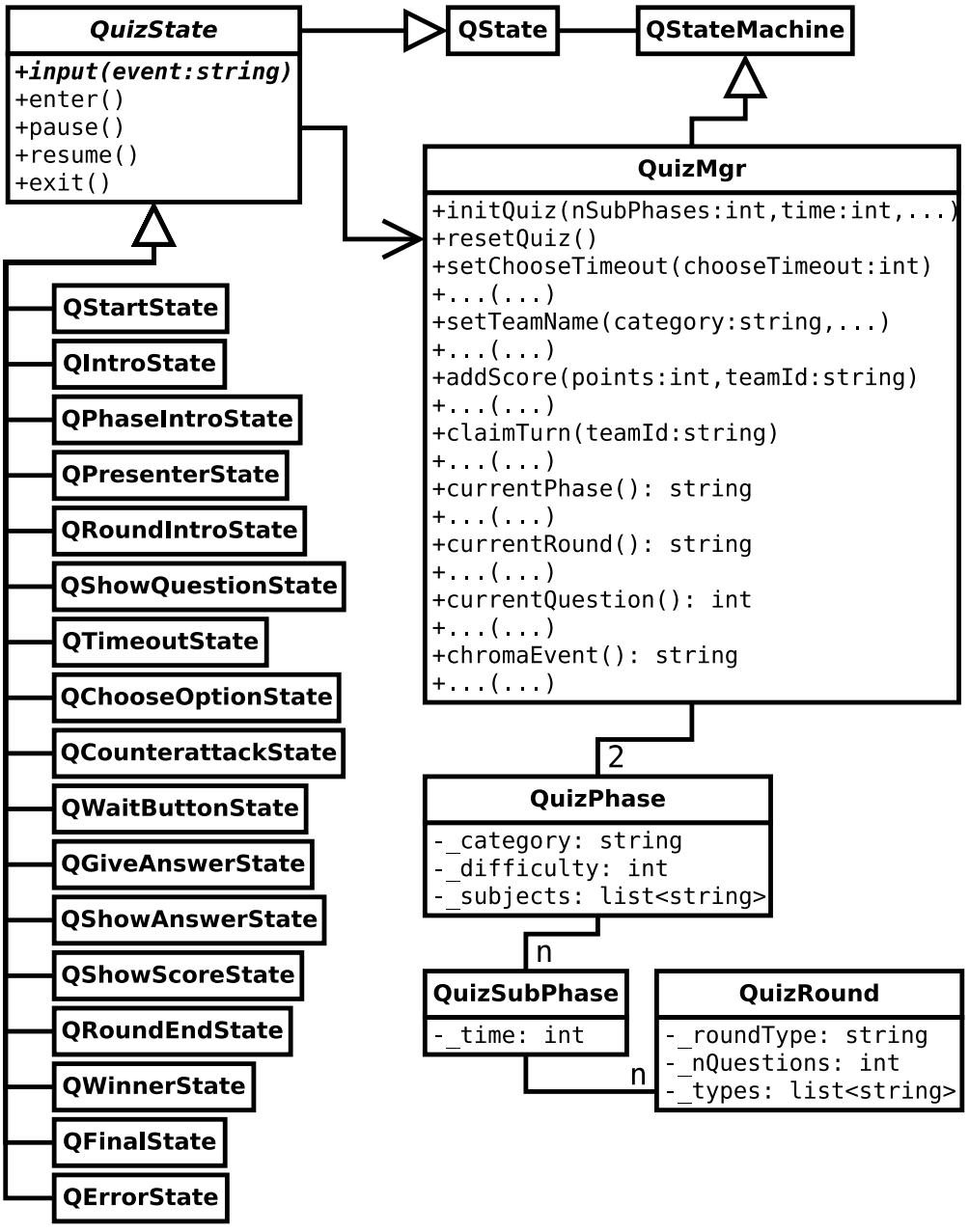

Figura 5.6: Diagrama de clases del módulo de concurso.

A continuación se describe brevemente el propósito de cada estado:

- **`QStartState`:** Es el estado inicial del concurso al arrancar la plataforma; suena la música de fondo y los marcadores y las aplicaciones *chroma* no muestran nada.
- **`QIntroState`:** Es el estado durante el cual se reproduce el vídeo de inicio del concurso.
- **`QPhaseIntroState`:** Se trata del estado inicial de cada categoría, en el cual se presenta a los concursantes.
- **`QPresenterState`:** Es el estado de descanso entre ronda y ronda.
- **`QRoundIntroState`:** En este estado se reproduce la introducción correspondiente al tipo de ronda actual.
- **`QShowQuestionState`:** Es el estado en el que la pregunta se muestra al presentador pero no a los concursantes, de forma que la pueda leer.
- **`QChooseOptionState`:** En este estado se muestra una pregunta de tipo test y se produce la cuenta atrás antes de la cual los concursantes deben proporcionar una respuesta.
- **`QTimeoutState`:** Es el estado al que se avanza cuando termina la cuenta atrás para que los concursantes elijan una opción.
- **`QCounterattackState`:** Tras elegir una opción a una pregunta tipo test, se pasa a este estado, en el que los contrincantes tienen la posibilidad de reclamar el turno.
- **`QWaitButtonState`:** En este estado se muestra una pregunta de tipo diferente a test y se espera a que los concursantes activen el pulsador.
- **`QGiveAnswerState`:** Cuando un equipo a accionado su pulsador, se pasa a este estado en el que deben proporcionar una respuesta.
- **`QShowAnswerState`:** Cuando se acierta una pregunta, sea del tipo que sea, se pasa a este estado en el que se muestra la solución.
- **`QShowScoreState`:** Al acabar una ronda se pasa a este estado en el que se anuncia su final.

- **`QRoundEndState`:** Es un estado de transición durante el cual se muestra la puntuación de los equipos.
- **`QWinnerState`:** Al final de cada categoría se llega a este estado en el que se muestra la puntuación final de cada equipo y se declaran los ganadores.
- **`QFinalState`:** Al acabar todas las categorías se pasa a este estado en el que se anuncian los ganadores del público.
- **`QErrorState`:** A este estado se llega en caso de que no sea posible terminar una ronda por no quedar más preguntas adecuadas.

Además de gestionar las transiciones entre estados, estos se encargan de llamar a algunas de las funciones que modifican el estado del concurso en la clase `QuizMgr`.

`QuizMgr` lleva la lógica del concurso, incluyendo equipos, puntuaciones, categorías, rondas, tipo de preguntas, etc. El concurso se configura a través de los argumentos pasados a la función `initQuiz`. Estos parámetros son cargados previamente por el módulo de configuración. La función `resetQuiz` reinicia el estado del concurso al cambiar de categoría. El resto de funciones de la clase están relacionadas con la gestión del tiempo, los equipos, las puntuaciones, el turno, la categoría, las rondas, las preguntas y la notificación del resto de aplicaciones del sistema a través del patrón *observer*. Cabe mencionar la función `popQuestion`, la cual genera una petición para el módulo de preguntas a partir del estado del concurso (Listado 5.1). La forma en la que el módulo de preguntas trata esta petición se describe en la sección 5.3.4.

Las clases `QuizPhase`, `QuizSubPhase` y `QuizRound` facilitan la gestión del desarrollo del concurso. Gracias a ellas, se conoce la categoría, la dificultad, los temas, el tiempo, el tipo de ronda, el número de preguntas restantes de la ronda y los tipos de preguntas de la ronda. Algunos de estos valores se almacenan en colas circulares. Gracias a esta estructura de datos se recuperan cíclicamente valores como el tipo o el tema de la pregunta, beneficiando así la aparición de preguntas variadas durante el concurso.

5.3.4. Módulo de preguntas

El módulo de preguntas carga las preguntas desde un archivo de texto a una lista, lleva a cabo la selección dinámica en función de una petición con el estado del concurso y gestiona la creación y destrucción de los objetos implicados (Fig. 5.7).

La principal clase del módulo es `QuestionMgr`, la cual implementa el patrón *singleton* y gestiona la carga, selección y eliminación de las preguntas. La función `loadQuestions`

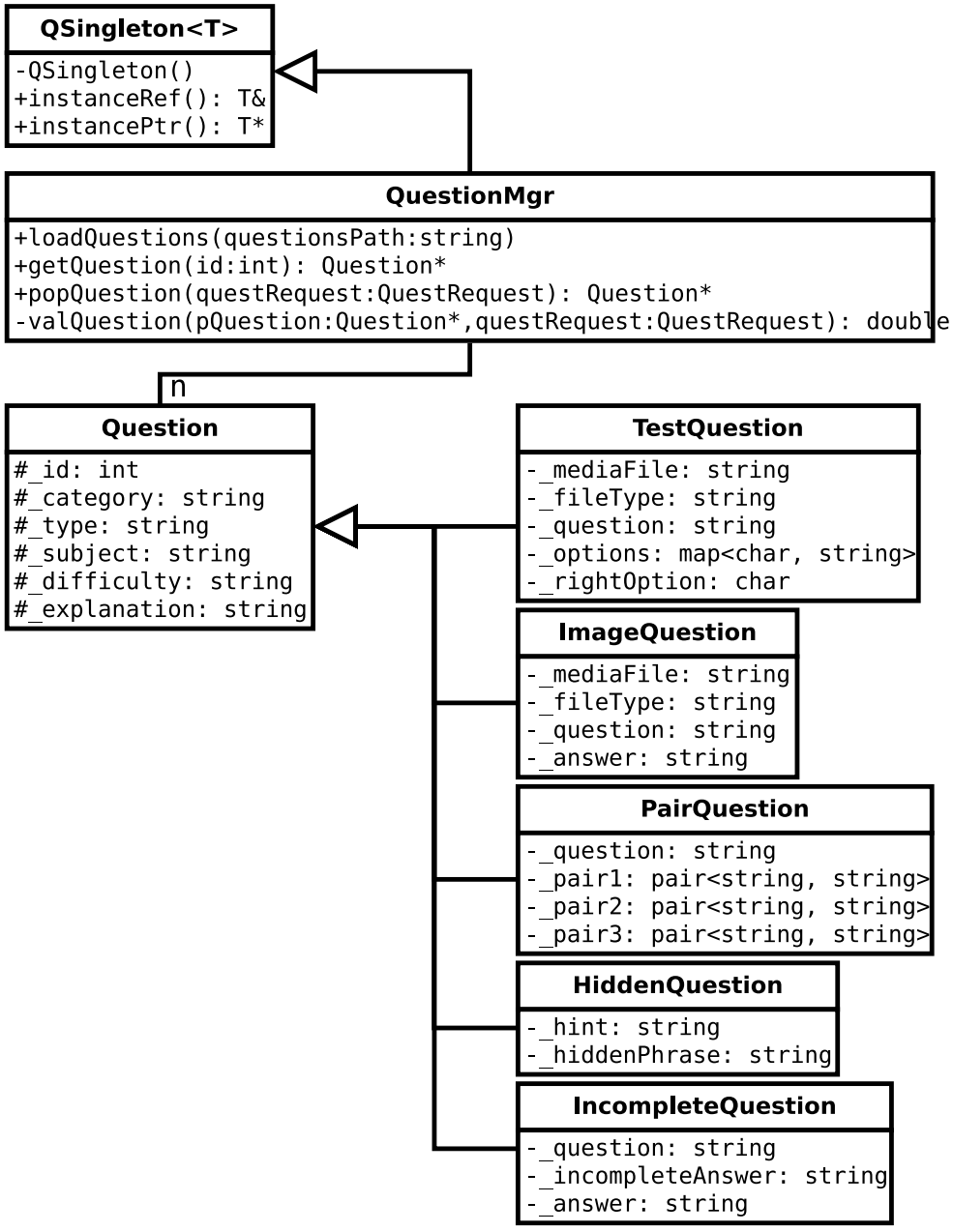

Figura 5.7: Diagrama de clases del módulo de preguntas.

```
// Obtiene una pregunta
bool
QuizMgr::popQuestion() {
    QuestRequest questRequest;
    double val;
    // Ronda
    questRequest.round = currentRound();
    // Categoria
    questRequest.category = currentPhase();
    // Tema
    questRequest.subject = popSubject();
    // Tipo
    questRequest.type = popType();
    // Dificultad
    questRequest.difficulty = _phases.front().difficulty();
    // Obtener pregunta
    _pQuestion = _pQuestionMgr->popQuestion(questRequest, val);
    // Ninguna pregunta satisface la peticion
    if (_pQuestion == nullptr)
        return false;
    return true;
}
```

Listado 5.1: Código fuente de la función `popQuestion`.

recibe como argumento la ruta del archivo que contiene las preguntas. Tras abrir el archivo, lo recorre línea a línea, creando los objetos que representan a cada tipo de pregunta e incluyéndolo en una lista. La función `getQuestion` permite recuperar el puntero a una pregunta directamente a través de su identificador. Por su parte, la función `popQuestion` recibe una petición con información sobre el estado del concurso y llama a la función `valQuestion` para obtener la pregunta óptima en función de tal estado. La función `popQuestion` implementa un algoritmo voraz que recorre la lista de preguntas y devuelve la más adecuada en base a la petición recibida; en caso de no existir una pregunta óptima, devuelve la que más se aproxime (Listado 5.2).

La clase `Question` es la base de todos los tipos de preguntas. Incluye como miembros aquellos comunes a todos ellos; el identificador, la categoría, el tipo, el tema, la dificultad y la explicación. El resto de clases son los objetos que representan a los diferentes tipos de preguntas (Véase Anexo B):

- **`TestQuestion`:** Representa la pregunta de tipo test multimedia. Cuenta con el nombre del archivo multimedia y su tipo, la pregunta, las opciones y la opción correcta.

- **`ImageQuestion`:** Representa la pregunta de tipo adivinar imagen. Incluye como miembros el nombre del archivo de imagen asociado, su tipo (por compatibilidad), la pregunta y la respuesta.

```
double
QuestionMgr::valQuestion(Question* pQuestion,
                         QuestRequest questRequest) {
    double cat, sub, typ, dif;
    // Evaluar categoria
    // Si el tema es cultura general se ignora la categoria
    if (questRequest.subject == SUBJ_MISC) { cat = 1; }
    // Si no, la categoria debe coincidir con la requerida
    else {
        if (pQuestion->category() == questRequest.category) cat = 1;
        else cat = 0;
    }
    // Evaluar tema
    // Si el tema es el requerido se valora con uno
    if (pQuestion->subject() == questRequest.subject) sub = 1;
    // Tema no es el requerido ni es cultura general se valora con medio
    else if (pQuestion->subject() != SUBJ_MISC) sub = 0.5;
    // Se penaliza cultura general frente a otros temas
    else sub = 0.25;
    // Evaluar tipo
    // Ronda test solo valen tipo test
    if (questRequest.round == TEST_ROUND) {
        if (pQuestion->type() == TYPE_TEST) typ = 1;
        else typ = 0;
    }
    // Ronda misc no valen tipo test
    else if (questRequest.round == MISC_ROUND) {
        if (pQuestion->type() == TYPE_TEST) typ = 0;
        else {
            if (pQuestion->type() == questRequest.type) typ = 1;
            else typ = 0.5;
        }
    }
    // Ronda final vale cualquier tipo
    else {
        if (pQuestion->type() == questRequest.type) typ = 1;
        else typ = 0.5;
    }
    // Evaluar dificultad
    // Si la dificultad coincide se valora con uno
    if (pQuestion->difficulty() == questRequest.difficulty) { dif = 1; }
    // Si no se valora en funcion de la dificultad requerida
    else {
        // Dificultad facil requerida, se prefieren las normales
        if (questRequest.difficulty == DIFF_EASY) {
            if (pQuestion->difficulty() == DIFF_NORM) dif = 0.5;
            else dif = 0.25;
        }
        // Dificultad normal, se prefieren faciles
        else if (questRequest.difficulty == DIFF_NORM) {
            if (pQuestion->difficulty() == DIFF_EASY) dif = 0.5;
            else dif = 0.25;
        }
        // Dificultad dificil, se prefieren normales
        else {
            if (pQuestion->difficulty() == DIFF_NORM) dif = 0.5;
            else dif = 0.25;
        }
    }
    // Devolver evaluacion
    return cat * typ * (0.75 * sub + 0.25 * dif);
}
```

Listado 5.2: Código de la función `valQuestion`.

- **`PairQuestion`:** Se asocia con la pregunta de tipo emparejar elementos. Cuenta con la pregunta y los elementos correctamente emparejados.

- **`HiddenQuestion`:** Representa la pregunta de tipo frase oculta. Incluye la pista y la frase oculta como variables miembro.

- **`IncompleteQuestion`:** Se asocia con la pregunta de tipo respuesta incompleta. Cuenta con la pregunta y la respuesta incompleta y completa.

5.4. APLICACIÓN PARA EL *CHROMA*

La plataforma *QuQuSI* requiere un componente gráfico que permita tanto a los concursantes como al público visualizar las preguntas del concurso.

Esta interfaz debe reaccionar ante las diferentes señales de control y eventos ocurridos durante el concurso. Ha de ofrecer un *feedback* que permita entender lo que está ocurriendo en todo momento.

Para poder combinar la imagen real obtenida por las cámaras con la imagen producida por computador, la interfaz debe permitir seleccionar entre diferentes modos de visualización, de modo que las preguntas puedan mostrarse tanto superpuestas como de forma autónoma.

Se requiere además la reproducción de contenidos multimedia asociados a los diferentes tipos de prueba.

Para garantizar que todas las preguntas se visualizan correctamente es adecuado contar con algún tipo de modo *debug* que permita comprobarlas rápidamente.

Un concurso necesita también un cuidado apartado sonoro. Los diferentes estados y eventos de la interfaz deben estar acompañados por música y sonidos que concuerden con las situaciones de juego.

Puesto que se trata de una aplicación que forma parte del sistema, debe ser capaz de comunicarse con el servidor y recibir las señales de control y eventos para sincronizarse con el estado del concurso.

A partir de estas necesidades, se plantean las siguientes funcionalidades y características para el componente gráfico de *QuQuSI*:

- Contar con una interfaz capaz de mostrar todos los elementos necesarios de cada tipo de pregunta.

- Su complejidad debe ser baja, pudiéndose entender de forma rápida e intuitiva en el menor tiempo posible a fin de evitar confusiones y perder tiempo dando explicaciones.

- La interfaz debe ofrecer un *feedback* ante las acciones de los usuarios, de forma que sea fácil seguir el desarrollo del concurso.
- Permitir diferentes modos de representación de las pruebas, permitiendo así que se pueda proyectar o combinar la imagen mediante la técnica del *chroma key*.
- Reproducir música y efectos de sonido acordes con el flujo de eventos del concurso, reforzando el *feedback* recibido por los participantes.
- Gestionar el volumen de las diferentes fuentes de sonido reproducidas de forma dinámica.
- Ha de ser capaz de reproducir diferentes formatos de contenidos multimedia, adaptando la interfaz y el sonido de forma adecuada en cada caso.
- Comunicarse con el servidor del concurso, recibiendo la información pertinente.
- En caso de perder la conexión debe tratar de reconectarse periódicamente, volviendo al estado adecuado en cuanto se recobre la comunicación.
- Ha de tener acceso a las preguntas y a los contenidos multimedia asociados.
- Contar con un modo *debug* que permita visualizar las preguntas de forma rápida.

La aplicación *chroma* se encarga de cumplir estas características (Fig. 5.8).

Esta aplicación, implementada en C++, se ocupa de la representación gráfica del concurso, de la música y de los efectos de sonido.

Utiliza el motor gráfico *OGRE 3D* (capítulo 3) para representar los diferentes tipos de preguntas mediante paneles (*overlays*) y texto que se ajusta dinámicamente al tamaño de su contenedor.

La reproducción de sonido y de contenidos multimedia se lleva a cabo con el *framework GStreamer* (capítulo 3). Implementa tres cauces de reproducción para tener varios efectos de sonido de forma simultánea, otro cauce para la música de fondo y otro más para los contenidos multimedia.

La aplicación utiliza OIS (capítulo 3) para gestionar la entrada del usuario, permitiendo que se cambie entre los diferentes modos de visualización, se salga de la aplicación y se controle el modo *debug*.

En su diseño se aplicaron los siguientes patrones:

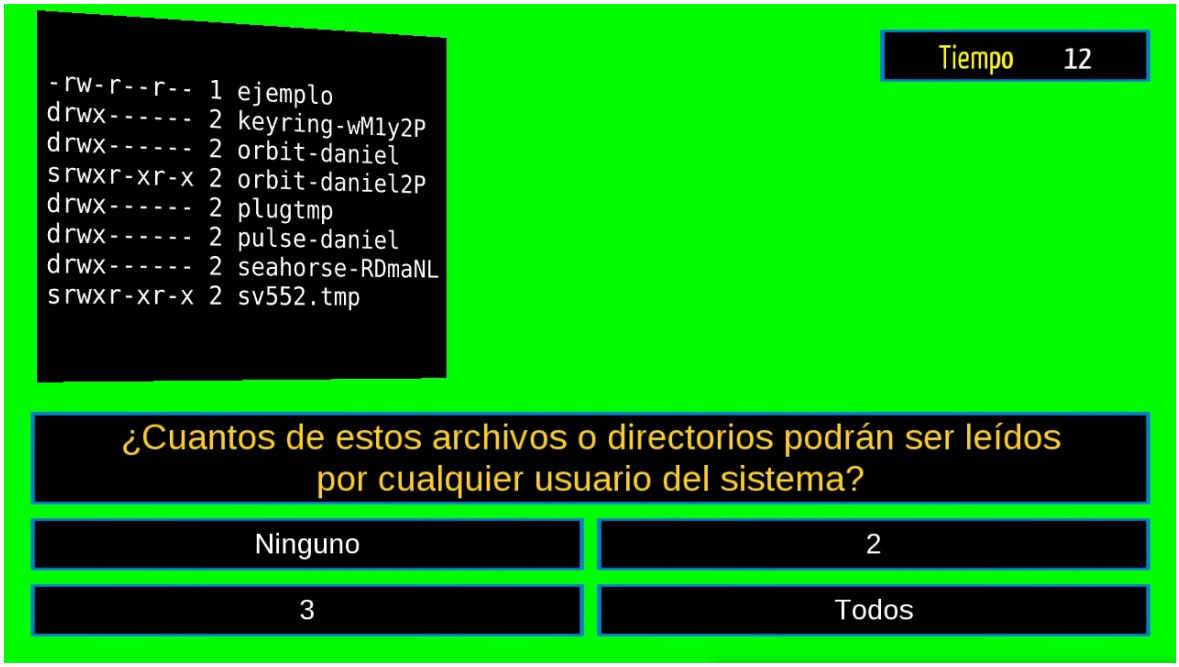

Figura 5.8: Captura de la interfaz de la aplicación para el *chroma*.

- ***Observer*:** Se utiliza para la lectura de *sockets* al igual que en el resto de aplicaciones. También se usa en los efectos de transición de los *overlays*.
- ***Singleton*:** El patrón es utilizado en los diferentes gestores de recursos.

Los módulos que forman la aplicación son:

- **Gestor de configuración:** Se trata del mismo módulo descrito en la sección 5.3. Permite cargar una lista de claves y valores desde un archivo.
- **Gestor de comunicaciones:** El módulo de comunicaciones de esta aplicación es una versión simplificada del descrito en la sección 5.3. Utiliza la función `select` para leer los datos recibidos desde el servidor a través de un *socket* TCP.
- **Gestor de preguntas:** Es el mismo módulo descrito en la sección 5.3. En este caso no se utiliza la selección dinámica de la pregunta pero si la recuperación directa a partir del identificador indicado por el servidor.
- **Gestor de overlays:** Es el módulo encargado de gestionar los diferentes paneles y fuentes de texto *TrueType* de la interfaz. Permite mostrar, cambiar el color y ocultar cada tipo de panel. También redimensiona el texto al tamaño del panel que lo contiene, facilitando la lectura. Cuenta con los siguientes submódulos:

  - **Panel de preguntas:** Este *overlay* muestra la pregunta. Cambia en función del tipo de pregunta que se quiere mostrar.
  - **Panel de puntuaciones:** Es el panel en el que se muestra la puntuación de cada equipo. Al final de cada fase muestra el orden del podio.
  - **Panel de contador:** Controla el panel en el que se muestran las cuentas atrás para responder y contraatacar.
  - **Panel de vídeo:** Gestiona el plano 3D sobre el que se reproducen los archivos multimedia de imagen y vídeo. La reproducción se lleva a cabo gracias al uso de *GStreamer*.

- **Gestor multimedia:** Es el módulo encargado de la reproducción de los contenidos multimedia. Con el uso de *GStreamer* se crean cuatro *pipelines* genéricos con los que se puede reproducir la música de fondo y hasta tres efectos de sonido de forma simultánea y un *pipeline* personalizado para convertir imágenes y vídeos en texturas que se aplican al panel de vídeo.

- **Gestor de *chroma***: Es el módulo superior que se encarga de actualizar los bucles principales de *OGRE 3D* y *GStreamer*, además de gestionar la pulsación de teclas y el modo *debug*. Llama a los diferentes métodos del resto de gestores en función de las señales recibidas desde el módulo de comunicación.

Gracias al módulo de *overlays* y al *framework OGRE 3D* se representan de forma clara y completa cada uno de los diferentes tipos de preguntas, además de otra información relevante del concurso, como la puntuación y el tiempo.

Mediante los cambios en el color de la interfaz gestionados por el módulo de *overlays* y la reproducción de varios efectos de sonido por el gestor de sonido se consigue el *feedback* que los participantes necesitan para seguir el concurso.

La reproducción de música, efectos de sonido y contenidos multimedia y la gestión dinámica del volumen se logran gracias a los diferentes cauces de reproducción implementados con el *framework GStreamer* en los gestores de *overlays* y sonido.

El uso del patrón *observer* junto con los *sockets* TCP en el módulo de comunicación satisfacen las necesidades de conexión con el servidor.

El módulo de *chroma* cuenta con un modo *debug* controlado desde teclado gracias a OIS.

En las siguientes secciones se detallan los módulos que componen la aplicación.

5.4.1. Módulo de configuración

Se trata exactamente del mismo módulo descrito en la sección 5.3.1. Carga los parámetros desde un archivo de configuración.

5.4.2. Módulo de comunicación

El módulo de comunicación de las aplicaciones cliente reutiliza las clases `QSelect` y `QSelectMgr` descritas en la sección 5.3.2 para gestionar la conexión con el servidor.

La clase `QChroma` implementa la interfaz `QSelect` y se añade al conjunto de descriptores de archivo de lectura de la clase `QSelectMgr` con la función `initSocket`. De este modo, cuando hay datos disponibles, se llama a `recvChromaMsg` para recibir el mensaje y a `processChromaMsg` para tratarlo. La función `recvChromaMsg` se encarga también de tratar de recuperar la conexión periódicamente en caso de haberse perdido (Fig. 5.9). Con esta solución, se consigue un código más claro y reutilizable al delegar el tratamiento del mensaje a la función `processChromaMsg`.

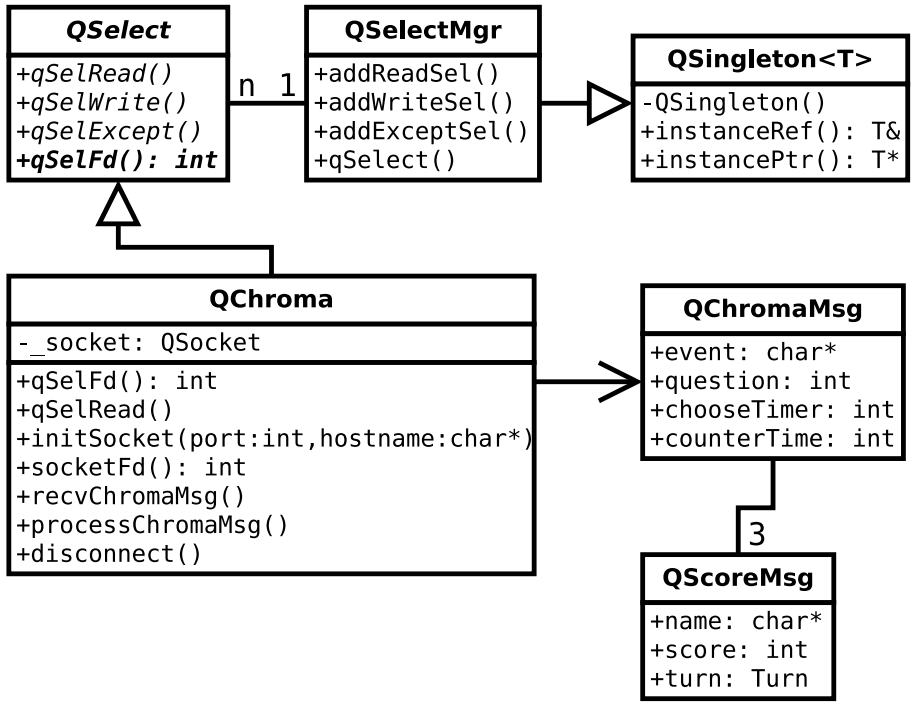

Figura 5.9: Diagrama de clases del módulo de comunicación de la aplicación *chroma*.

La función `disconnect` se llama al terminar la ejecución de la aplicación para cerrar la conexión con el servidor.

El protocolo de comunicación se implementa en la clase `QChromaMsg`. El mensaje que se recibe del servidor incluye los siguientes campos:

- **`event`:** El evento acontecido en el servidor que produjo el envío del mensaje.
- **`question`:** Identificador de la pregunta actual (valor negativo en caso de no proceder).
- **`chooseTimer`:** Tiempo actual de la cuenta atrás para responder.
- **`counterTime`:** Tiempo actual de la cuenta atrás para contraatacar.
- **`teamA`:** Estructura que contiene información sobre el equipo A (nombre, puntuación y turno)
- **`teamB`:** Igual que el anterior pero con los datos del equipo B.
- **`teamC`:** Lo mismo que en los anteriores, conteniendo la información del equipo C.

Gracias a la utilización de un protocolo de comunicación sencillo para cada aplicación frente a uno complejo compartido entre todas las aplicaciones se simplifica el código, se aumenta su mantenibilidad y se reduce el tráfico de red al enviarse los datos estrictamente necesarios a cada aplicación.

5.4.3. Módulo de preguntas

Es el mismo módulo detallado en la sección 5.3.4. La aplicación para el *chroma* se limita a utilizar la función `getQuestion` para obtener la pregunta cuyo identificador recibe desde el servidor *QuQuSI*.

5.4.4. Módulo de *overlays*

Esté módulo gestiona los diferentes paneles de la interfaz gráfica del concurso (Fig. 5.10).

El módulo se divide en cuatro submódulos, cada uno de los cuales implementa la interfaz `FrameListener` proporcionada por *OGRE 3D*. De este modo, se suscriben a la raíz del motor gráfico y utilizan la función `frameStarted` para actualizar sus paneles antes de dibujar cada *frame*.

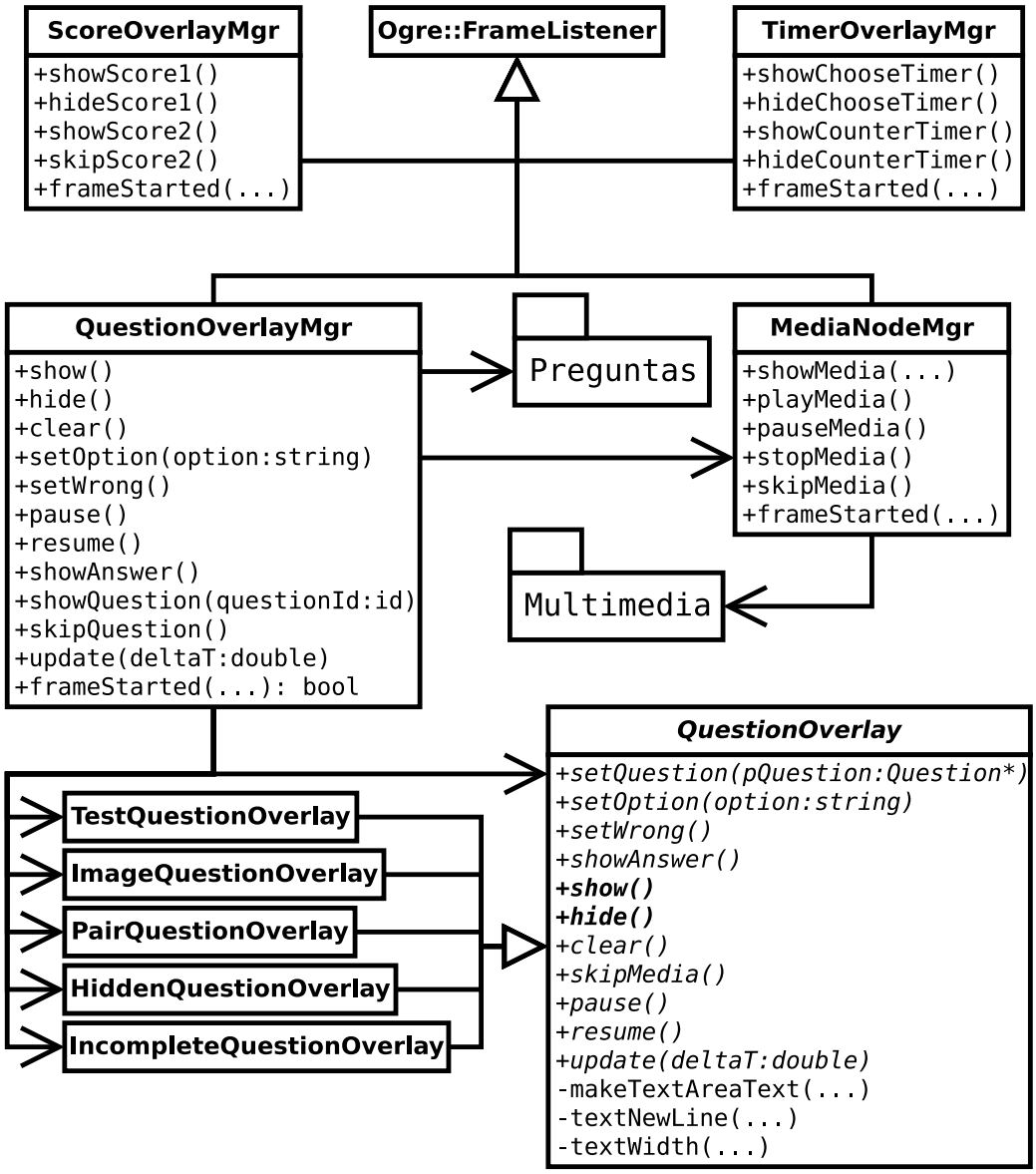

Figura 5.10: Diagrama de clases del módulo de *overlays*.

Panel de puntuación

Es el gestor que se encarga de los *overlays* en los que se muestran los nombres de los equipos y sus puntuaciones.

Cuenta con funciones para mostrar y ocultar los paneles en los dos modos gráficos con los que cuenta la aplicación.

Panel de contadores

Este gestor se ocupa de los *overlays* de las cuentas atrás para contestar y para realizar un contraataque.

Sus funciones permiten mostrar, actualizar y ocultar cada uno de los dos tipos de contadores.

Panel de preguntas

Este gestor controla la representación visual de los diferentes tipos de preguntas. Utiliza el gestor de nodo multimedia en aquellas preguntas que tienen algún contenido multimedia asociado y recupera las preguntas mediante la función `getQuestion` del módulo de preguntas.

Las clase `QuestionOverlay` proporciona una interfaz a implementar por cada tipo de pregunta disponible. De este modo, para cada tipo de pregunta existe una clase que gestiona los paneles necesarios para mostrarla. De esta función cabe destacar las funciones `makeTextAreaText` (Listado 5.3), `textNewLine` (Listado 5.4) y `textWidth` (Listado 5.5), las cuales permiten ajustar el tamaño de la fuente al panel que las contiene. `makeTextAreaText` va decrementando el tamaño de la fuente hasta que el texto cabe en el panel, `textNewLine` inserta los saltos de línea en el texto y `textWidth` calcula el tamaño de una palabra con una fuente *TrueType* con un tamaño determinado. Gracias a esto, se ajusta el texto al máximo, de forma que se logra un aprovechamiento del espacio de visualización óptimo.

La clase `QuestionOverlayManager` es el núcleo de este submódulo. Delega la funcionalidad requerida desde el módulo de *chroma* a la pregunta activa, gracias al polimorfismo. Para ello, cuenta con un puntero a cada una de las clases que heredan de la clase base `QuestionOverlay` y otro más a dicha clase. Cuando se recupera una pregunta, se consulta su tipo y se fija como activa la clase correspondiente. Esta clase cuenta con los métodos necesarios para mostrar las preguntas y reproducir los efectos necesarios en función de los eventos ocurridos durante el concurso. Gracias a este planteamiento, se pueden implementar

```
std::string makeTextAreaText(std::string text,
                             Ogre::OverlayElement* pTextArea) {
    // Cast de OverlayElement a TextAreaOverlayElement
    Ogre::TextAreaOverlayElement* pTextAreaOE =
        static_cast<Ogre::TextAreaOverlayElement*>(pTextArea);
    // Altura del panel que contiene el area de texto
    double panelHeight = pTextArea->getParent()->getHeight();
    // Cadena resultante
    std::string textAreaText;
    // Altura de los caracteres
    double height = 1;
    double charHeight;
    // Saltos de linea
    size_t n;
    // Mientras el texto desborde o haya mas de dos lineas
    do {
        // Decrementar tamano
        height = height - 0.05;
        charHeight = panelHeight * height;
        // Fijar tamano
        pTextAreaOE->setCharHeight(charHeight);
        // Aplicar saltos de linea
        textAreaText = textNewLine(text, pTextArea);
        // Contar saltos de linea
        n = std::count(textAreaText.begin(), textAreaText.end(), '\n');
        // Centrar el texto
        pTextArea->setTop(
            (double) (n + 1) * -(pTextAreaOE->getCharHeight() * 0.5));
    } while (
        // Dos lineas maximo
        (n > 1) ||
        // El texto no se sale de la caja que lo contiene
        (n + 1) * pTextAreaOE->getCharHeight() >  panelHeight * 0.95);
    // Devolver cadena con saltos de linea
    return textAreaText;
}
```

Listado 5.3: Código de la función `makeTextAreaText`.

```
std::string textNewLine(std::string text,
                        Ogre::OverlayElement* pTextArea) {
    // Strings de entrada, salida y palabra
    std::istringstream textISS(text);
    std::ostringstream textOSS;
    std::string word;
    // Anchura de la linea de area de texto actual
    double textAreaWidth = 0.0;
    // Anchura del panel que contiene al area de texto
    double panelWidth = pTextArea->getParent()->getWidth();
    // Anchura de la palabra actual
    double wordWidth;
    // Anchura de espacio en blanco
    double spaceWidth = textWidth("␣", pTextArea);
    // Cast de OverlayElement a TextAreaOverlayElement
    Ogre::TextAreaOverlayElement* pTextAreaOE =
        static_cast<Ogre::TextAreaOverlayElement*>(pTextArea);
    // Corregir anchura espacios
    pTextAreaOE->setSpaceWidth(spaceWidth);
    // Mientras haya palabras en la cadena
    while (textISS >> word) {
        // Calcular anchura en pantalla
        wordWidth = textWidth(word, pTextArea);
        // Si se sobrepasa el ancho del panel
        if (textAreaWidth + spaceWidth + wordWidth >
            panelWidth * 0.95) {
            // Anadir salto de linea y palabra
            textOSS << "\n" << word;
            // Reiniciar anchura de la nueva linea
            textAreaWidth = wordWidth;
        }
        // Si no se sobrepasa el ancho del panel
        else {
            // Para la primera palabra
            if (textOSS.str().empty()) {
                // Anadir palabra
                textOSS << word;
                // Incrementar anchura de la linea acual
                textAreaWidth += wordWidth;
            }
            // Para cualquier otra palabra
            else {
                // Anadir espacio y palabra
                textOSS << "␣" << word;
                // Incrementar anchura de la linea actual
                textAreaWidth += spaceWidth + wordWidth;
            }
        }
    }
    // Devolver cadena con saltos de linea
    return textOSS.str();
}
```

Listado 5.4: Código de la función `textNewLine`.

```
double textWidth(std::string text, Ogre::OverlayElement* pTextArea) {
    // Convertir texto a cadena de caracteres anchos
    wchar_t aux[text.length() + 1];
    memset(aux, 0, sizeof(aux));
    mbstowcs(aux, text.c_str(), text.length());
    std::wstring wtext(aux);
    // Obtener nombre de fuente
    std::string fontName =
        static_cast<Ogre::TextAreaOverlayElement*>(
            pTextArea)->getFontName();
    // Obtener gestor de fuentes
    Ogre::FontManager* pFontMgr =
        Ogre::FontManager::getSingletonPtr();
    // Obtener puntero a fuente
    Ogre::Font* pFont =
        static_cast<Ogre::Font*>(pFontMgr->getByName(fontName).get());
    // Calcular ancho de area de texto en pantalla
    double textWidth = 0.0;
    for (wchar_t c: wtext) {
        // Existe un bug en el ancho de los espacios
        if (c == ' ')
            c = '0';
        // Obtener relacion de aspecto
        textWidth += pFont->getGlyphAspectRatio(c);
    }
    // Obtener ancho multiplicando relacion de aspecto por alto
    textWidth *= static_cast<Ogre::TextAreaOverlayElement*>(
        pTextArea)->getCharHeight();
    // Obtener ancho multiplicando por relacion de aspecto ventana
    textWidth *= (double) _pWindow->getHeight() / _pWindow->getWidth();
    // Devolver ancho del texto
    return textWidth;
}
```

Listado 5.5: Código de la función `textWidth`.

nuevos tipos de preguntas sin tener que realizar grandes cambios en el módulo.

Nodo multimedia

Es el gestor que se encarga del panel tridimensional sobre el que se reproducen los contenidos multimedia.

Para ello, se comunica con el módulo multimedia, el cual proporciona la funcionalidad para reproducir cualquier tipo de contenido. Cuenta con funciones para mostrar el panel, reproducir, pausar y parar el contenido, y con funciones relacionadas con el volumen.

5.4.5. Módulo de multimedia

Este módulo se encarga de la reproducción de contenidos multimedia mediante el uso de la biblioteca *GStreamer* (Fig. 5.11).

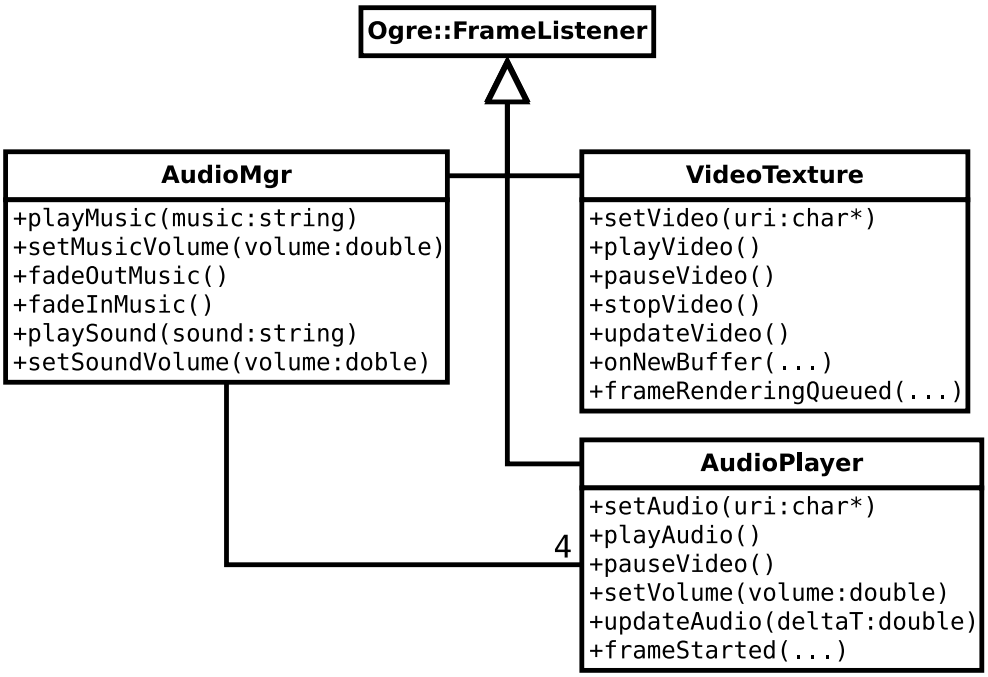

Figura 5.11: Diagrama de clases del módulo multimedia.

Cuenta con tres clases que implementan la interfaz `FrameListener` proporcionada por *OGRE 3D*, de modo que se actualizan a si mismas en cada iteración del bucle gráfico.

La clase `VideoTexture` permite el volcado de imágenes y vídeos en una zona de memoria que se asigna a una textura de *OGRE 3D*. Esto se logra gracias a que el *pipeline* para la reproducción implementado termina en un elemento para este fin (Lista-

do 5.6). La clase proporciona funciones para cargar, reproducir y pausar el contenido, y se encarga de su actualización mediante las funciones `updateVideo`, `onNewBuffer` y `frameRenderingQueued`. Esta solución proporciona un *widget* reutilizable para la reproducción de vídeo en *OGRE 3D* en tiempo real, sin bloquear el bucle gráfico en ningún momento por no estar disponible el siguiente *frame*.

La clase `AudioPlayer` emplea un *pipeline* genérico (*playbin2*) para la reproducción de sonidos. Permite fijar, reproducir, pausar y cambiar el volumen del sonido. Implementa `frameStarted` para habilitar el efecto de fundido del volumen, de modo que se pueda decrementar o incrementar gradualmente. Esta clase es totalmente reutilizable, pudiendo usarse en cualquier proyecto con *OGRE 3D*.

La clase `AudioMgr` utiliza `AudioPlayer` para reproducir la música de fondo y los efectos de sonido. Cuenta con una instancia de `AudioPlayer` para la música de fondo y tres más para la reproducción de hasta tres efectos de sonido simultáneos. Las funciones `fadeOutMusic` y `fadeInMusic` permiten decrementar e incrementar gradualmente la música de fondo, respectivamente. Cuando se pide a `AudioMgr` que reproduzca un sonido, busca el primer `AudioPlayer` que haya terminado su reproducción y le asigna el nuevo sonido a reproducir. Gracias a esta solución, se pueden reproducir varias pistas de sonido de forma simultanea.

5.4.6. Módulo de *chroma*

El módulo de *chroma* se encarga de la inicialización de la aplicación y del control de su lógica (Fig. 5.12).

La función `init` de la clase `QChroma` recibe los argumentos leídos desde el módulo de configuración e inicializa la interfaz gráfica, los dispositivos de entrada y la conexión con el servidor.

La función `initSocket` realiza la conexión con el servidor en la dirección y el puerto indicados en sus argumentos.

En la función `initOgre` inicializa el *framework*, creando la raíz, el gestor de escena, la cámara y la ventana y cargando los recursos. Una vez iniciado *OGRE 3D*, se carga el módulo de *overlays*.

La función `initOIS` inicializa los dispositivos de entrada. Se crean los objetos del teclado y el ratón, con los que se puede salir de la aplicación.

La función `initChroma` realiza una consulta al servidor sobre el estado del concurso.

En la función `processChromaMsg` se lleva a cabo el tratamiento de los eventos recibidos desde el servidor. En función del evento se utilizan diferentes métodos del módulo de

```
// Inicializar GStreamer
gst_init(0, 0);
// Inicializar soporte para hilos
if (!g_thread_supported())
    g_thread_init(0);
// Crear tuberia de reproduccion playbin2
_pPlayBin = gst_element_factory_make("playbin2", nullptr);
// Escuchar mensajes del bus de playbin2
GstBus* bus = gst_pipeline_get_bus(GST_PIPELINE(_pPlayBin));
gst_bus_add_watch(bus, onBusMessage, getUserData(_pPlayBin));
gst_object_unref(bus);
// Emitir senal cuando el video vaya a terminar
g_signal_connect(G_OBJECT(_pPlayBin), "about-to-finish",
                 G_CALLBACK(aboutToFinish), this);
// Crear appsink para volcar la salida
_pAppSink = gst_element_factory_make("appsink", "app_sink");
// Emitir senales
g_object_set(G_OBJECT(_pAppSink), "emit-signals", true, nullptr);
// Bufferes maximos
g_object_set(G_OBJECT(_pAppSink), "max-buffers", 1, nullptr);
// Reproducir en tiempo real (descartar frames)
g_object_set(G_OBJECT(_pAppSink), "drop", true, nullptr);
// Escuchar senales de nuevo buffer
g_signal_connect(G_OBJECT(_pAppSink), "new-buffer",
                 G_CALLBACK(onNewBuffer), this);
// Crear un filtro para producir datos bgra
GstCaps* caps =
    gst_caps_new_simple("video/x-raw-rgb",
                        "red_mask",   G_TYPE_INT, 0x0000FF00,
                        "green_mask", G_TYPE_INT, 0x00FF0000,
                        "blue_mask",  G_TYPE_INT, 0xFF000000,
                        "alpha_mask", G_TYPE_INT, 0x000000FF,
                        nullptr);
GstElement* rgbFilter =
    gst_element_factory_make("capsfilter", "rgb_filter");
g_object_set(G_OBJECT(rgbFilter), "caps", caps, nullptr);
gst_caps_unref(caps);
// Crear una bin donde combinar el filtro con el appsink
GstElement* appBin = gst_bin_new("app_bin");
// Anadir el filtro a la bin
gst_bin_add(GST_BIN(appBin), rgbFilter);
// Conectar el sink de bin al sink de capsfilter mediante ghostpad
GstPad* rgbSinkPad = gst_element_get_static_pad(rgbFilter, "sink");
GstPad* ghostPad = gst_ghost_pad_new("app_bin_sink", rgbSinkPad);
gst_object_unref(rgbSinkPad);
gst_element_add_pad(appBin, ghostPad);
// Anadir el appsink a la bin
gst_bin_add_many(GST_BIN(appBin), _pAppSink, nullptr);
gst_element_link_many(rgbFilter, _pAppSink, nullptr);
// Reemplazar la ventana de salida por defecto con la appsink
g_object_set(G_OBJECT(_pPlayBin), "video-sink", appBin, nullptr);
```

Listado 5.6: Código de la creación de un *playbin2* para la reproducción de vídeo con *OGRE 3D*.

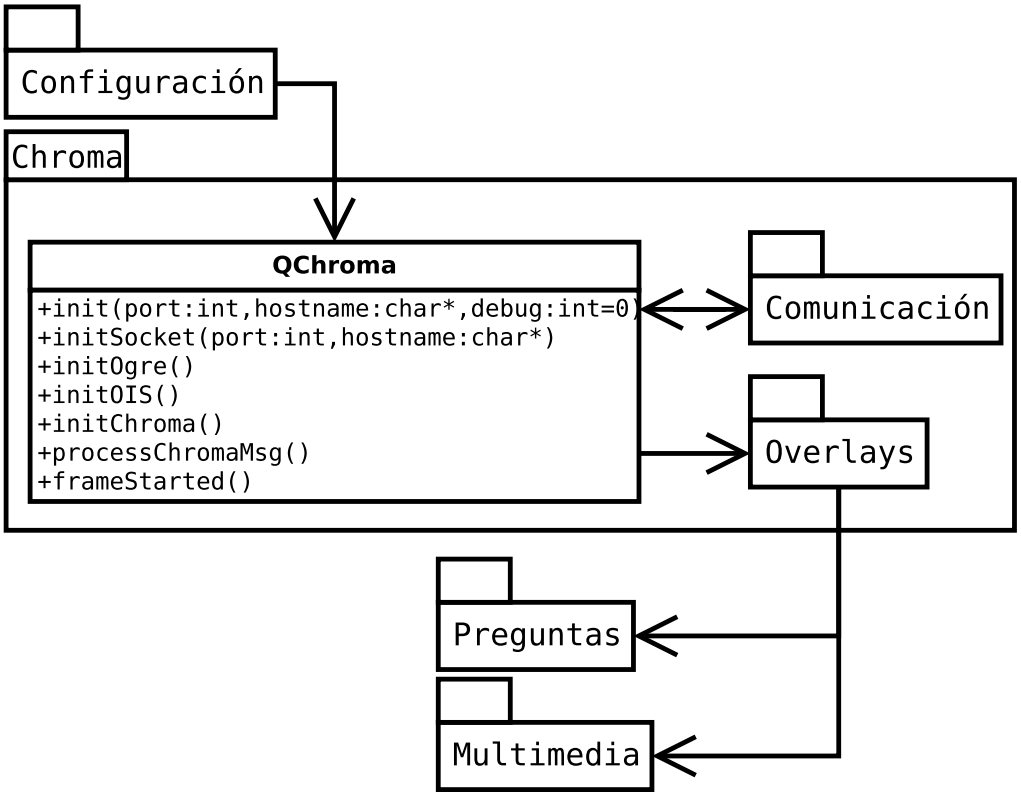

Figura 5.12: Diagrama de clases del módulo *chroma*.

overlays, el cual se encarga de representar el estado del concurso.

5.5. APLICACIÓN PARA LOS MARCADORES

Un elemento recurrente en los concursos televisivos son los marcadores de puntuación. La plataforma *QuQuSI* también da soporte a este aspecto.

Este tipo de interfaces permiten visualizar el nombre de cada concursante o equipo, su puntuación y otra información como el turno.

Los marcadores son otra parte del sistema distribuido por lo que requieren capacidad de conexión con el servidor, de forma que conozcan la información relevante del estado del concurso.

Así, las funcionalidades deseables en una aplicación de marcadores son:

- Permitir mostrar el nombre del concursante o equipo.
- Mostrar la puntuación actualizada en cada momento.
- Ofrecer un *feedback* ante las acciones de los concursantes.
- Visualizar otro tipo de información como el turno o el orden de respuesta.
- Configurar el nombre del equipo al que corresponde el marcador.

La aplicación para los marcadores da soporte a estas funcionalidades (Fig. 5.13).

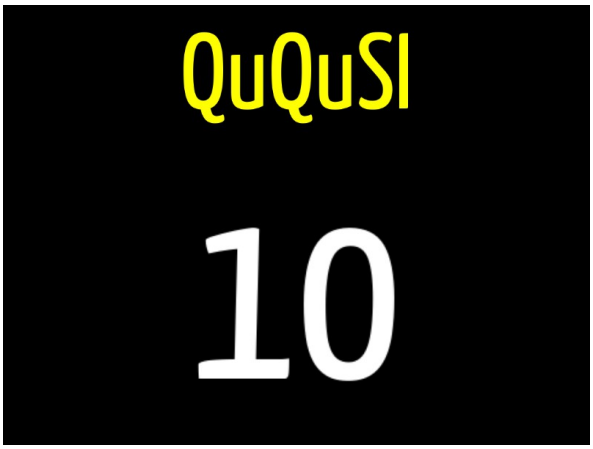

Figura 5.13: Captura de la interfaz de la aplicación para los marcadores.

Está implementada en C++ y se ocupa de representar de forma gráfica la información de cada equipo.

Para ello, utiliza el motor gráfico *OGRE 3D* (capítulo 3). Mediante el uso de *overlays* y fuentes de texto *TrueType* muestra en pantalla el nombre y la puntuación de un equipo, cambiando además de color en función del turno.

En su diseño se tuvieron en cuenta los siguientes patrones:

- ***Observer:*** Se utiliza para la lectura de *sockets* en el módulo de comunicación.
- ***Singleton:*** Se implementa en los diferentes módulos de la aplicación.

La aplicación está compuesta de los siguientes módulos:

- **Gestor de configuración:** Es el mismo módulo descrito en la sección 5.3. Permite cargar una serie de parámetros desde un archivo de configuración.
- **Gestor de comunicación:** Es una versión igual a la descrita en la sección 5.4. Permite la comunicación con el servidor y se reconecta automáticamente en caso de perder la conexión.
- **Gestor de marcadores:** Es el módulo superior de la aplicación. Gestiona los mensajes recibidos desde el módulo de comunicación, actualizando el texto y el color de los *overlays*. Llama al bucle principal de *OGRE 3D* y gestiona la entrada por teclado.

La información se presenta en pantalla gracias a la gestión de los *overlays* de *OGRE 3D* por parte del módulo de marcador.

Esta información se mantiene actualizada gracias al módulo de comunicación.

El nombre del equipo al que corresponde el marcador se indica utilizando el gestor de configuración.

A continuación se describen los módulos que componen la aplicación:

5.5.1. Módulo de configuración

Es el mismo módulo descrito en la sección 5.3.1. Carga los parámetros desde un archivo de configuración.

5.5.2. Módulo de comunicación

Es una versión muy similar a la descrita en la sección 5.4.2 (Fig. 5.14).

Se diferencia en el tratamiento que se da al mensaje recibido y en el protocolo utilizado, el cual está implementado en la estructura `QScoreMsg`. Esta estructura cuenta con los siguientes campos:

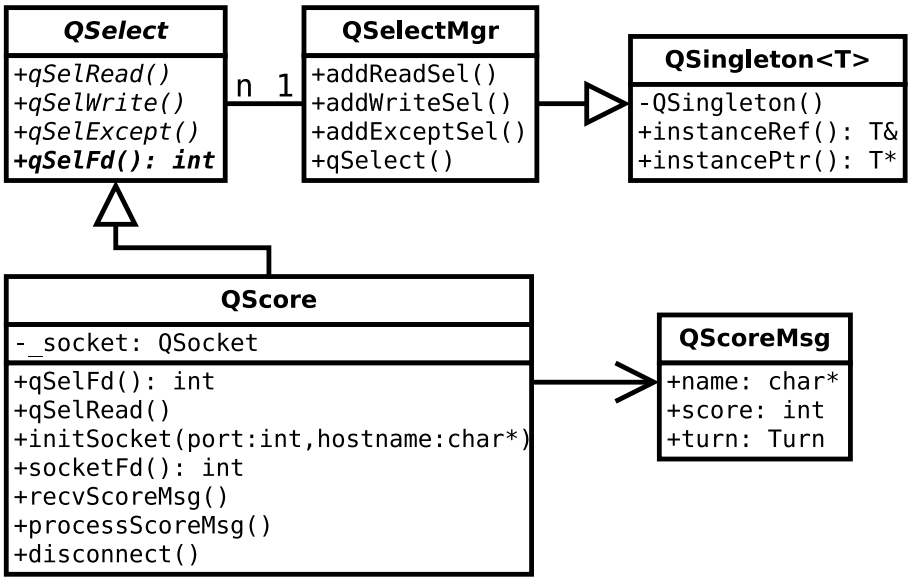

Figura 5.14: Diagrama de clases del módulo de comunicación de la aplicación para los marcadores.

- **`name`:** El nombre del equipo al que pertenece el marcador.
- **`score`:** La puntuación actual del equipo.
- **`Turn`:** Es una enumeración que cuenta con tres valores; `YES` si el equipo tiene el turno, `NO` si no lo tiene y `QUEUED` si está en la cola de turnos.

El uso de este sencillo protocolo de comunicación frente a uno más complejo que pudiera ser compartido entre todas las aplicaciones disminuye el tráfico de red generado y simplifica el código para el tratamiento de los mensajes.

5.5.3. Módulo de marcadores

El módulo de marcadores se encarga de la inicialización, la gestión de la ventana gráfica y la entrada por teclado (Fig. 5.15).

La función `initSocket` toma como argumento la dirección y el puerto del servidor y gestiona la conexión con este.

En la función `initOgre` inicializa el *framework*, creando la raíz, el gestor de escena, la cámara y la ventana y cargando los recursos. Se ocupa entonces de precargar las fuentes de texto utilizadas y de recuperar del gestor de *overlays* los paneles utilizados para mostrar el nombre y la puntuación del equipo.

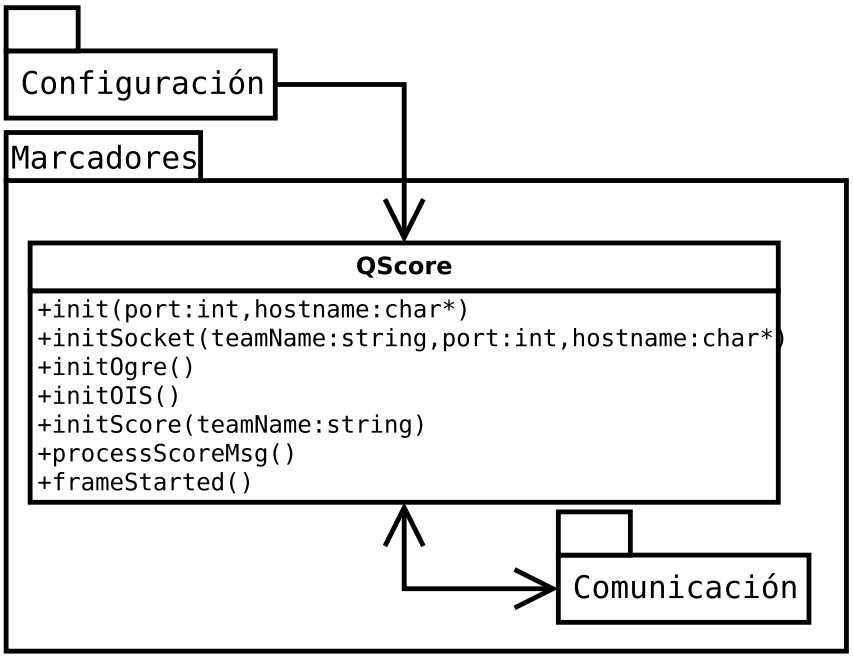

Figura 5.15: Diagrama de clases del módulo de marcadores.

La función `initOIS` inicializa los dispositivos de entrada. Se crea el objeto que representa al teclado, que se utiliza para salir de la aplicación, y el que representa al ratón, el cual no se usa pero se debe crear para ocultar el puntero.

La función `initScore` toma como argumento el identificador del equipo con el que se configuró el marcador y consulta en el servidor el nombre de dicho equipo, inicializando la interfaz con esta información.

La función `frameStarted` se ejecuta con cada iteración del bucle gráfico y en ella se llama a la función `qSelect` del módulo de comunicación y se trata los eventos de teclado.

5.6. APLICACIÓN PARA LOS PULSADORES

Una plataforma como *QuQuSI* requiere la gestión de dispositivos físicos con los que los concursantes puedan interactuar con el sistema.

La reacción ante la activación de uno de estos dispositivos debe ser rápida, ofreciendo el *feedback* adecuado al usuario.

Los pulsadores son parte del sistema, por lo que necesitan conectarse con el servidor a fin de enviar las pulsaciones.

Las funcionalidades de una aplicación que permita la interacción con pulsadores son:

- Transmitir las pulsaciones de forma rápida y en el orden correcto.

- Gestionar la conexión con el servidor, reconectándose en caso de fallo.
- Comunicarse con el puerto serie en el que se conecten los pulsadores.
- Controlar la conexión con el puerto serie, recuperando la conexión en caso de perderse.

La aplicación para los pulsadores realiza estas funcionalidades.

Las alternativas para llevar a cabo la comunicación con dispositivos físicos son muy numerosas. De entre todas ellas se eligió utilizar la plataforma *Arduino* por los motivos descritos en el capítulo 3.

La comunicación con *Arduino* se realiza a través del puerto serie. En *GNU/Linux*, esta comunicación se realiza mediante un archivo de dispositivo con las propiedades de un terminal virtual (*Teletypewriter*, TTY).

Los terminales virtuales requieren una configuración previa a su uso, teniendo que fijar la velocidad de transferencia y el modo.

Esta aplicación cuenta con dos partes. La primera, escrita en el lenguaje de programación *Arduino*, se carga en el microcontrolador de la placa *Arduino* y detecta la pulsación de los interruptores en el circuito electrónico. La segunda, escrita en C++, hace de puente entre el puerto serie de *Arduino* y el servidor *QuQuSI*.

En el diseño de esta aplicación se emplearon los siguientes patrones:

- ***Observer:*** Se usa en el módulo de comunicación para llevar a cabo la lectura síncrona de *sockets*.
- ***Singleton:*** Se emplea en los gestores que forman la aplicación.

La aplicación está formada por los siguientes módulos:

- **Gestor de configuración:** Se trata del módulo para cargar la configuración desde un archivo. Es igual al descrito en la sección 5.3.
- **Gestor de comunicación:** Es una versión muy similar a la descrita en la sección 5.4. Puesto que la comunicación con el servidor es unidireccional en el caso de ésta aplicación, se reutiliza el módulo para recibir datos desde el puerto serie en lugar de desde un *socket*. Trata de recuperar la conexión tanto con el servidor como con el puerto serie periódicamente en caso de perderla.
- **Gestor de pulsadores:** Es el módulo superior de la aplicación. Hace de puente entre el módulo de terminal y el módulo de comunicación, enviando los mensajes recibidos desde el puerto serie a través del *socket* al servidor.

Las pulsaciones se transmiten en el orden correcto gracias al código cargado en el microprocesador de la placa *Arduino*.

El diseño simple y directo de la aplicación logra que las pulsaciones se transmiten de forma rápida.

La gestión de la reconexión se lleva a cabo en el gestor de terminal en el caso del puerto serie y en el gestor de comunicación en el caso del servidor.

Los módulos que forman la aplicación se describen en las siguientes secciones.

5.6.1. Módulo de configuración

Es el mismo módulo descrito en la sección 5.3.1. Carga los parámetros desde un archivo de configuración.

5.6.2. Módulo de comunicación

El módulo de comunicación de la aplicación para los pulsadores es similar al descrito en la sección 5.4.2 (Fig. 5.16).

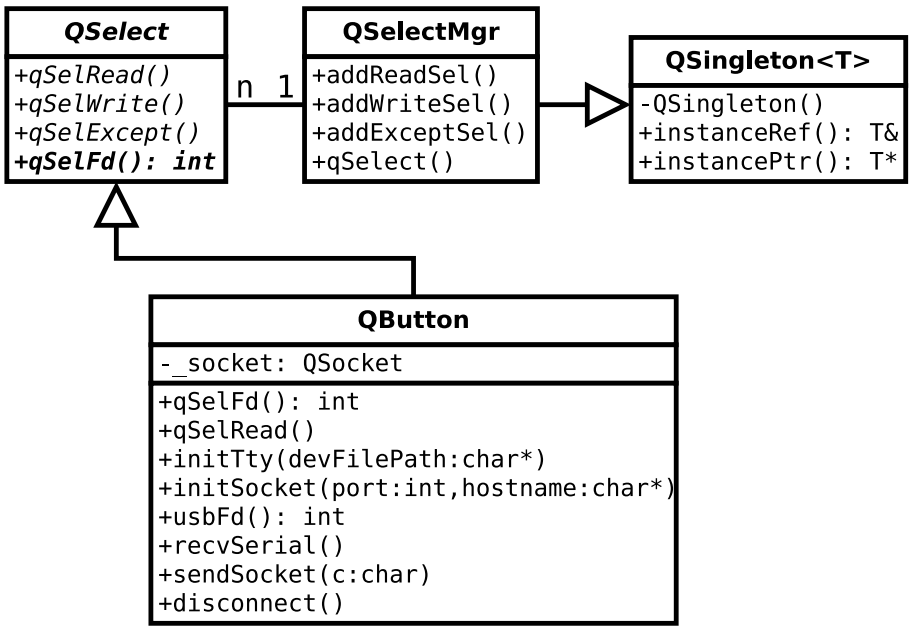

Figura 5.16: Diagrama de clases del módulo de comunicación de la aplicación para los pulsadores.

La principal diferencia es que el descriptor de archivo que utiliza para recibir datos no es el de un *socket*, sino el del archivo de dispositivo que representa al puerto USB. Se añade

este descriptor al conjunto de lectura de `QSelectMgr` y, cuando se recibe un carácter desde el microcontrolador, se gestiona en la función `sendSocket`.

El microcontrolador *Arduino* se encarga de la comunicación con el mundo físico a través del circuito electrónico al que se conectan los pulsadores (Fig. 5.17).

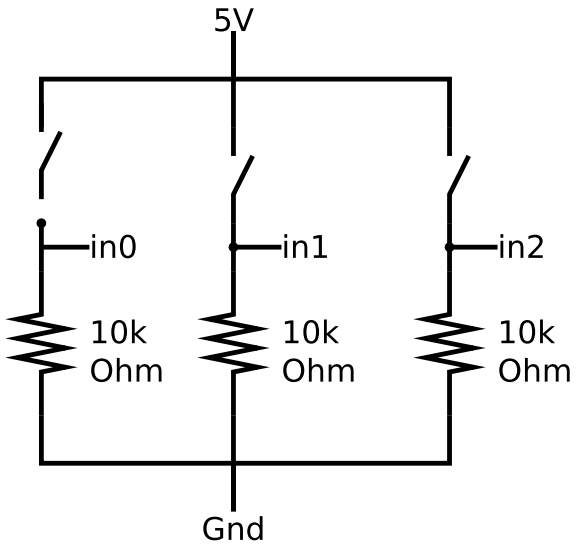

Figura 5.17: Circuito electrónico conectado al microcontrolador *Arduino*.

El programa del microcontrolador inicializa el puerto serie y los pines de la placa y entra en el bucle de ejecución. En cada iteración detecta si uno de los pulsadores acaba de ser presionado. En caso afirmativo, envía el carácter asociado por el puerto serie (Listado 5.7).

5.6.3. Módulo de pulsadores

Este módulo se encarga de la inicialización del puerto serie y el *socket* y la conexión con el microcontrolador *Arduino* y el servidor *QuQuSI* (Fig. 5.18).

La clase `QButton` recibe en su función `init` los parámetros de configuración desde el módulo de configuración. La función `initTty` trata de conectarse con el puerto serie a través del archivo de dispositivo indicado por argumento. Por su parte, `initSocket` realiza la conexión con el servidor en la dirección y el puerto indicados. La función `sendSocket` compone el mensaje que contiene al evento que se envía al servidor *QuQuSI*.

5.7. APLICACIÓN WEB DE CONTROL

Un sistema como *QuQuSI* necesita una interfaz para enviar señales al servidor, controlando así el desarrollo del concurso en tiempo real.

```
#define BUTTONS 3
int buttons[] = {2, 3, 4}; // Pulsadores
int pressed[] = {0, 0, 0}; // Presionado
int LEDs[] = {10, 11, 12}; // LEDs
char events[] = {'a', 'b', 'c'}; // Eventos
void setup() {
  for (int i = 0; i < BUTTONS; i++)  {
    pinMode(buttons[i], INPUT); // Pulsadores
    pinMode(LEDs[i], OUTPUT); // LEDs
  }
  Serial.begin(9600); // Puerto serie
}
void setPressed(int i) {
   if (pressed[i])
     return;
   digitalWrite(LEDs[i], HIGH);
   pressed[i] = 1;
   Serial.print(events[i]);
}
void clearPressed(int i) {
  if (!pressed[i])
    return;
  digitalWrite(LEDs[i], LOW);
  pressed[i] = 0;
}
void loop() {
  for (int i = 0; i < BUTTONS; i++)
    if (digitalRead(buttons[i]) == HIGH)
      setPressed(i);
    else
      clearPressed(i);
  delay(10);
}
```

Listado 5.7: Código del programa cargado en el microcontrolador *Arduino*.

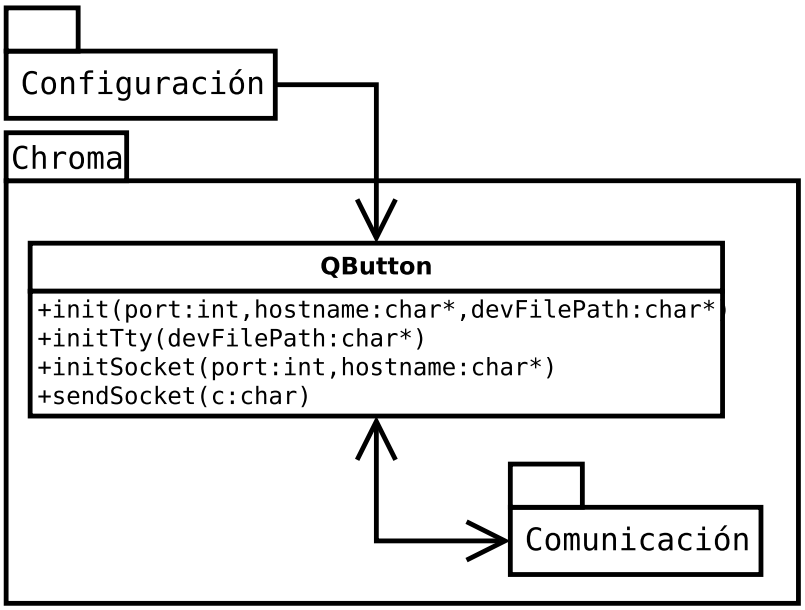

Figura 5.18: Diagrama de clases del módulo de pulsadores.

Esta interfaz debe reaccionar ante los cambios de estado y los eventos que ocurran durante el concurso, actualizándose de forma automática.

Debe resultar intuitiva y fácil de usar, previniendo así errores humanos que pudieran alterar el correcto desarrollo del concurso.

Además, el sistema de control debe poder distribuirse, permitiendo repartir las responsabilidades durante la ejecución del concurso.

De todo esto se extraen las siguientes funcionalidades y características:

- Posibilidad de ejecutar múltiples instancias, distribuyendo la responsabilidad de control.
- Mostrar la información sobre la pregunta actual.
- Marcar la opción elegida por los concursantes.
- Saltar una pregunta en caso de que se acabe el tiempo de respuesta.
- Proporcionar una interfaz simple e intuitiva.
- Actualizarse automáticamente en función del estado del concurso.
- Mostrar información sobre el estado del concurso.

Para el control distribuido del concurso se diseñó la aplicación Web de control (Fig. 5.19). Esta aplicación se instala en un servidor Web que no debe confundirse con el servidor *QuQuSI* descrito en la sección 5.3.

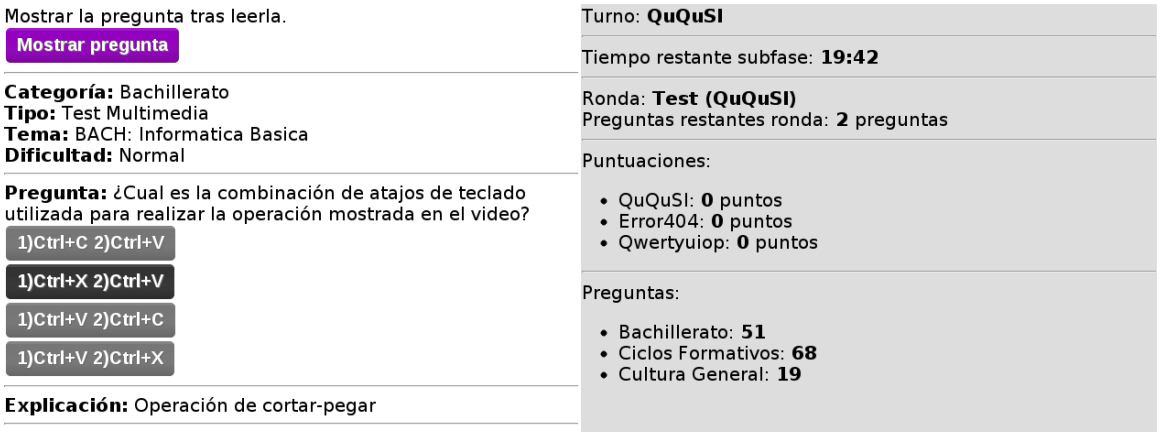

Figura 5.19: Captura de la interfaz de la aplicación Web de control.

Está aplicación, implementada en HTML, PHP, JavaScript y AJAX, permite enviar las señales de control necesarias para cambiar el estado del concurso. También muestra información durante el concurso como el tiempo o el número de preguntas restante de cada ronda. La apariencia de su interfaz se define mediante CSS.

La aplicación utiliza JavaScript, AJAX, PHP y *sockets* TCP para realizar un *polling* que refresca la página automáticamente en función del estado del concurso y la pregunta actual. También usa APC (capítulo 3) como caché para evitar que se sobrecargue el servidor *QuQuSI* con peticiones.

El funcionamiento de la aplicación es el siguiente (Fig. 5.20):

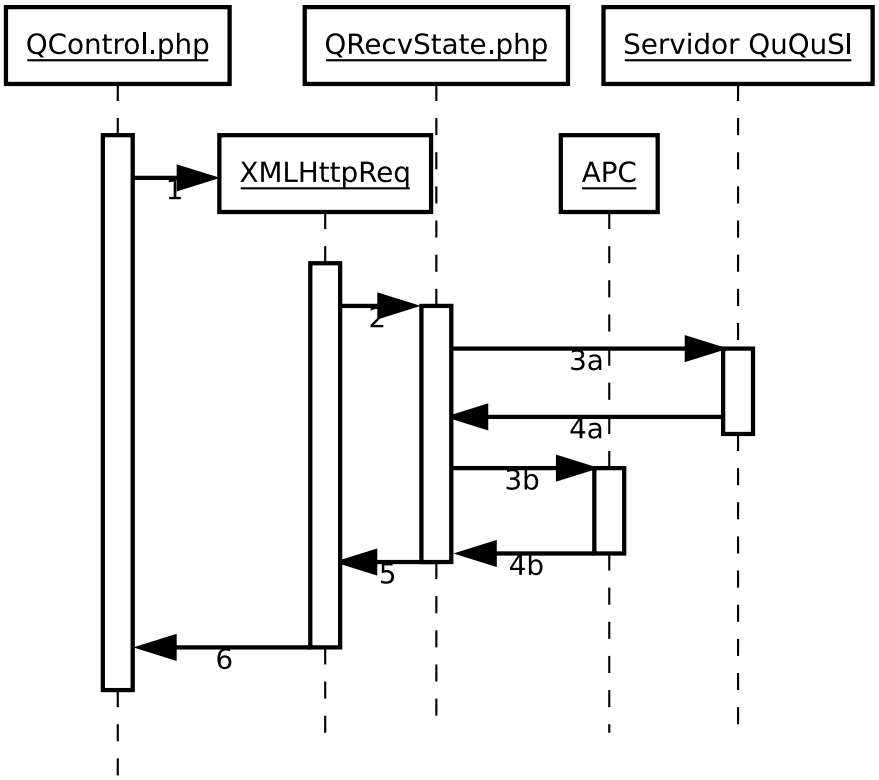

Figura 5.20: Diagrama de secuencia del funcionamiento de la aplicación Web de control.

1. Al cargar la página se llama a la función JavaScript que crea la petición AJAX (1).

2. Se ejecuta la petición AJAX al servidor Web sobre el *script* PHP que consulta el estado del concurso (2).

3. El *script* PHP utiliza un semáforo implementado con APC para evitar que se realicen demasiadas peticiones al servidor. Si el semáforo está cerrado, lee el estado desde

la caché de APC (3b, 4b). Si está abierto, lo cierra y realiza la consulta al servidor *QuQuSI* a través de un *socket* TCP (3a, 4a).

4. Una vez recibido el estado, el *script* devuelve un código HTML correspondiente con el estado y la pregunta actual (5).

5. Una vez se completa la petición AJAX, se muestra su contenido en una sección de la página Web y se llama de nuevo a la función JavaScript con un retardo de un segundo (6).

Al tratarse de una aplicación Web, se puede acceder a esta desde varios computadores, distribuyendo el control del concurso.

Gracias a la interfaz que ofrece, se puede visualizar la información sobre el concurso y seleccionar las opciones pertinentes en cada momento.

Al aplicar un CSS con un tema simple se consigue una interfaz fácil de utilizar.

El *polling* realizado mediante AJAX y JavaScript permite que la página se actualice en función del estado del concurso.

Gracias al uso de la caché de APC no se sobrecarga ni el servidor Web ni el servidor *QuQuSI*.

5.8. APLICACIÓN WEB PARA EL PÚBLICO

La gestión integral del concurso televisivo al que da soporte *QuQuSI* requiere que la plataforma contemple la participación del público.

La aplicación debe poder utilizarse desde el mayor número de dispositivos diferentes.

Necesita ofrecer un mecanismo de *login* que permita a cada persona identificarse de forma única. De este modo resulta posible conocer a los ganadores de forma inequívoca.

La interfaz debe presentar al público la misma pregunta a la que se enfrentan los concursantes, contando con un tiempo determinado para responderla.

También debe informar sobre el estado del concurso con mensajes que ayuden a seguir el desarrollo del mismo.

Al terminar el concurso, debe poderse saber quien ha ganado evitando posibles empates. Para ello, se ha de tener en cuenta el tiempo de respuesta además de las preguntas contestadas correctamente.

De ello se obtienen las siguientes características y funcionalidades:

- Ser heterogénea, pudiendo ejecutarse desde el mayor número de dispositivos diferentes posible.

- Permitir que el usuario realice un *login* que lo identifique de forma inequívoca.
- Ofrecer la opción de responder a las preguntas del concurso en tiempo real.
- Proporcionar un *feedback* sobre el estado del concurso que permita comprender el desarrollo del mismo.
- Calificar a los participantes en función de las preguntas acertadas y el tiempo de respuesta.

La aplicación Web para el público se diseñó para permitir la participación desde cualquier navegador Web (Fig. 5.21). La aplicación se instala en un servidor Web que no debe confundirse con el servidor *QuQuSI* descrito en la sección 5.3.

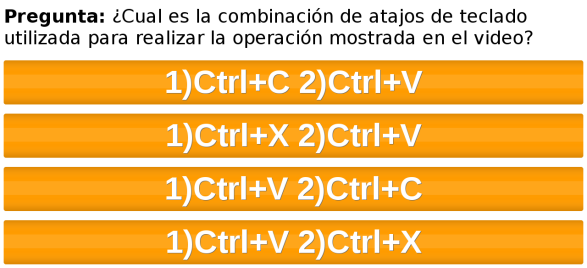

Figura 5.21: Captura de la interfaz de la aplicación Web para el público.

La aplicación está implementada en HTML, PHP, JavaScript y AJAX. Permite que los miembros del público se identifiquen y contesten a las preguntas al mismo tiempo que los concursantes. La apariencia de su interfaz está especificada mediante CSS.

El funcionamiento de la aplicación es prácticamente idéntico al de la aplicación Web de control descrita en la sección 5.7. También utiliza APC, compartiendo algunas variables de caché con la aplicación de control Web, disminuyendo aún más el tráfico entre el servidor Web y el servidor *QuQuSI*.

La principal diferencia es que requiere un *login*. Cuenta con otra aplicación Web de apoyo para introducir el nombre del público invitado. A cada invitado se le entrega un identificador único una vez queda registrado en el sistema. La lista de identificadores se obtiene mediante un programa que genera cadenas de caracteres aleatorios.

Como se mencionó en la sección 5.2, la base de datos se amplía con dos nuevas tablas utilizadas por esta aplicación (Fig. 5.2).

La tabla de jugadores cuenta con los siguientes campos:

- **Identificador:** Es el identificador único del jugador obtenido desde la aplicación de generación de cadenas de caracteres aleatorios.

- **Categoría:** La categoría a la que pertenece el participante. Hay un ganador por cada categoría.
- **Nombre:** El nombre completo del jugador.

Por su parte, la tabla de respuestas consta de los siguientes campos:

- **Jugador:** El identificador del jugador que da la respuesta.
- **Pregunta:** El identificador de la pregunta a la que se responde. Junto con el identificador del jugador forma la clave primaria de la tabla.
- **Respuesta:** La opción seleccionada por el jugador en la aplicación.
- **Respuesta correcta:** La opción correcta recuperada desde la tabla preguntas de la base de datos.
- **Tiempo de respuesta:** El tiempo, en milisegundos, empleado por el jugador en seleccionar la opción.

Su funcionamiento, como se mencionó anteriormente, es muy similar al de la aplicación Web de control descrita en la sección 5.7:

1. En primer lugar el usuario realiza el *login* a la aplicación mediante una cadena de caracteres personal que le es entregada antes del concurso.
2. Una vez identificado el usuario, se carga la página principal en donde se llama a la función JavaScript que crea la petición AJAX.
3. Se ejecuta la petición AJAX al servidor Web sobre el *script* PHP que consulta el estado del concurso.
4. El *script* PHP utiliza un semáforo implementado con APC para evitar que se realicen demasiadas peticiones al servidor. Si el semáforo está cerrado, lee el estado desde la caché de APC. Si está abierto, lo cierra y realiza la consulta al servidor *QuQuSI* a través de un *socket* TCP.
5. Una vez recibido el estado, el *script* devuelve un código HTML correspondiente con el estado y la pregunta actual.
6. Una vez se completa la petición AJAX, se muestra su contenido en una sección de la página Web y se llama de nuevo a la función JavaScript con un retardo de un segundo.

7. Cuando el usuario contesta a una pregunta, se marca ésta en la interfaz y se activa una variable que impide que se responda de nuevo. La respuesta dada se almacena en la base de datos.

Gracias a que se trata de una aplicación Web se puede ejecutar en el mayor número de dispositivos y plataformas, ya que tan solo requiere un navegador compatible con JavaScript para su ejecución.

Con la ampliación de la base de datos se consigue un mecanismo de *login* que permite identificar a los usuarios de forma unívoca.

Mediante la actualización de la interfaz con AJAX se logra que el público pueda contestar en tiempo real a las mismas preguntas que los concursantes.

Los mensajes mostrados durante los diferentes estados del concurso ayudan a seguir el desarrollo del mismo.

Incluyendo el campo tiempo a la tabla de respuestas se pueden resolver los empates en función del tiempo de respuesta.

5.9. REPRODUCTOR DE EVENTOS

La gestión integral de un concurso televisivo requiere que se de soporte a la fase de postproducción que precede a la grabación del concurso.

Durante esta fase puede ser interesante volver a reproducir ciertas partes de la salida gráfica y sonora del concurso. Por ejemplo, para corregir alguna toma o para obtener una salida con una mayor resolución de la disponible durante la grabación.

También puede resultar útil reproducir el historial de preguntas empleadas durante el concurso, de modo que el público pueda participar en diferido.

Así, surgen las siguientes características deseables:

- Poder cargar el historial de eventos ocurridos durante la grabación del concurso.
- Iniciar la reproducción a partir de un momento determinado, de forma que no haya que recorrer el historial completo cada vez.
- Enviar los mensajes pertinentes a la aplicación de *chroma* para obtener la misma salida gráfica y sonora que la obtenida durante la grabación del concurso.

El reproductor de eventos se encarga de satisfacer estas características.

Esta aplicación, implementada en C++, carga el historial de eventos y simula ser el servidor del concurso para que la aplicación de *chroma* repita la salida gráfica obtenida durante la grabación.

En el diseño del reproductor se emplearon los siguientes patrones:

- ***Observer:*** Se utiliza en el gestor de comunicación, realizando la lectura síncrona de *sockets*.
- ***Singleton:*** Se implementa en los gestores de recursos utilizados por la aplicación.

Los módulos que forman la aplicación son:

- **Gestor de configuración:** Se ocupa de cargar la configuración desde un archivo. Es el mismo módulo que el descrito en la sección 5.3.
- **Gestor de comunicación:** Es una versión simplificada del módulo descrito en la sección 5.3. Acepta las aplicaciones *chroma* clientes, a las que notifica de los eventos leídos en el historial.
- **Gestor de eventos:** Es el módulo superior de la aplicación. Carga el historial de eventos desde un archivo a una lista. Lleva un contador de tiempo que compara con el tiempo del primer evento de la lista; cuando coincide, lo envía a las aplicaciones cliente conectadas.

Gracias al módulo de eventos, se lee el historial desde un archivo y se carga en una lista ordenada en memoria.

Mediante un parámetro leído por el gestor de configuración se puede determinar el momento en el que comenzar a reproducir los eventos.

El módulo de comunicación se encarga de transmitir los eventos a las aplicaciones *chroma* como si se tratara del servidor del concurso.

CAPÍTULO 6. EVOLUCIÓN Y COSTES

En este capítulo se describe la evolución sufrida por el proyecto bajo el Proceso Unificado de Desarrollo, como se indicó en el capítulo 4. Al tratarse de una metodología iterativa, se detalla el trabajo realizado en cada iteración, especificando la fase en la que se da y las tareas llevadas a cabo en cada flujo de trabajo.

También se desglosan los costes asociados al desarrollo del proyecto y al despliegue de la plataforma durante la fase final de las VII Olimpiadas Informáticas.

6.1. EVOLUCIÓN DEL PROYECTO

Previamente al comienzo del desarrollo de *QuQuSI* se realizaron una serie de reuniones con los responsables del material de grabación del Centro de Cálculo y de la Escuela Superior de Informática de Ciudad Real. Se decidió entonces utilizar exclusivamente los equipos de la Escuela Superior de Informática por su mayor disponibilidad.

Tras elaborar un inventario del material (Véase Anexo C), se realizó el primer boceto de un sistema distribuido que aprovechara los recursos disponibles.

Al mismo tiempo se fue concibiendo la mecánica de juego del concurso, determinando el número de participantes, la duración y el tipo de pruebas (Véase Anexo B).

A partir del boceto del sistema y la mecánica de juego se determinaron los objetivos del presente Proyecto Fin de Carrera descritos en el capítulo 2.

Se realizó entonces un estudio de los conceptos y tecnologías necesarias para el desarrollo de la plataforma, cuyo resultado puede consultarse en el capítulo 3.

El proyecto se ha llevado a cabo siguiendo el Proceso Unificado de Desarrollo. El resultado obtenido se describe en el capítulo 5. En las siguientes secciones se detalla cada iteración del proceso, desglosando las tareas realizadas en cada flujo de trabajo (Tabla 6.1).

6.1.1. Inicio

La fase de inicio sirvió para poner en marcha el proyecto, estableciendo los objetivos y el calendario de entregas.

Iteración	Fecha de entrega
1	22-12-2012
2	10-1-2013
3	23-2-2013
4	30-3-2013
5	10-4-2013
6	15-4-2013
7	20-4-2013
8	26-4-2013
9	30-4-2013
10	13-5-2013
11	23-5-2013
12	29-5-2013

Tabla 6.1: Fechas de entrega según iteración.

Se realizó un primer análisis de requisitos que la plataforma debería soportar y se creó la aplicación Web para la gestión de preguntas.

Se llevaron a cabo tres iteraciones durante la fase de inicio.

Iteración 1

En la primera iteración los esfuerzos se dirigieron a la captura de requisitos y a la elaboración de esquemas generales del sistema.

- **Requisitos:** A través de reuniones con el personal implicado y del estudio del material disponible se determinaron los principales requisitos funcionales que la plataforma debería soportar de forma general.

- **Análisis:** En base a los requisitos descubiertos y al material disponibles se planteó un primer esquema del sistema en el que se identificaron sus diferentes aplicaciones.

Como resultado de la iteración se obtuvieron una lista con los principales requisitos funcionales de la plataforma y un esquema general del sistema en el que se identificaron sus diferentes aplicaciones.

Iteración 2

La segunda iteración se centró en la implementación de la aplicación Web para gestionar las preguntas. Esta aplicación comenzó a utilizarse por parte de los profesores y personal

implicado en las VII Olimpiadas Informáticas para introducir las preguntas del concurso en la base de datos.

- **Requisitos:** Se especificaron los requisitos del *workflow* para la generación de las preguntas.
- **Análisis:** A partir de los requisitos del *workflow* se determinaron las funcionalidades de la aplicación Web para las preguntas y se elaboraron sus diagramas de casos de uso.
- **Diseño:** Se continuó con el diseño de la arquitectura cliente servidor. Se diseñó el módulo de gestión de preguntas mediante diagramas de clases.
- **Implementación:** Se generó el código empleando el *framework* CakePHP y se realizaron las modificaciones necesarias para satisfacer las necesidades de *login*, seguridad y gestión de archivos multimedia.
- **Pruebas:** Se probaron las funcionalidades de la aplicación Web para las preguntas.

El resultado de esta iteración fue la aplicación Web para las preguntas.

Iteración 3

En la tercera iteración se continuó con la captura de requisitos, se elaboraron los primeros diagramas de casos de uso y se desarrolló un prototipo de la interfaz gráfica para la aplicación de la técnica de *chroma key*.

- **Requisitos:** Se repartieron los requisitos generales entre las aplicaciones del sistema y se descubrieron nuevos requisitos particulares de cada una.
- **Análisis:** A partir de los requisitos particulares se elaboraron los diagramas de casos de uso de cada aplicación. Se analizó cómo llevar a cabo el despliegue de contenidos multimedia.
- **Diseño:** Se continuó con el diseño de la arquitectura cliente servidor. Se diseñó el módulo de gestión de preguntas mediante diagramas de clases. Se diseñó el *widget* para el despliegue de contenidos multimedia utilizando el *framework* GStreamer.
- **Implementación:** Se creó un prototipo de la interfaz gráfica con una funcionalidad limitada, capaz de reproducir contenidos multimedia. Se implementó también una primera versión del módulo de gestión de preguntas.

- **Pruebas:** Se realizaron pruebas unitarias para el módulo de gestión de preguntas implementado.

El resultado de esta iteración fueron los diagramas de casos de uso de cada aplicación, el prototipo de la interfaz gráfica y la primera versión del gestor de preguntas.

6.1.2. Elaboración

Con los requisitos generales identificados, la aplicación Web implementada y el prototipo de la interfaz gráfica se llevo a cabo una reunión con los usuarios para mostrarles los objetivos del proyecto y el funcionamiento de la aplicación Web.

Se pasó entonces a la fase de elaboración, en la que se centraron los esfuerzos en terminar el diseño de la arquitectura y en identificar y diseñar los módulos de las aplicaciones.

En esta fase tuvieron lugar dos iteraciones.

Iteración 4

En la cuarta iteración se diseñó la capa de comunicación de la arquitectura y se comenzó la implementación del servidor *QuQuSI*.

- **Requisitos:** Se analizaron los nuevos requisitos detectados en la capa de comunicación de la arquitectura.
- **Análisis:** Se analizaron las necesidades de comunicación entre las aplicaciones del sistema y se determinó la información requerida por cada una.
- **Diseño:** Se elaboraron los diagramas de clase del módulo de comunicación para el servidor *QuQuSI*.
- **Implementación:** Se implementó la base del módulo de comunicación, formado por la clase de *sockets* y el patrón *observer* junto con la función *select*.

El resultado de la iteración es una versión funcional de la capa de comunicaciones del sistema.

Iteración 5

En la quinta iteración se diseñaron los módulos de configuración y concurso y se implementaron los patrones *singleton*, *state* y *observer* mediante clases plantilla.

- **Análisis:** Se llevó a cabo el análisis de la mecánica de juego para incorporar las reglas al módulo de concurso.
- **Diseño:** Se identificaron las estructuras de datos útiles para la implementación de los módulos y se elaboraron los diagramas de clases.
- **Implementación:** Se implementaron los patrones *singleton*, *state* y *observer* mediante clases plantilla. Se implementó el módulo de configuración y una versión básica del módulo de concurso que incluía la máquina de estados.
- **Pruebas:** Se realizaron pruebas unitarias en el módulo de configuración y de integración al unirlo con el módulo de comunicaciones en el servidor *QuQuSI*.

De esta iteración se obtuvo el módulo de configuración completo y una versión con funcionalidad básica del servidor *QuQuSI*.

6.1.3. Construcción

Con los requisitos generales asociados a cada aplicación del sistema, la aplicación Web para las preguntas en fase de explotación y la implementación dela arquitectura en un estado avanzado se pasó a la fase de construcción.

En esta fase se implementaron en sucesivas iteraciones las diferentes aplicaciones del sistema mientras se realizaban los cambios necesarios en el servidor para soportar las nuevas funcionalidades añadidas.

Durante esta fase se dieron seis iteraciones.

Iteración 6

Durante la sexta iteración se implementó la aplicación para los pulsadores y se escribió el código para el microcontrolador Arduino. Se amplió también el módulo de comunicación del servidor para soportar la nueva funcionalidad.

- **Análisis:** Se analizó el mecanismo de comunicación entre aplicaciones implementadas en C++ a través de *sockets* y cómo conectarse a través del puerto serie con el microcontrolador Arduino.
- **Diseño:** Se diseñó el mecanismo de comunicación de paso de mensajes entre aplicaciones implementadas en C++ a través de *sockets*. Se creó para ello una versión simplificada del módulo de comunicación del servidor, reutilizando así el código en

las aplicaciones clientes. Se diseñó también una forma de comunicación con el puerto serie a través de un archivo de dispositivo con las propiedades de un terminal virtual y el circuito electrónico de los pulsadores que se conecta a la placa Arduino.

- **Implementación:** Se implementó la aplicación para los pulsadores reutilizando código del módulo de comunicación del servidor *QuQuSI*, la aplicación para detectar pulsaciones para el microcontrolador y se amplió el código del servidor *QuQuSI* para soportar la nueva funcionalidad.

- **Pruebas:** Se llevaron a cabo pruebas unitarias y de integración durante la implementación de la aplicación para los pulsadores. Se realizaron pruebas de sistema para probarla junto con el servidor.

Al acabar la iteración se tenía una versión funcional de la aplicación para los pulsadores, el diseño del circuito electrónico y una versión ampliada del servidor *QuQuSI*.

Iteración 7

En la séptima iteración se implementaron las aplicaciones Web de control y participación del público. Se ampliaron los módulos de comunicación y concurso del servidor *QuQuSI*.

- **Análisis:** Se analizó el mecanismo de comunicación entre las aplicaciones Web y el servidor *QuQuSI*.

- **Diseño:** Se diseñó el mecanismo de comunicación basado en *polling* entre las aplicaciones Web y el servidor *QuQuSI* y se amplió el diagrama de clases del servidor *QuQuSI* con el módulo de concurso y el módulo de preguntas.

- **Implementación:** Se implementaron las aplicaciones Web de control y de participación del público y se amplió la funcionalidad del módulo de concurso mejorando la máquina de estados e incluyendo elementos de la mecánica de juego.

- **Pruebas:** Se llevaron a cabo pruebas de sistema para probar la comunicación entre las aplicaciones Web y el servidor *QuQuSI*.

El resultado de la iteración son las primeras versiones de la aplicación Web de control y de participación del público y un servidor *QuQuSI* con una funcionalidad más avanzada.

Iteración 8

Durante la octava iteración se implementó la aplicación de marcadores, ampliando el módulo de comunicación del servidor para soportar la nueva funcionalidad.

- **Diseño:** Se diseñó la aplicación de marcadores teniendo en cuenta los *frameworks* OGRE 3D y OIS.
- **Implementación:** Se implementó la aplicación de marcadores y se amplió el servidor para que el módulo de comunicaciones soportara la nueva funcionalidad.
- **Pruebas:** Se realizaron pruebas unitarias y de integración durante la implementación de la aplicación de marcadores. Posteriormente se llevaron a cabo pruebas de sistema con todas las aplicaciones implementadas hasta el momento.

Al finalizar esta iteración se contaba con una versión completamente funcional de la aplicación de marcadores y un servidor ampliado capaz de comunicarse con ella.

Iteración 9

La novena iteración se centró en completar el módulo de concurso, ampliando la máquina de estados, incluyendo la mecánica de juego e incorporando el gestor de preguntas implementado junto con el prototipo de la interfaz gráfica.

- **Implementación:** Se adaptó e integró el gestor de preguntas en el código del servidor *QuQuSI*, se implementaron las mecánicas de juego restantes y se dotó al módulo de concurso con capacidad de configuración externa. También se reorganizó la máquina de estados.
- **Pruebas:** Se realizaron pruebas de integración al incorporar el gestor de preguntas previamente implementado al servidor *QuQuSI*.

Iteración 10

La décima iteración cerró la fase de construcción con la implementación de la aplicación para el *chroma*. Se reutilizó parte del código del prototipo de la interfaz gráfica.

- **Diseño:** Se diseñó la aplicación reutilizando el código del prototipo de la interfaz gráfica, lo que constituyó el gestor de *overlays*. Se reutilizó además parte del diseño de la aplicación de marcadores. También se diseñó el módulo de gestión de audio utilizando GStreamer.

- **Implementación:** Se llevó a cabo la implementación de la aplicación para el *chroma* reutilizando la mayor cantidad de código posible y se amplió el servidor *QuQuSI* para soportar la nueva funcionalidad.
- **Pruebas:** Se realizaron pruebas unitarias y de integración durante la implementación de la aplicación para el *chroma*. Una vez en funcionamiento, se llevaron a cabo pruebas de sistema con el resto de aplicaciones del sistema.

Al finalizar la iteración, se tenía una primera versión beta, completamente funcional, del sistema aún sin refinar.

6.1.4. Transición

Con todas las aplicaciones implementadas y el sistema funcionando en versión beta se pasó a la fase de transición.

En esta fase se realizaron las pruebas funcionales, las cuales permitieron detectar y corregir errores e incorporar nuevas funcionalidades sugeridas por los usuarios. Por último, se planificó y realizó el despliegue de la plataforma.

Iteración 11

En la undécima iteración se rediseñaron algunas de las aplicaciones del sistema, refactorizando código e incorporando nuevas funcionalidades simples sugeridas por los usuarios.

- **Implementación:** Se mejoraron las aplicaciones Web de control y de participación del público, se incorporaron eventos temporales tanto al servidor *QuQuSI* como a la aplicación para el *chroma*, permitiendo el uso de cuentas atrás y temporizadores en el concurso y se modificó la mecánica de juego, implementando una ronda final.
- **Pruebas:** Se realizaron pruebas funcionales con el sistema completo, gracias a las cuales se detectaron algunos errores. Las sesiones fueron grabadas y están disponibles en el DVD adjunto a la documentación.

Tras esta iteración se cuenta con la primera versión de la plataforma completa.

Iteración 12

La última iteración se dio tras la grabación del concurso y sirvió para implementar el reproductor de eventos que se utilizaría en el montaje del vídeo final durante la postproducción.

- **Diseño:** Se diseño la aplicación para la reproducción del historial de eventos, teniendo en cuenta las estructuras de datos necesarias.
- **Implementación:** Se implementó el reproductor de eventos reutilizando parte del código del servidor *QuQuSI*.

Tras la última iteración, se tiene la primera versión del sistema y el reproductor de eventos que se utilizó durante la fase de postproducción.

6.2. RECURSOS Y COSTES

En esta sección se desglosan los costes de desarrollo y despliegue del proyecto. Gracias a la utilización del material disponible en las instalaciones de la Escuela Superior de Informática la inversión realizada fue mínima.

6.2.1. Coste del desarrollo

El desarrollo de *QuQuSI* abarca desde septiembre de 2012 hasta mayo de 2013. Hasta febrero de 2013 se trabajó en el proyecto a media jornada cinco días a la semana, mientras que desde marzo hasta mayo se trabajó a jornada completa seis días a la semana. Se estiman un total de 1144 horas de trabajo. Para estimar el coste del salario del desarrollador de software se considera un sueldo de 30 €[1] brutos por hora de trabajo. Los efectos de sonido y la música empleada tienen licencias libres. Los recursos que no están marcados con $*$ estaban disponibles antes de comenzar el desarrollo. En la (Tabla 6.2) se desglosan los costes del desarrollo del proyecto.

Recurso	**Cantidad**	**Coste**
$*$ Sueldo desarrollador de software	1	34.320 €
Acer Aspire 5920G	1	1.100 €
Samsung Galaxy S+	1	300 €
Total		**35.720 €**
$*$ **Total**		**34.320 €**

Tabla 6.2: Coste estimado del desarrollo de *QuQuSI*.

6.2.2. Coste del despliegue

El despliegue de *QuQuSI* para la fase final de las VII Olimpiadas Informáticas duró dos semanas. En el montaje participaron a media jornada dos miembros de la unidad técnica de

[1] http://www.infojobs.net/

la Escuela Superior de Informática de Ciudad Real. Para simplificar los cálculos, no se ha tenido en cuenta el salario del personal técnico de la ESI. Se compró una cámara de vídeo adicional y se contrató una empresa para la impresión de los atriles. Los recursos que no están marcados con ∗ estaban disponibles antes de comenzar el desarrollo. En la (Tabla 6.3) se desglosan los costes del despliegue del proyecto.

El concurso tuvo lugar el día 24 de mayo de 2013 y contó con la asistencia de más de un centenar de personas.

Recurso	**Cantidad**	**Coste**
Sobremesa *Intel Core 2 Duo*	4	300 €
Acer Aspire 5920G	1	1.100 €
Portatil HP táctil	1	800 €
Servidor compartido	1	2.000 €
Cámara de control remoto	2	300 €
∗ Cámara de control remoto	1	300 €
Control remóto	1	400 €
Mesa de mezcla de vídeo	1	1.000 €
Panel de cuatro pantallas	1	800 €
Monitor	6	70 €
Proyector	3	300 €
Grabador DVD	1	40 €
Grabador HDD	1	90 €
Micrófonos con soporte	3	60 €
Micrófono de diadema	1	60 €
Mesa de mezcla de sonido	1	300 €
Amplificador	1	200 €
Altavoces	2	120 €
Microcontrolador *Arduino*	1	40 €
Pulsadores artesanales	3	5 €
Puntos de acceso Wifi	2	150 €
Conversor de señal	4	30 €
∗ Decoración	1	600 €
Total		**11.105 €**
∗ **Total**		**900 €**

Tabla 6.3: Coste estimado del despliegue de *QuQuSI*.

6.2.3. Estadísticas del repositorio

Para el control de versiones se ha empleado un servidor *Mercurial* alojado en *Bitbucket*.

El número de líneas de código de cada lenguaje empleado en la elaboración de *QuQuSI* puede consultarse en la (Tabla 6.4).

Lenguaje	**Archivos**	**Vacías**	**Comentarios**	**Código**
C++	50	1132	995	3865
Cabeceras C/C++	76	1387	461	3406
PHP	39	458	1078	1853
CSS	4	29	34	801
SQL	6	61	112	621
make	5	105	35	143
Total:	**180**	**3172**	**2715**	**10689**

Tabla 6.4: Lineas de código de *QuQuSI* contabilizadas con la aplicación *cloc*.

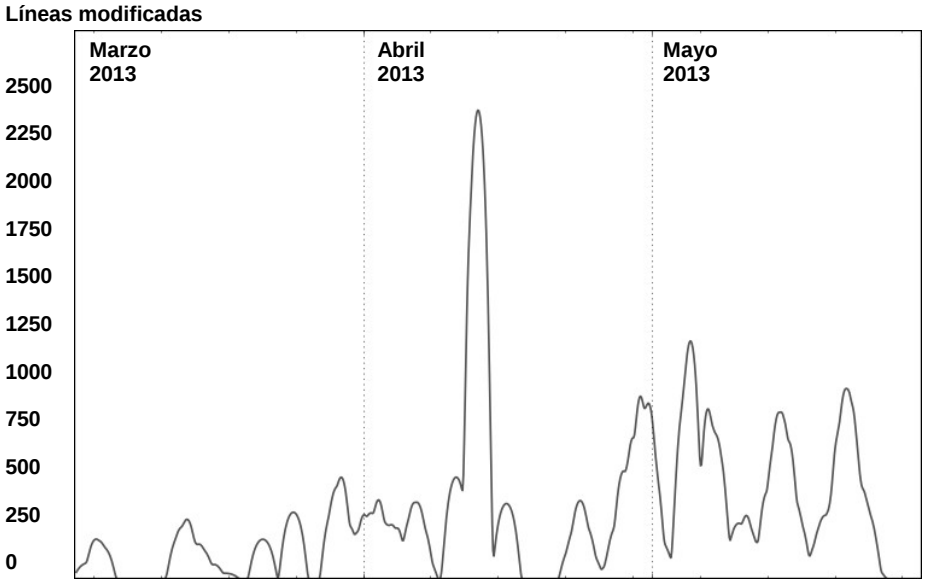

Figura 6.1: Lineas de código modificadas por fecha obtenida con la extensión *activity* de *Mercurial*.

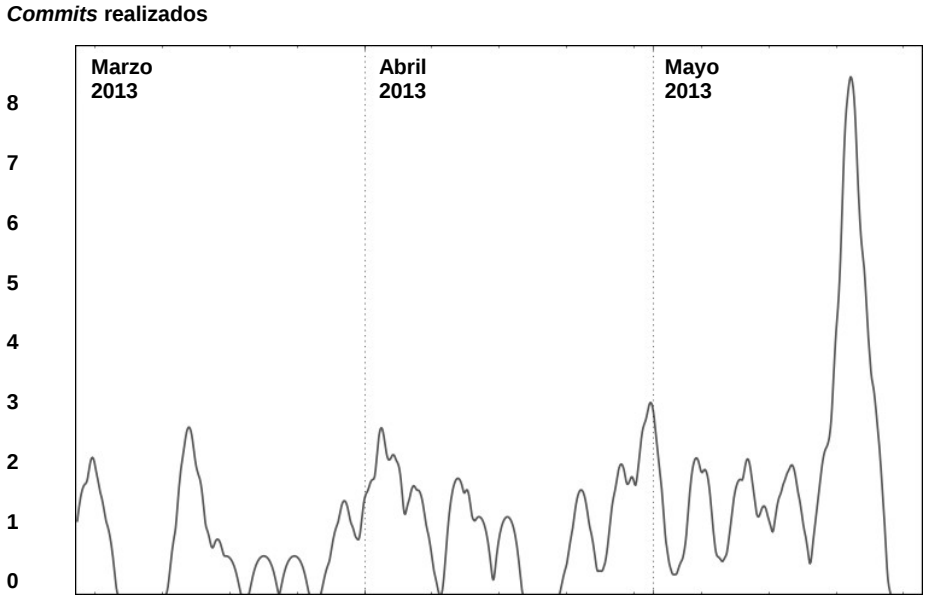

Figura 6.2: *Commits* realizados al servidor *Mercurial* por fecha obtenidos con la extensión *activity*.

CAPÍTULO 7. CONCLUSIONES Y PROPUESTAS

En este capítulo se describen los objetivos alcanzados y se proponen una serie de líneas de trabajo futuro para mejorar y extender la plataforma.

7.1. OBJETIVOS ALCANZADOS

Se han cumplido todos los objetivos descritos en el capítulo 2. *QuQuSI* es una plataforma para la gestión integral de un concurso televisivo que permite la participación del público.

El objetivo general de gestionar de forma integral un concurso televisivo permitiendo la participación del público se ha satisfecho completamente, como queda demostrado tras la utilización de la plataforma en la fase final de las VII Olimpiadas Informáticas. El resultado puede verse en el DVD adjunto a la documentación. El concurso contó con la asistencia de más de un centenar de personas. Se realizaron dos grabaciones; la primera es la que se proyectó en el salón de actos en directo y la segunda se produjo en la fase de postproducción mezclando el sonido grabado en directo con las imágenes obtenidas por cámaras de alta definición.

Los objetivos específicos se han cumplido igualmente. El despliegue gráfico multimedia 2D y 3D y la reproducción de sonido se han logrado mediante el uso de los *frameworks OGRE 3D* y *GStreamer*. La plataforma permite que se muestren imágenes, vídeos y texto y se reproduzcan efectos de sonido y música.

La integración con los sistemas existentes se ha conseguido al utilizar el material disponible en la Escuela Superior de Informática. La inversión realizada ha sido mínima ya que se han empleado los equipos informáticos y el material de grabación del salón de actos (Tabla 6.3). Como se describió en el capítulo 6, se realizó un análisis previo de la infraestructura de grabación disponible a partir del cual se identificaron las diferentes aplicaciones que permitieran sacarle el mayor partido.

La gestión de preguntas multimedia se logró con el desarrollo de la aplicación Web que habilita el *workflow* que resulta en el conjunto de preguntas validadas utilizadas en el concurso. La aplicación Web fue utilizada por once profesores que elaboraron las preguntas y

tres validadores que las revisaron. Se generaron un total de 192 preguntas, de las cuales 139 fueron validadas.

La plataforma permite la participación del público a través de sus dispositivos móviles. Gracias a haber elegido una interfaz Web se puede participar desde cualquier dispositivo con un navegador compatible con Javascript. Durante el concurso se registraron 1772 respuestas dadas por 57 miembros del público.

La aplicación cuenta con múltiples interfaces de control gracias a que se pueden ejecutar varias instancias de la aplicación Web de control en diferentes equipos. De este modo, durante la grabación del concurso, tanto el presentador como el encargado de la realización emplearon la misma interfaz ejecutándose en diferentes máquinas.

La robustez del sistema se logró a través de las pruebas realizadas durante el desarrollo. La recuperación de errores es posible gracias a la capacidad de reconexión de las aplicaciones clientes. Justo antes del comienzo del concurso, un error humano causó que el nombre de uno de los equipos fuese incorrecto. El servidor se reinició con el nombre correcto y el resto de aplicaciones se reconectaron sin más incidentes.

La arquitectura cliente servidor y el uso de interfaces Web dotan a la plataforma de la heterogeneidad necesaria para poder ejecutarse en diferentes máquinas. El sistema desarrollado cuenta con cinco aplicaciones escritas en C++ que pueden ser compiladas para sistemas operativos *GNU/Linux* o *Windows* y tres aplicaciones Web cuyo despliegue tan solo requiere un servidor Web con PHP.

El uso de patrones de diseño, herencia y plantillas confieren a la plataforma una arquitectura flexible y extensible. Todas las aplicaciones pueden configurarse desde archivo, permitiendo que se pueda personalizar sin que se tenga que compilar de nuevo.

La plataforma se desarrolló utilizando estándares y tecnologías libres. En el capítulo 4 se enumeran los lenguajes y herramientas empleados durante su elaboración, todos ellos con licencias libres.

7.2. TRABAJO FUTURO

Dados los exigentes requisitos de calendario en el desarrollo de la versión final del proyecto, hubo que tomar ciertas decisiones para cumplir los plazos con la mejor calidad y garantizando la robustez de la solución. Esto se ve reflejado en la disminución de ciertas características de generalización y personalización que, aunque deseables, no pudieron ser implementadas en la versión final. Por igual motivo, en algunos aspectos se eligió un diseño mas sencillo y robusto frente a otros más elegantes para garantizar la funcionalidad requerida en tiempo y forma.

A continuación se enumeran las líneas de trabajo futuro que permitirían mejorar y extender la plataforma:

- **Aplicar procesadores de lenguajes:** Los analizadores sintácticos y semánticos son herramientas muy útiles con multitud de campos de aplicación.

  En *QuQuSI* la lectura de archivos de configuración y preguntas se realizó de la forma más sencilla, mediante la lectura palabra a palabra. Esta solución, aunque válida, cuenta con ciertas limitaciones. El módulo de configuración implementado únicamente permite que se tengan pares clave valor y cuenta con una detección de errores simple.

  Mediante el uso de analizadores sintácticos y semánticos se podrían añadir funcionalidades a varias partes de la plataforma. Por ejemplo, el módulo de configuración podría cargar listas de valores u otras estructuras de datos. La mecánica de juego podría describirse mediante un lenguaje de especificación, abstrayéndola del código. La carga del archivo con las preguntas exportadas también podría llevarse a cabo mediante un analizador semántico, lo que permitiría que se pudieran editar las preguntas fuera del sistema para la gestión de preguntas multimedia.

  En concreto, habría que modificar los módulos de configuración, preguntas y concurso descritos en las secciones 5.3.1, 5.3.4 y 5.3.3 respectivamente. Como los módulos se reutilizan entre aplicaciones, solo habría que realizar las modificaciones una vez y propagar los cambios. Mediante el uso de un generador de analizadores léxicos como *Flex* [19] se diseñarían los lenguajes de especificación para los archivos de configuración, las preguntas exportadas desde la aplicación Web y un lenguaje que permitiera personalizar la mecánica de juego del concurso.

  Se estima que la modificación de los módulos de configuración y preguntas se podría llevar a cabo en unas dos semanas, mientras que la modificación del módulo de concurso y el diseño de un lenguaje de especificación para la mecánica de juego del concurso tendría un coste de más de un mes, ya que requeriría que se refactorizase buena parte del núcleo del servidor *QuQuSI*.

- **Rediseñar la interfaz gráfica:** Para el despliegue gráfico se utilizó *OGRE 3D*. Este *framework* permite la visualización de modelos de mallas tridimensionales.

  En *QuQuSI* esta característica se utilizó exclusivamente en el plano en el que se reproducen los contenidos multimedia (imágenes y vídeos), prefiriendo el uso de *overlays* para mostrar el contenido de las preguntas. Está solución resultó adecuada y permitió lograr un aspecto visual elaborado.

Sería interesante aprovechar la capacidad de visualizar y animar modelos 3D para diseñar una interfaz tridimensional más atractiva. Los diferentes paneles en los que se muestra el contenido de las preguntas podrían sustituirse por elementos tridimensionales animados. El texto podría mostrarse sobre estos elementos mediante la técnica de *rendering* a textura. Se podrían aprovechar también los sistemas de particulas, la iluminación y las sombras para crear efectos más complejos a modo de *feedback* para los participantes.

El rediseño de la interfaz gráfica supondría realizar cambios en la aplicación para el *chroma*, en concreto en el módulo de *overlays* descrito en la sección 5.4.4. Tras diseñar los elementos tridimensionales que formarían parte de la interfaz, habría que sustituir los *overlays* por nodos de escena que incluyeran estos elementos. Respecto al texto, se podría crear un módulo que permitiera mostrar texto *TrueType* sobre una textura o bien seguir utilizando el texto bidimensional y situarlo justo sobre los elementos 3D de la interfaz en el momento preciso.

El diseño y elaboración de una nueva interfaz 3D, su integración con la aplicación para el *chroma* y la creación de un módulo para la gestión de texto tridimensional tendría una duración estimada de un mes y medio.

- **Mejorar la robustez del sistema:** Se garantizó el funcionamiento del sistema a través de diferentes técnicas de pruebas de software.

  Se garantizó además que el sistema pudiera recuperarse de problemas en la red dotando de capacidad de reconexión a las aplicaciones clientes. Sin embargo, esta medida pudiera ser insuficiente en caso de producirse una situación crítica que comprometiera la ejecución del servidor *QuQuSI*.

  Por cuestiones temporales se aplazó la implementacion de un módulo de serialización o *backup* capaz de guardar y cargar el estado del concurso. De este modo, el sistema se podría recuperar de un fallo crítico, como una perdida de corriente o un fallo irrecuperable durante la ejecución. La implementación del módulo mejoraría la robustez de la plataforma.

  Mejorar la robustez del sistema supondría la creación de un nuevo módulo de serialización o *backup*. Este nuevo módulo conocería el estado del módulo de concurso descrito en la sección 5.3.3. Sus funcionalidades principales serían guardar el estado del concurso en memoria no volatil y ofrecer la posibilidad cargar dicho estado al lanzar el servidor *QuQuSI*. El estado del concurso podría almacenarse mediante una

biblioteca de serialización como *Boost* [20] o bien mediante la creación de un lenguaje de especificación que fuese interpretado por un analizador como *Bison* [18].

El tiempo que llevaría la creación de un módulo de estas características se estima entre dos y tres semanas, ya que podría requerir que se realizaran cambios en el complejo módulo de concurso.

- **Aumentar la capacidad de personalización:** Parte de la mecánica de juego se implementó en el código fuente del servidor *QuQuSI*.

  La máquina de estados del concurso se implementó siguiendo el patrón *state*. El diseño permite que se configuren algunos parámetros desde un archivo de configuración externo, como el número de rondas, el tiempo o la puntuación de cada tipo de pregunta. Esto permite suficiente libertad para adaptar el concurso a bastantes situaciones. Sin embargo, esta solución no permite realizar grandes cambios en la mecánica de juego.

  Externalizar la mecánica de juego del concurso y otros factores, como las categorías y los temas dotaría de una mayor capacidad de personalización a la plataforma. Como se comentó anteriormente, mediante el uso de analizadores sintácticos y semánticos se podría determinar la mecánica de juego mediante un lenguaje de especificación. El resto de factores, como las categorías y los temas de las preguntas, deberían cargarse también desde archivos externos, de forma que no tuviera que realizar modificaciones en el código.

  Para llevar a cabo estos cambios se habría de modificar el módulo de concurso descrito en la sección 5.3.3. Se tendrían que modificar la mayor parte de sus clases, volviendolas más flexibles y genéricas. Para llevar a cabo esta modificación, se habría de llevar primero a cabo la aplicación de procesadores de lenguajes, de modo que se pudiera definir la mecánica de juego del concurso mediante un lenguaje de especificación.

  Llevar a cabo este cambio supondría primero realizar las tareas para la aplicación de procesadores de lenguajes a la plataforma, por lo que su duración se estima en un mes más otro mes para cambiar la máquina de estados del concurso.

- **Especializar la aplicación Web del público:** Se logró que la plataforma fuera heterogénea gracias al uso de interfaces Web.

  Para lograr que se pudiera jugar con el mayor número de dispositivos diferentes posibles, se tomó la decisión de utilizar una interfaz Web. De este modo, para jugar en directo al concurso tan solo se requiere un dispositivo móvil con un navegador Web

compatible con Javascript. Esta solución benefició la heterogeneidad de la plataforma, pero limitó la usabilidad de la aplicación. En la mayoría de casos los dispositivos móviles apagan la pantalla e incluso desconectan las interfaces de red después de un tiempo de inactividad. Esta condición no se puede controlar desde una aplicación Web.

Mediante la implementación diferentes versiones de la aplicación, cada una específica para un sistema operativo móvil, se conseguiría solventar este inconveniente. El coste de desarrollar estas aplicaciones se vería compensado con la ganancia en usabilidad. Esta conversión podría resultar trivial en algunos casos, ya que existen *frameworks* y Webs para la conversión automática de páginas Web en aplicaciones móviles (como la clase *WebView* del API de *Android* o la Web *DroidYourSite* [21]).

Este cambio supondría la creación de al menos tres versiones diferentes de la aplicación Web para el público descrita en la sección 5.8, una para cada uno de los sistemas operativos móviles más populares (*Android*, *iOS* y *Windows Phone*). Se requeriría un análisis previo de los *frameworks* de desarrollo de cada uno de los sistemas y, una vez seleccionados los más adecuados, se llevaría a cabo la implementación.

El coste temporal del análisis de los diferentes *frameworks* de desarrollo de aplicaciones móviles y la implementación de las diferentes versiones de la aplicación sería de dos meses aproximadamente.

7.3. CONCLUSIÓN PERSONAL

La elaboración del presente Proyecto Fin de Carrera me ha permitido aplicar la mayor parte de los conocimientos aprendidos durante mis años de estudio en la Escuela Superior de Informática a un caso práctico cercano a la realidad laboral a la que me enfrentaré en el futuro.

La realización de una plataforma para la gestión integral de concursos televisivos me ha planteado un gran número de desafios de diferente naturaleza, los cuales he podido superar gracias a la versatilidad de la Ingeniería Informática. La variedad de técnicas y herramientas que ofrece me han permitido analizar los problemas y escoger de entre las múltiples soluciones aquellas más adecuadas en cada caso.

La ingeniería del software, las bases de datos, las redes de computadores, la informática gráfica, etc. me han demostrado su utilidad y su potencial, ayudandome a ganar confianza en mis capacidades como analista y desarrollador de software. Aunque otros campos, como la arquitectura de computadores o la inteligencia artificial, han quedado en un segundo plano, también me han ayudado mejorando mi forma de trabajar y pensar.

En definitiva, estoy satisfecho de haber elegido la titulación de Ingeniería Informática como carrera universitaria, y espero que en los próximos años pueda seguir aprovechando y expandiendo mis conocimientos.

APÉNDICE A. MANUAL DE USUARIO

En este apéndice se incluyen las instrucciones a seguir para la instalación, despliegue y maniobra de *QuQuSI*.

A.1. INSTALACIÓN

La instalación de la plataforma requiere la compilación de código y el despliegue de aplicaciones Web, además de la configuración de la red para proporcionar un mayor nivel de seguridad y aislamiento, aunque esto no es estrictamente necesario. b Aunque este manual considera que la instalación se realiza sobre el sistema operativo *GNU/Linux* con una distribución *Debian* versión 7.0 o superior, como las bibliotecas empleadas en el desarrollo de *QuQuSI* son libres y multiplataforma deberían poder compilarse e instalarse igualmente en sistemas *Microsoft Windows*.

A.1.1. Aplicación Web para las preguntas

La instalación de la aplicación Web para las preguntas requiere un servidor Web con PHP (se recomienda Apache) y el módulo `mod_probe` habilitado y *MySQL*.

Primero se debe importar el archivo `olimpiadas_informatica_full.sql` con *MySQL*, el cual que genera la base de datos.

```
bd: olimpiadas\_informatica
user: oinformatica
password: 01nf0rm4t1c4
```

A continuación, se debe publicar la carpeta oinformatica contenida en el directorio base de *CakePHP* con el servidor Web.

Se podrá entonces ingresar al sistema de preguntas con la cuenta del administrador.

```
user: administrador
password: 01nf0rm4t1c4
```

La contraseña puede cambiarse una vez se acceda al sistema.

A.1.2. Aplicación Web de control y para el público

Estas dos aplicaciones requieren un servidor Web con PHP y Javascript y el módulo APC instalado mediante los siguientes comandos:

```
aptitude install php-pear
aptitude install php5-dev
aptitude install apache2-dev
pecl install apc
echo "extension=apc.so" > /etc/php5/apache2/conf.d/apc.ini
#service apache2 restart
```

Una vez instalado APC, se publican con el servidor Web los directorios `control`, `phone` y `player` incluidos en el directorio base de la plataforma.

Es recomendable que el servidor Web en el que se instale la aplicación de control se encuentre en la misma máquina en la que se vaya a ejecutar el servidor *QuQuSI*.

A.1.3. Aplicación para el *chroma* y para los marcadores

Estas aplicaciones requieren que se instale *OGRE 3D* en su versión 1.8.1. En el DVD adjunto a la documentación se proporciona un archivo `.deb` que lleva a cabo la instalación automáticamente. Se deben instalar además los siguientes paquetes:

```
cmake
cmake-gui
build-essential
libfreetype6-dev
libfreeimage-dev
libzzip-dev
libxaw7-dev
libxrandr-dev
freeglut3-dev
libgl1-mesa-dev
libglu1-mesa-dev
libois-dev
gstreamer0.10-alsa
gstreamer0.10-plugins-base
gstreamer0.10-plugins-good
```

```
libgstreamer0.10-dev
libgstreamer-plugins-base0.10-dev
colorgcc
libstdc++6-4.3-dev
```

Una vez instalado *OGRE 3D* y todos los paquetes, se compila cada aplicación llamando a `make` en su directorio.

A.1.4. Servidor *QuQuSI* y aplicación para los pulsadores

Estas dos aplicaciones tan solo requieren ser compiladas con un compilador que soporte el estándar C++11 como *GCC* (4.7.2). Para compilarlas, llamar a `make` desde su directorio.

A.2. DESPLIEGUE

Se recomienda configurar dos redes, una externa en la que se ejecute la aplicación para el público y otra interna con el resto de aplicaciones. Sí se sigue esta configuración, el equipo que ejecute el servidor *QuQuSI* debe tener una interfaz a la red externa a través de la cual pueda comunicar a la aplicación para el público el estado del concurso.

Las aplicaciones cuentan con unos puertos por defecto que se recomienda no alterar. Si se habrá de configurar en cada aplicación la dirección IP del servidor *QuQuSI*. Esta dirección no tiene por qué ser fija en ninguna de las aplicaciones del sistema, pero es recomendable que lo sea al menos en la máquina que ejecute el servidor *QuQuSI*.

A.2.1. Aplicación Web para las preguntas

Para generar un archivo con las preguntas que el servidor *QuQuSI* pueda cargar se debe acceder a la aplicación como administrador y pulsar el botón `exportar`.

El archivo se descargará a la carpeta de `descargas` y se creará una copia junto a los archivos multimedia subidos a la aplicación. De este modo, todo lo necesario para ejecutar el concurso se encontrará en la carpeta media del directorio `webroot` de la aplicación de *CakePHP*.

El servidor *QuQuSI* requiere que el archivo `questions.txt` generado se incluya en su directorio `media/quiz` y la aplicación para el *chroma* necesita tanto las preguntas como los archivos multimedia en su directorio `media/quiz`. Este objetivo se puede lograr de diferentes modos, aunque el más recomendable es el uso de *Unison*.

A.2.2. Aplicación Web de control y para el público

Estas dos aplicaciones requieren que la máquina donde se hayan instalado cuente con una copia de la base de datos *MySQL* con todas las preguntas.

Si se han seguido las recomendaciones del manual, la aplicación Web de control estará desplegada en el mismo equipo que el servidor *QuQuSI* lista para funcionar. Si se configuraron dos redes y la base de datos con la que trabaja la aplicación Web para el público es diferente de la utilizada por la aplicación Web de control, se requiere modificar la dirección IP de la base de datos en la función `mysqli_connect` del archivo `QRecvWinner.php`.

Si se configuraron dos redes, la aplicación Web para el público requerirá que se cambie la dirección IP del servidor en los archivos `QRecvQuest.php` y `QRecvState.php`. Mediante la aplicación auxiliar incluida en el directorio `player` (Véase Anexo I) se introducen los jugadores del público en la base de datos.

Si ambas aplicaciones estan instaladas en servidores Web diferentes, se debe modificar el archivo `QRecvWinner.php` de la aplicación Web de control para que acceda a la base de datos del servidor Web donde se encuentra instalada la aplicación Web para el público.

A.2.3. Servidor *QuQuSI*

El servidor *QuQuSI* requiere que se incluya en su directorio `media/quiz` el archivo generado por la aplicación Web de preguntas.

Como se recomendó anteriormente, la máquina en la que se ejecute esta aplicación debería configurarse con una dirección IP fija para evitar que se tengan que modificar los archivos de configuración del resto de aplicaciones cada vez que esta cambiase.

Los nombres de los equipos que vayan a participar y diferentes factores que afectan a la mecánica del concurso se personalizan en el archivo de configuración del servidor.

El archivo de configuración (`uqusi.config`) debe contar necesariamente con los siguientes parámetros (los valores entre paréntesis son los recomendados):

- **`selMSec` (1000):** Es el tiempo en milisegundos que la función *select* espera a que haya cambios en alguno de los descriptores suscritos.
- **`scorePort` (3000):** Puerto utilizado para la comunicación con la aplicación para los marcadores.
- **`buttonPort` (3001):** Puerto utilizado para la comunicación con la aplicación para los pulsadores.

- **`chromaPort` (3002):** Puerto utilizado para la comunicación con la aplicación para el *chroma*.
- **`statePort` (3003):** Puerto utilizado para enviar el estado del concurso a las aplicaciones Web.
- **`eventPort` (3004):** Puerto utilizado para recibir las señales de control desde la aplicación Web de control.
- **`questionPort` (3005):** Puerto utilizado para enviar la pregunta actual a las aplicaciones Web.
- **`timePort` (3006):** Puerto utilizado para enviar información a la aplicación Web de control.
- **`nSubPhases`:** Numero de subfases de cada categoría. Cada subfase consiste en una ronda de tipo test para cada equipo y una ronda de tipo miscelánea.
- **`time`:** Es el tiempo que dura cada subfase en segundos.
- **`fTime`:** Es el tiempo que dura la ronda final.
- **`nTestQ`:** Número de preguntas de una ronda test.
- **`nMiscQ`:** Número máximo de preguntas de una ronda miscelánea.
- **`testPoints`:** Puntos obtenidos por contestar correctamente a una pregunta de una ronda test (se penaliza con la mitad).
- **`miscPoints`:** Puntos obtenidos por contestar correctamente a una pregunta de una ronda test (se penaliza con la mitad).
- **`chooseTime`:** Tiempo que se da a los concursantes para dar una respuesta a una pregunta de tipo test, en segundos.
- **`counterTime`:** Tiempo que se da a los concursantes para contraatacar a una respuesta.
- **`bachTeamA`:** Nombre del primer equipo de bachillerato.
- **`bachTeamB`:** Nombre del segundo equipo de bachillerato.
- **`bachTeamC`:** Nombre del tercer equipo de bachillerato.

- **`cyclTeamA`:** Nombre del primer equipo de ciclos formativos.
- **`cyclTeamB`:** Nombre del segundo equipo de ciclos formativos.
- **`cyclTeamC`:** Nombre del tercer equipo de ciclos formativos.

A.2.4. Aplicación para el *chroma*

La aplicación para el *chroma* requiere que se incluya en su directorio `media/quiz` el archivo generado por la aplicación Web de preguntas y los contenidos multimedia asociados.

Una vez hecho esto, tan solo se ha de establecer la dirección IP del servidor *QuQuSI* en el archivo de configuración. También se pueden alterar algunos parámetros como el volumen y activar el modo *debug* desde el archivo de configuración.

En el modo *debug* se retrocede y avanza de pregunta con las flechas de cursor izquierda y derecha y se muestra y oculta la pregunta con las flechas de cursor arriba y abajo.

El archivo de configuración (`chroma.config`) debe contar necesariamente con los siguientes parámetros (los valores entre paréntesis son los recomendados):

- **`chromaServerPort` (3002):** Puerto del servidor para atender a la aplicación para el *chroma* (configurado en el parámetro chromaPort del archivo de configuración del servidor).
- **`hostname`:** URL o dirección IP de la máquina donde se ejecuta el servidor.
- **`debug`:** Modo *debug* activado (0 no, 1 si).
- **`musicVolume`:** Volumen de la música de fondo (un valor entre 0 y 1).
- **`soundVolume`:** Volumen de los efectos de sonido (un valor entre 0 y 1).
- **`videoVolume`:** Volumen de los vídeos de introducción (un valor entre 0 y 1).
- **`mediaVolume`:** Volumen de los contenidos multimedia asociados a las preguntas (un valor entre 0 y 1).

A.2.5. Aplicación para los marcadores

La aplicación para los marcadores tan solo requiere que se configure la dirección IP del servidor *QuQuSI* y que se fije un identificador de equipo válido.

El archivo de configuración (`score.config`) debe contar necesariamente con los siguientes parámetros (los valores entre paréntesis son los recomendados):

- **`scoreServerPort` (3000):** Puerto del servidor para atender a la aplicación para los marcadores (configurado en el parámetro scorePort del archivo de configuración del servidor).
- **`hostname`:** URL o dirección IP de la máquina donde se ejecuta el servidor.
- **`team`:** El identificador del equipo al que corresponde el marcador. Los valores válidos son `equipo_a`, `equipo_b`y `equipo_c`.

A.2.6. Aplicación para los pulsadores

Su despliegue requiere dos pasos. El primero es crear el circuito eléctrico de los pulsadores (Fig. 5.17). El segundo es cargar el código del archivo `arduino.ino` en un microcontrolador *Arduino* mediante el IDE.

Una vez hecho esto, tan solo se ha de configurar la dirección IP del servidor *QuQuSI* en el archivo de configuración.

El archivo de configuración (`buttonClient.config`) debe contar necesariamente con los siguientes parámetros (los valores entre paréntesis son los recomendados):

- **`buttonServerPort` (3001):** Puerto del servidor para atender a la aplicación para los pulsadores (configurado en el parámetro buttonPort del archivo de configuración del servidor).
- **`arduinoDevFile`:** Ruta absoluta al archivo de dispositivo del puerto serie al que está conectado el microcontrolador Arduino (ejemplo: `/dev/ttyUSB0`).
- **`hostname`:** URL o dirección IP de la máquina donde se ejecuta el servidor.

A.3. ARRANQUE

Para ejecutar la plataforma se han de llevar a cabo las siguientes tareas:

1. Lanzar el servidor *QuQuSI* en primer lugar.
2. Ejecutar el resto de aplicaciones en cualquier orden. Una vez establecida la conexión con el servidor, los clientes tratarán de volver a conectarse si se pierde la comunicación.
3. Configurar las aplicaciones para el *chroma* en el modo deseado (tecla 1 *chroma*, tecla 2 *proyector*).

4. Probar el circuito eléctrico de los pulsadores viendo los mensajes lanzados por la aplicación Web para los pulsadores.

5. Comprobar que los marcadores se han configurado con un identificador de equipo correcto.

6. Abrir la aplicación Web de control con un navegador Web.

7. Seguir las instrucciones de los mensajes pulsando los botones correspondientes en cada momento.

APÉNDICE B. MECÁNICA DE JUEGO

En este anexo se describe la mecánica de juego del concurso al que da soporte la plataforma QuQuSI en su fase de demostración. Este concurso se corresponde con la fase final de las VII Olimpiadas Informáticas celebrada en las instalaciones de la Escuela Superior de Informática de Ciudad Real.

B.1. NORMAS GENERALES

En el concurso se enfrentan tres equipos. Cada equipo cuenta con un marcador de puntuación y un pulsador. El objetivo del concurso es ser el equipo con mayor puntuación cuando se termina el tiempo. El desarrollo del concurso es el que sigue:

1. Se lleva a cabo una ronda test para el primer equipo.
2. Se lleva a cabo una ronda test para el segundo equipo.
3. Se lleva a cabo una ronda test para el tercer equipo.
4. Se lleva a cabo una ronda miscelánea.
5. Se repiten los pasos anteriores tantas veces como proceda.
6. Se lleva a cabo la ronda final.
7. Se anuncia el equipo ganador.
8. Se cambia a la siguiente categoría si procede, realizando de nuevo los pasos anteriores con los nuevos concursantes.
9. Se anuncian los ganadores del público.
10. El concurso termina.

B.2. TIPOS DE RONDA

El concurso tiene tres tipos de rondas diferentes que tienen lugar en el orden que se indicó en la sección anterior:

- **Ronda test:** Es la ronda principal. Se realizan un número determinado de preguntas de tipo test multimedia a cada uno de los equipos. Cuando el equipo elige una opción o no da una respuesta a tiempo, se pasa al contraataque. Si otro equipo piensa que la opción elegida es incorrecta, puede activar su pulsador para tomar el turno. Si la opción dada fue correcta, se puntúa al equipo que dio la respuesta y se penaliza a los que contraatacaron. Si la opción elegida fue incorrecta, se pasa el turno al primer equipo que contraatacara. El público participa durante esta ronda, pudiendo responder mientras dure el tiempo de la cuenta atrás para responder.

- **Ronda miscelánea:** Esta ronda ocurre después de haber realizado una ronda test para cada equipo. Se realizan preguntas de diferente tipo a test multimedia. La ronda finaliza cuando se han lanzado un número determinado de preguntas o se agota el tiempo disponible. El público no participa en este tipo de ronda.

- **Ronda final:** Es la última ronda. En ella, se lanzan preguntas de todos los tipos. La ronda finaliza cuando ha transcurrido un tiempo determinado. Las puntuaciones y penalizaciones valen el doble durante esta ronda. El público no participa durante la ronda final.

B.3. TIPOS DE PREGUNTA

Se cuenta con los siguientes cinco tipos de preguntas:

- **Test multimedia:** Es una pregunta tipo test con cuatro opciones, de las cuales solo una es correcta. Puede tener asociado contenido multimedia (imagen, vídeo o sonido) que se reproduce a la vez que se muestra la pregunta. El objetivo es elegir la opción correcta entre las cuatro proporcionadas.

- **Adivinar imagen:** En esta pregunta se muestra una imagen a la que se ha aplicado un zoom y una rotación. Conforme avanza el tiempo, la imagen va volviendo a su aspecto original. El objetivo de la prueba es averiguar que es lo que se muestra en la imagen. Se proporciona una pista a los concursantes para ayudarles.

- **Emparejar elementos:** Se muestra una pista y tres parejas de elementos desordenados. El objetivo es emparejar los elementos de la izquierda con los de la derecha de forma correcta.
- **Respuesta incompleta:** Se trata de una prueba tipo "ahorcado". Se muestra una pregunta y algunos caracteres de la solución. El objetivo es completar la respuesta rellenando los huecos en blanco.
- **Frase oculta:** En esta pregunta se realiza una pregunta y se muestran los espacios que ocuparían los caracteres de su respuesta. Conforme avanza el tiempo en estos espacios van apareciendo los caracteres que forman la solución. El objetivo es dar la respuesta exacta completando los huecos.

APÉNDICE C. INVENTARIO

En esta sección se detallan las especificaciones del material empleado en el despliegue de *QuQuSI* durante la fase final de las VII Olimpiadas Informáticas.

- 3 x Cámara de control remoto SONY EVI-D100
  - Señal de vídeo
    NTSC color, estándares JEITA
  - Lente
    10x (óptico), 40x (digital)
  - Salida de video
    Pin jack RCA
  - Alimentación
    CC 12 V (CC 10.8 a 13 V)
  - Dimensiones
    113 x 120 x 132 mm
  - Peso
    880 g
- 1 x Control remoto SONY Remote Control Unit RM-BR300
  - Entrada/salida de control
    VISCA RS-232C OUT: Tipo Mini DIN de 8 terminales
    VISCA RS-422: Tipo 9 terminales
    TALLY IN/CONTACT OUT: Tipo 9 terminales
  - Formato de señal de control
    9.600 bps/38.400 bps
    Datos: 8 bits
    Bit de parada: 1
  - Conector de alimentación
    JEITA tipo 4 (DC IN 12V)

  - Tensión de entrada
    12 V CC (10,8 a 13,2 V CC)
  - Consumo de corriente
    0,2 A máx. (a 12 V CC), 2,4 W
  - Temperatura de funcionamiento
    0°C a +40°C
  - Dimensiones 391,3 x 185 x 145,9 mm (ancho/alto/fondo)
  - Peso Aprox. 950g

- 1 x Conmutador de vídeo Datavideo Digital Video Switcher SE-800.

  - Formatos de vídeo
    Y/C analógico, NTSC CCIR601 compuestoy PAL
    Vídeo Y.U.V analógico; estándar Sony Betacam
  - Entradas de vídeo
    4 - DV, componentes, S (Y/C), compuesto
  - Salidas de vídeo
    4 - DV, componentes, S (Y/C), compuesto
    1 - Salida SDI
  - Entradas de audio
    4 - Fuentes de vídeo A a D, música, auxiliar, micrófono estéreo
  - Salida de audio
    2 salidas estereo principales y 1 salida para cascos con control de volumen
  - Efectos digitales
    PIP, chroma key, zoom, mosaico, blanco y negro, etc.
  - Dimensiones
    430 x 420 x 120 mm
  - Peso
    5,5 Kg
  - Alimentación
    Entrada: CA 90 a 240V
    Salida: CC 15V/70W

- 1 x Panel LCD Datavideo TLM-404

  - Visualización
    4” TFT LCD matriz activa, resolución 480 x 234

  - Relación de aspecto
    4:3
  - Entrada de vídeo
    1.0 Vp-p, 75 Ohm
  - Sistema de vídeo
    NTSC/PAL automático
  - Alimentación
    CC 12 V 50 W

- 6 x Monitor LG 16”
- 3 x Proyectores
- 1 x Grabador DVD genérico
- 1 x EMTEC Movie Cube V700H

  - Entrada/salida
    Disco Duro 3.5” SATA
    2 Puertos USB 2.0 HOST
    1 Puerto USB DEVICE
    Tarjetas SD/SDHC/MMC/MS
    Clavija compuesta A/V
    Clavijas AV IN
    Clavija componentes Y/Pb/Pr
    Clavija Euroconector
    Interfaz HDMI V1.3
    DC-IN 12V
  - Sistemas de archivos soportados
    FAT32/NTFS
  - Formatos soportados
    AVI/MKV/TS/MPG/MP4/MOV/VOB/ISO/IFO/DAT/WMV/RMVB, resolución hasta 720p/1080i/1080p
    MP3/OGG/WMA/WAV/FLAC
  - Codecs de vídeo soportados
    MPEG-1
    HD MPEG-2
    HD MPEG-4 SP /ASP/AVC WMV9
  - Codecs de audio soportados
    MP2/3, OGG Vorbis, WMA estándar, PCM, LPCM, RA

  - Formato de vídeo para las grabaciones
    TS
    MPG(MPEG2, MP2)

- 3 x Micrófono AKG WMS 40 FLEXX

  - Frecuencia portadora 660 - 865 MHz
  - Modulación FM
  - Ancho de banda de transmisión audio 65 - 20.000 Hz
  - Alimentación 1 pila de 1,5 V tamaño AA
  - Tiempo de operación 31 horas
  - Dimensiones 229 x 53 x 53 mm
  - Peso 214 g

- 1 x Micrófono de diadema MicroMic C417
- 1 x Mezclador de audio MACKIE DFX6

  - 2 canales mono
    Entrada de ganancia variable
    Alimentación fantasma
    Indicador LED de ajuste de ganancia
    Jack de entrada micrófono XLR
    Jack de entrada de línea 1/4” TRS
    Jack de inserción 1/4” TRS
    Filtro de corte bajo de 75 Hz cambiable
    Conmutador “Mute”
    Indicador LED de sobrecarga
  - 2 canales mono-mic / de línea-estéreo
    Entrada de ganancia variable
    Indicador LED de ajuste de ganancia
    Jack de entrada de micrófono XLR
    Jack de entrada de línea 1/4” TRS izquierdo y derecho
    EQ de 2 bandas
    Conmutador de “Mute”
    Indicador LED de sobrecarga

- 1 x Amplificador OMNITRONIC P-500
- 2 x Altavoz OMNITRONIC KPX-115

- 1 x Arduino Diecimila
- 3 x Pulsadores artesanales
- 2 x Puntos de acceso Wifi
- 4 x Conversores de señal

APÉNDICE D. FOTOS DEL MONTAJE

En este apéndice se muestran las fotografías tomadas durante el montaje del escenario del concurso realizado para las VII Olimpiadas Informáticas.

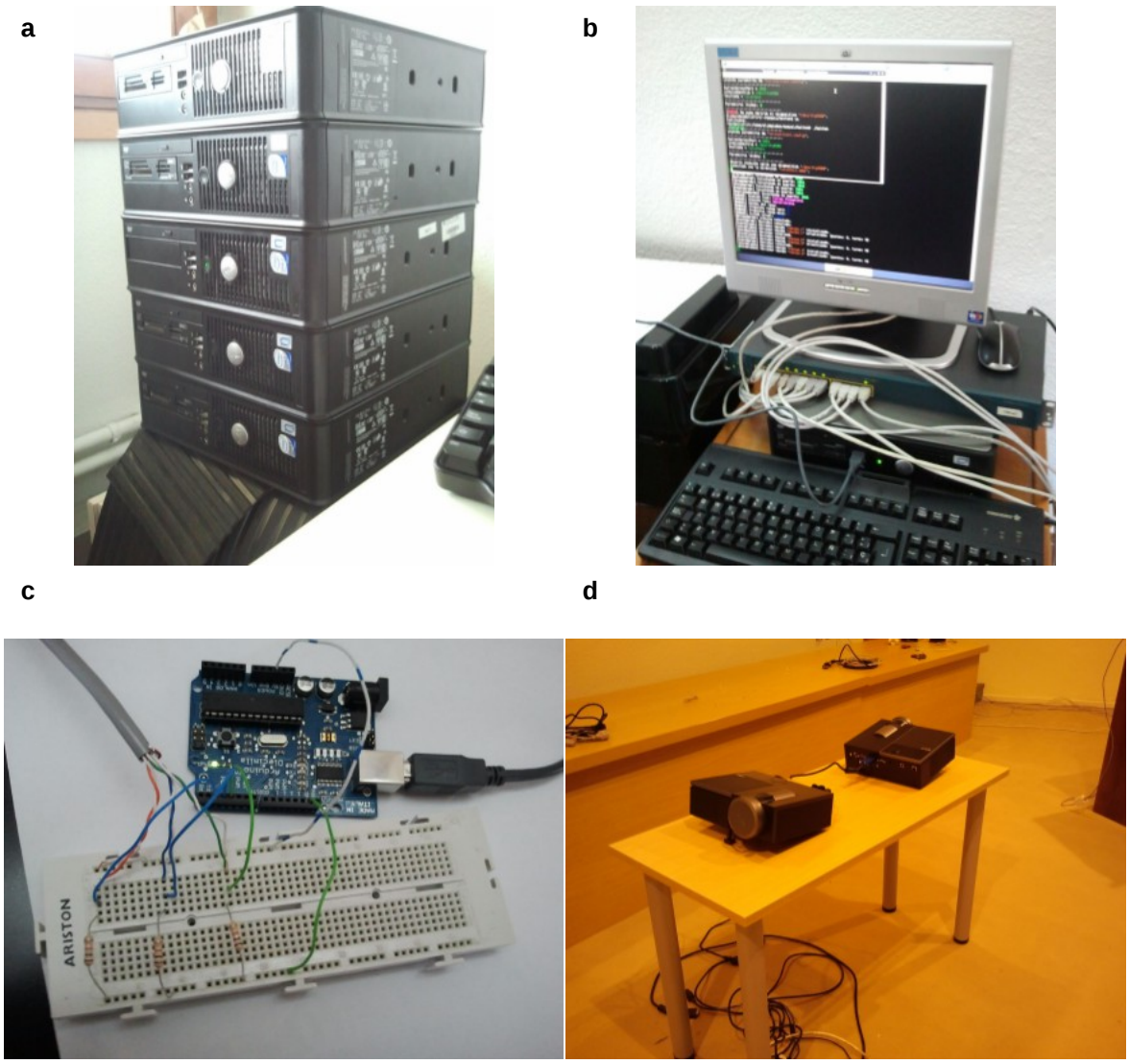

Figura D.1: a) Equipos empleados durante el despliegue (un equipo de reserva). b) Conmutador y servidor principal conectado por USB al microcontrolador Arduino ejecutando el servidor QuQuSI y la aplicación para los pulsadores. c) Microcontrolador Arduino y circuito electrónico para los pulsadores. d) Detalle de la mesa con los proyectores para mostrar la aplicación para el *chroma* en las paredes del salón de actos.

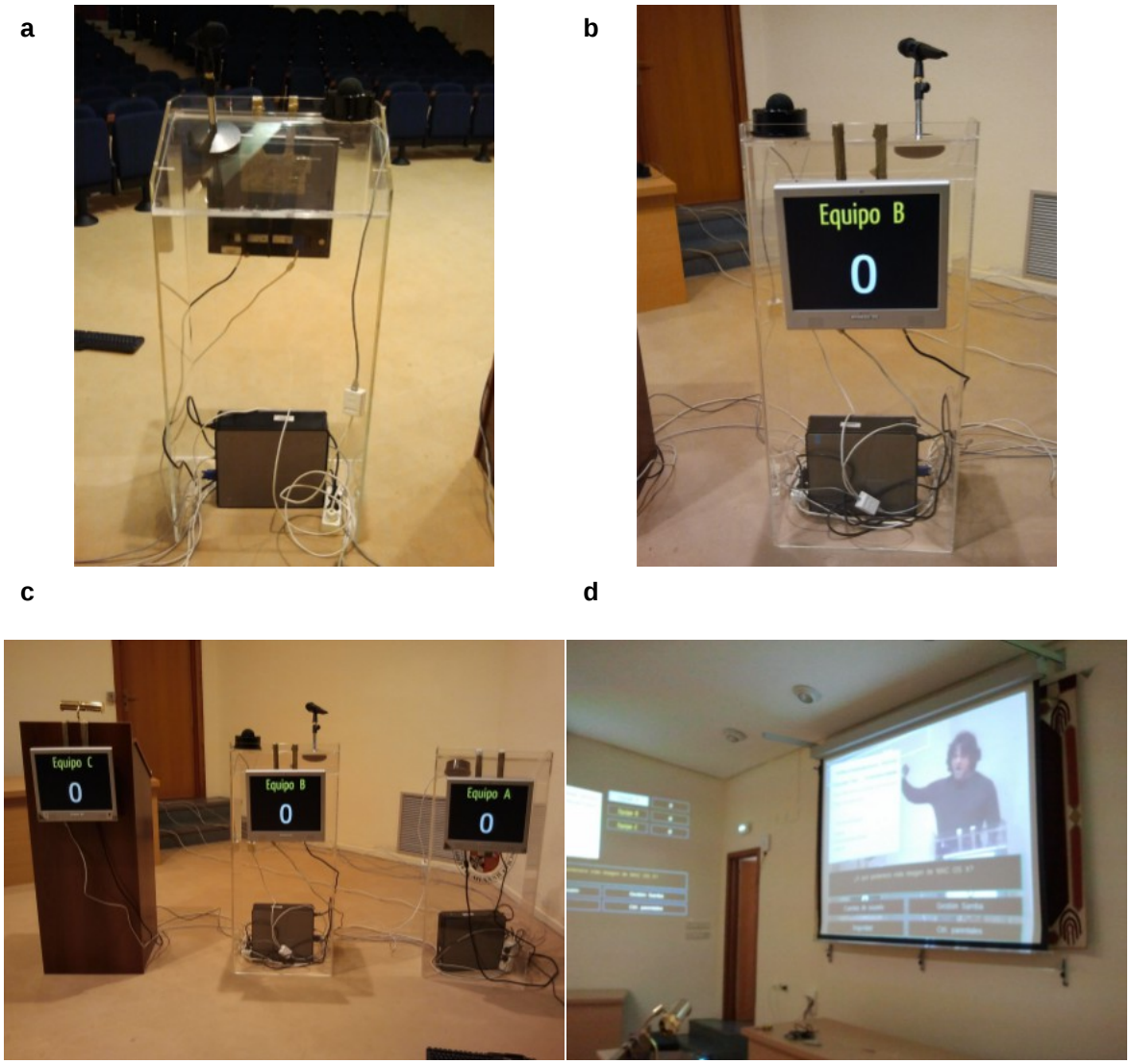

Figura D.2: a) Vista trasera de un atril de los concursantes a falta de la decoración frontal. b) Vista frontal de un atril de los concursantes a falta de la decoración. c) Atriles con los equipos ejecutando la aplicación para los marcadores. d) Prueba de proyección de la aplicación para el *chroma*.

a

b

c

d

Figura D.3: a) Pulsador artesanal elaborado por José Antonio Fernández, miembro del equipo técnico de la escuela. b) Salón de actos con la plataforma desplegada y el sistema en funcionamiento. c) Vista frontal del atril del presentador. d) Vista desde el portátil táctil del presentador con la aplicación Web de control en funcionamiento.

a

b

c

d

Figura D.4: a) Vista desde los atriles de los concursantes. b) Proceso de montaje de la decoración impresa para cubrir los atriles. c) Mesa de mezcla de audio y equipos para la aplicación para el *chroma* y para la aplicación Web de control del realizador. d) Salón de actos con la plataforma desplegada y la decoración montada.

a

b

c

d

Figura D.5: a) Vista frontal del montaje provisional de la decoración de los atriles. b) Vista desde el portátil táctil del presentador con los atriles decorados. c) Vista trasera de los atriles de los concursantes con la decoración montada. d) Detalle de los atriles con los micrófonos y los pulsadores montados.

Figura D.6: a) Miembros de la Escuela Superior de Informática prueban el sistema. b) Mesa de mezcla de vídeo con las pantallas de visualización y los grabadores. c) Vista trasera de un atril de los concursantes con el micrófono y el pulsador fijados y la decoración montada. d) Detalle de la mesa con los proyectores y los pulsadores de repuesto con la decoración montada.

a

b

Figura D.7: a) Vista trasera de los atriles de los concursantes con los micrófonos y los pulsadores fijados y la decoración montada. b) Vista frontal de un atril de los concursantes con la decoración montada.

APÉNDICE E. FOTOS DEL CONCURSO

En este capítulo se recogen capturas del video final editado en la fase de postproducción tras la grabación del concurso, realizado para la fase final de las VII Olimpiadas Informáticas en la Escuela Superior de Informática de Ciudad Real el día 24 de mayo de 2013.

El vídeo final del evento está accesible en el canal de YouTube de la Escuela Superior de Informática de Ciudad Real[1].

[1] http://www.youtube.com/user/esiuclm

Figura E.1: a) Los equipos de bachillerato preparados para empezar. b) El equipo Fbr03 se enfrenta a una pregunta test sobre sistemas operativos. c) Varios miembros del público contestando a la misma pregunta que los participantes.

a

b

c

Figura E.2: a) El equipo Desmagnetizar solicita el turno para responder a una pregunta de emparejar elementos.b) Se muestra la solución de una pregunta de tipo respuesta incompleta. c) Los concursantes se enfrentan a una pregunta de tipo frase oculta.

Figura E.3: a) El equipo Fbr03 se piensa la respuesta a la pregunta test de informática básica. b) El equipo Error404 respondiendo a una pregunta test con un vídeo asociado. c) Plano del salón de actos durante el cambio de turno de los equipos.

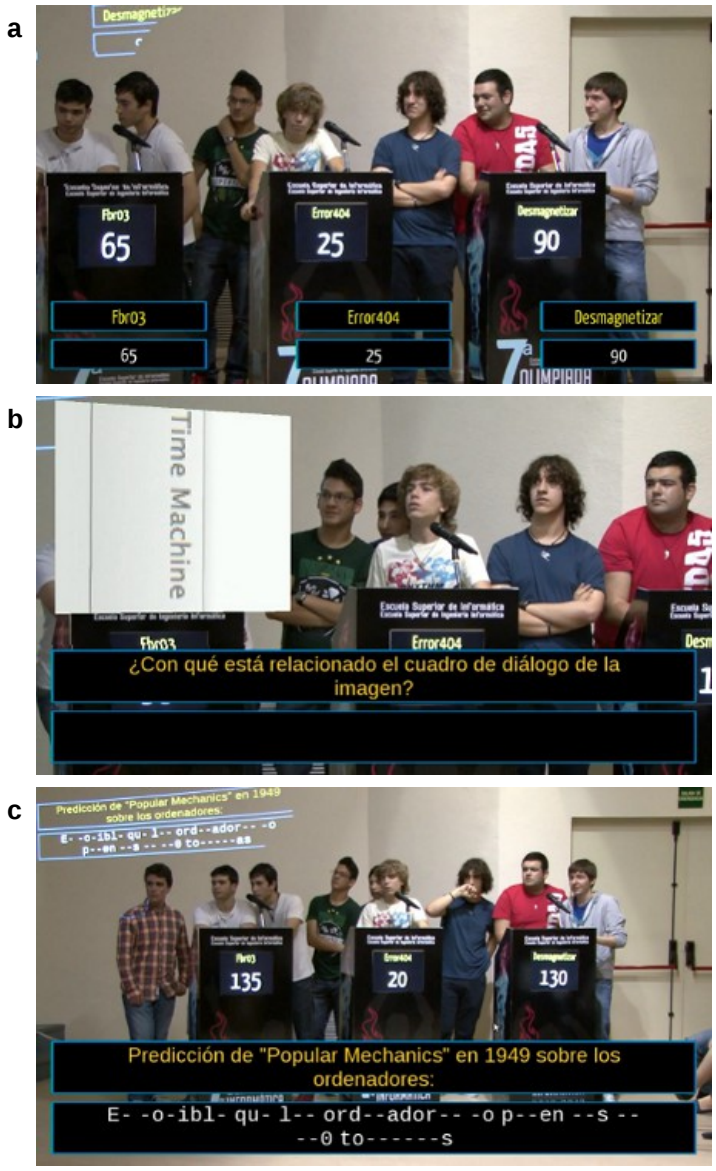

Figura E.4: a) Clasificación parcial de los equipos tras finalizar una ronda. b) Los equipos tratan de encontrar la solución a una pregunta de tipo adivinar imagen. c) La tensión aumenta en la ronda final del concurso cuando los errores penalizan el doble.

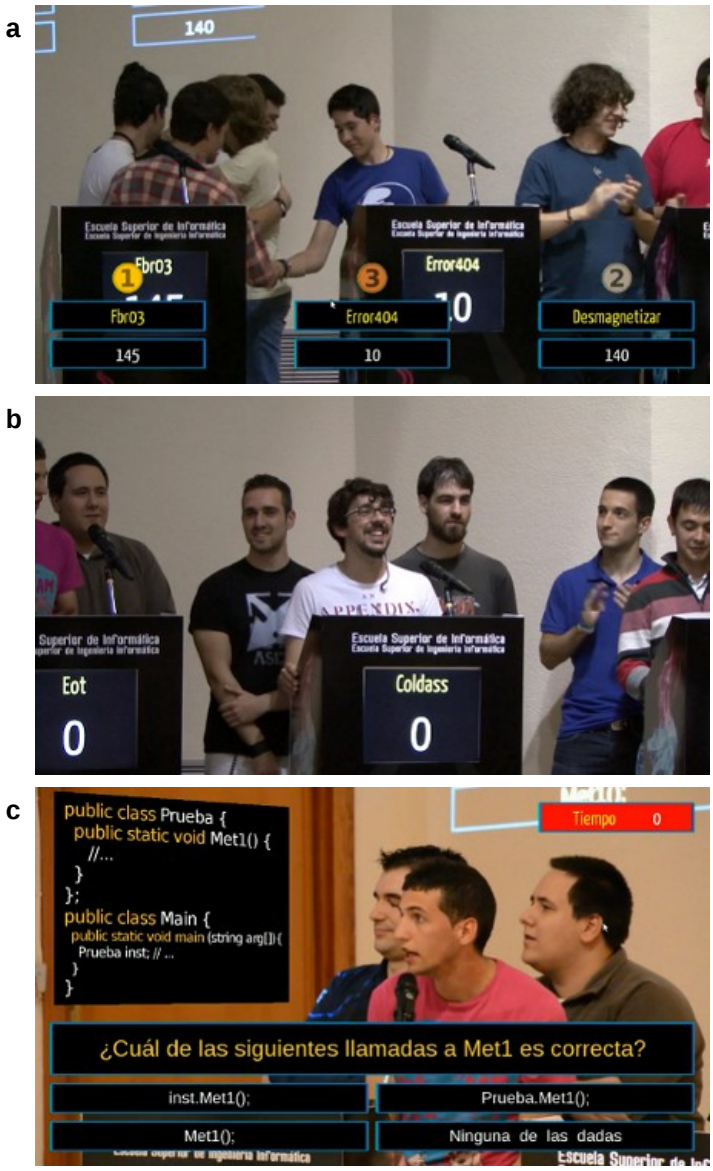

Figura E.5: a) Deportividad entre equipos tras la finalización de la categoría de bachillerato. b) Presentación de los equipos de ciclos formativos. c) El equipo Eot da la respuesta a una pregunta de programación en el último momento.

Figura E.6: a) El equipo Coldass piensa la solución a una pregunta de sistemas operativos. b) El equipo Fbr contesta correctamente a una pregunta test de sistemas operativos. c) El equipo Eot activa el pulsador antes que el equipo Mbr.

Figura E.7: a) Clasificación parcial de los equipos de ciclos formativos tras finalizar una ronda. b) Clasificación final de los equipos de ciclos formativos. c) Los ganadores de cada categoría del público.

APÉNDICE F. CARTAS DE RECOMENDACIÓN

En este apéndice se recogen las cartas de recomendación recibidas hasta el momento tras la grabación del concurso. Se estudiarán además otras vías de comercialización de la plataforma. Las entidades y productoras que han mostrado su interés son:

- **UPSLP:** Universidad Politécnica de San Luis Potosí (México), especializada en la enseñanza de las nuevas tecnologías de la información.
- **CrTV:** Canal de televisión municipal de Ciudad Real.
- **Imás TV:** Canal de televisión emitido en Ciudad Real capital y poblaciones adyacentes.

UNIVERSIDAD POLITÉCNICA DE SAN LUIS POTOSÍ

San Luis Potosí, México
A 10 de Julio de 2013

Atención
Carlos González Morcillo
Subdirector de Calidad y Relaciones externas
Escuela Superior de Informática de Ciudad Real.

Por medio de la presente me permito solicitar de manera formal el software denominado "*Sistema para la construcción de concursos televisivos*", el cual fue desarrollado por el alumno **Eduardo Monroy Martínez** de la Escuela Superior de Informática. Dicho software nos resultaría de suma ayuda, como parte de la divulgación de la carrera de Ingeniería en Tecnologías de la Información, ayudándonos de forma específica al implementar un concurso de conocimientos entre los alumnos de los distintos Bachilleratos en la ciudad. Resaltando que dicho software es totalmente personalizable acorde a nuestras necesidades, como la gestión de diversos tipos de preguntas multimedia, además de que permite la participación del público espectador mediante dispositivos móviles.
Sin más por el momento me permito enviarles un saludo esperando una respuesta favorable a mi solicitud.

Atte.

Dr. César A. Guerra García
Profesor-Investigador de la UPSLP
Academia de Tecnologías de la Información y Telemática

Figura F.1: Carta de Cesar A. Guerra García, Profesor-Investigador de la Universidad Politécnica de San Luis Potosí.

CARTA DE RECOMENDACIÓN DE PRODUCTO

Ciudad Real a 11 de Julio de 2013

Estimados señores,

En calidad de responsable técnico de CRTV, quisiera destacar el interés despertado por nuestro canal de televisión en la incorporación a futuro del sistema para la construcción de concursos televisivos desarrollado en la Escuela Superior de Informática de Ciudad Real por ***Eduardo Monroy Martínez.*** La decisión de utilizar este sistema en la producción de contenidos audiovisuales para nuestro medio, se puede resumir en lo que consideramos tres puntos clave para nosotros :

- Posibilidad de gestión de preguntas multimedia en diversos formatos.
- Fácil adaptación a nuestro sistema de escenografía virtual y chroma.
- Opción de participación del público en directo mediante terminales móviles.

Por lo anteriormente expuesto y dada la versatilidad de opciones que abre esta herramienta, consideramos este sistema como un recurso fundamental para el desarrollo de productos televisivos enmarcados dentro de un formato que actualmente viene siendo habitual en muchas parrillas de programación.

Fdo : Miguel Ángel Bajo Cabezas

Jefe Técnico de Televisión Ciudad Real

C/ Borja , nº 3

TLF.: 926 21 10 44

Figura F.2: Carta de Miguel Ángel Bajo Cabezas, Jefe Técnico de Televisión Ciudad Real.

POLIGONO LA NAVA 3
C/FRANCIA, 4
13500.- PUERTOLLANO
C.REAL

A quien corresponda:

Por medio de la presente, yo D. Julián Camacho Morejudo, en calidad de Consejero Delegado de Imás TV, expreso el interés que ha suscitado el sistema "QuQuSI: Plataforma para la Gestión Integral de Concursos Televisivos", desarrollado en la Escuela Superior de Informática de Ciudad Real, para su posible adaptación como entorno para la generación de concursos en los canales de nuestro grupo de empresas.

En la actualidad no existe ninguna plataforma abierta de características similares, que integre tanto la parte de gestión de preguntas, como su posterior despliegue 2D y 3D, emisión en tiempo real y postproducción. Otro elemento a destacar de la plataforma son las opciones de participación en tiempo real por parte del público empleando cualquier dispositivo con navegador y conexión a Internet. Las capacidades de la plataforma quedan patentes en su utilización para la producción de la fase final en las VII Olimpiadas de la Informática.

Y para que así conste, y a petición de D. Eduardo Monroy Martínez, firmo el presente documento en Ciudad Real, a 23 de Agosto de 2013.

Atentamente

Fdo: D. Julián Camacho Morejudo
Consejero Delegado Imás TV

Figura F.3: Carta de D. Julián Camacho Morejudo, Consejero Delegado de Imás TV.

APÉNDICE G. *DOWNGRADE* CONTROLADOR NVIDIA

Durante el desarrollo del proyecto el controlador de tarjetas gráficas *NVIDIA* del repositorio de Debian Wheezy fueron actualizados a una versión que no permite utilizar resoluciones no nativas en los monitores y proyectores conectados desde el gestor de configuración *NVIDIA*.

Para recuperar esta funcionalidad, se volvió a una versión anterior del controlador siguiendo los pasos descritos a continuación:

1. Desinstalar los drivers de la versión actual. El procedimiento cambia en función del modo de instalación [1].

2. Para instalar la versión anterior de los drivers se utilizará `module-assistant`, por lo que es necesario instalar dicho paquete:

   ```
   # aptitude install module-assistant
   ```

3. Comentar todos los repositorios en `/etc/apt/sources.list` e incluir los repositorios con la versión deseada (295.59 en `squeeze-backports`):

   ```
   deb http://backports.debian.org/debian-backports #
       squeeze-backports main contrib non-free
   ```

4. Instalar el siguiente paquete mediante el gestor de paquetes:

   ```
   # aptitude install nvidia-kernel-common
   ```

5. Utilizar `module-assistant` para compilar el módulo de kernel *NVIDIA* mediante:

   ```
   # m-a a-i nvidia
   ```

6. Este comando instala los siguientes paquetes en el sistema:

   ```
   nvidia-kernel-source
   nvidia-kernel-`uname -r` (3.2.0-4-amd64 en este caso)
   ```

[1] http://wiki.debian.org/NvidiaGraphicsDrivers

7. A continuación, instalar el siguiente paquete mediante el gestor de paquetes:

```
# aptitude install nvidia-glx
```

8. Este paquete instala las siguientes dependencias:

```
glx-alternative-mesa{a}
glx-alternative-nvidia{a}
glx-diversions{a}
libgl1-nvidia-alternatives{a}
libgl1-nvidia-glx{a}
libglx-nvidia-alternatives{a}
nvidia-alternative{a}
nvidia-installer-cleanup{a}
nvidia-support{a}
nvidia-vdpau-driver{a}
xserver-xorg-video-nvidia{a}
```

9. Instalar el gestor de configuración nvidia con el gestor de paquetes:

```
# aptitude install nvidia-settings
```

10. Volver a editar `/etc/apt/sources.list` para quitar los comentarios y comentar los repositorios con la versión deseada (`squeeze-backports`).

11. En el gestor de paquetes, marcar como retenidos todos los paquetes que se han instalado durante el proceso (salvo `module-assistant`). Si se utilizan varios gestores de paquetes se han de bloquear los paquetes en todos ellos para prevenir una actualización no deseada.

```
# aptitude hold \
nvidia-kernel-common \
nvidia-kernel-source \
nvidia-kernel-`uname -r` \
nvidia-glx \
glx-alternative-mesa \
glx-alternative-nvidia \
glx-diversions \
libgl1-nvidia-alternatives \
```

```
libgl1-nvidia-glx \
libglx-nvidia-alternatives \
nvidia-alternative \
nvidia-installer-cleanup \
nvidia-support \
nvidia-vdpau-driver \
xserver-xorg-video-nvidia \
nvidia-settings
```

12. Instalar el generador de archivos de configuración *NVIDIA* con el gestor de paquetes:

    ```
    # aptitude install nvidia-xconfig
    ```

13. Ejecutar el comando nvidia-xconfig como superusuario para generar el nuevo archivo de configuración xorg.conf:

    ```
    # nvidia-xconfig
    ```

14. Si tras realizar todo el proceso se nota una bajada en el rendimiento de algunas aplicaciones gráficas (*Google Chrome* en este caso) se puede solucionar cambiando la biblioteca `libcairo2` a una versión anterior siguiendo el mismo procedimiento.

APÉNDICE H. CÓDIGO CLASES PATRÓN

En esta sección se muestra el código de las clases plantilla con las que se implementan los patrones *singleton*, *observer* y *state*.

H.1. CÓDIGO PATRÓN *SINGLETON*

Para utilizar el patrón *singleton*, heredar de la clase `QSingleton` (Listado H.1), declarar la clase plantilla como amiga y declarar el constructor de la clase como privado.

H.2. CÓDIGO PATRÓN *OBSERVER*

Para utilizar el patrón *observer*, heredar con la clase observadora de la clase `QObserver` (Listado H.2) y con la clase observada de `QSubject` (Listado H.3).

H.3. CÓDIGO PATRÓN *STATE*

Para utilizar el patrón *state*, heredar de la clase `QState` (Listado H.4) con cada estado particular e instanciar la clase `QStateMachine` (Listado H.6). También puede crearse una nueva clase que herede de la clase base `QStateMachine` y sobreescriba sus métodos virtuales para, por ejemplo, mostrar mensajes o llamar a otras funciones al cambiar de estado.

```
#ifndef QSINGLETON_H
#define QSINGLETON_H
template<class T> class QSingleton {
public:
    // Recuperar instancia ------------------------------------------
    // Referencia a instancia
    static T& instanceRef() {
        static T instance; // Instancia unica
        return instance;
    }
    // Puntero a instancia
    static T* instancePtr() {
        return &instanceRef();
    }
private:
    // Constructor y destructor privados ----------------------------
    friend T; // Clase derivada amiga
    QSingleton() {}
    virtual ~QSingleton() {}
    // Copia y asignacion eliminados --------------------------------
    QSingleton(const QSingleton&) = delete;
    QSingleton& operator=(const QSingleton&) = delete;
};
#endif
```

Listado H.1: Código del archivo `QSingleton.h`.

```
#ifndef QOBSERVER_H
#define QOBSERVER_H
template<class T> class QObserver {
public:
    // Constructor y destructor -------------------------------------
    QObserver() {}
    virtual ~QObserver() {}
    // Funciones observador -----------------------------------------
    virtual void update(T* pSubject) = 0;
};
#endif
```

Listado H.2: Código del archivo `QObserver.h`.

```
#ifndef QSUBJECT_H
#define QSUBJECT_H
#include "QObserver.h"
#include <list>
template<class T> class QSubject {
public:
    // Constructor y destructor ----------------------------------------
    QSubject() {}
    virtual ~QSubject() {}
    // Funciones observado ---------------------------------------------
    // Anadir observador
    void attach(QObserver<T>* pObserver) {
        _observers.push_back(pObserver);
    }
    // Quitar observador
    void detach(QObserver<T>* pObserver) {
        _observers.remove(pObserver);
    }
    // Notificar observadores
    void notify(T* pSubject) {
        std::list<QObserver<T>*> observers = _observers;
        for (QObserver<T>* pObserver: observers)
            pObserver->update(pSubject);
    }
private:
    std::list<QObserver<T>*> _observers; // Lista de observadores
};
#endif
```

Listado H.3: Código del archivo `QSubject.h`.

```
#ifndef QSTATE_H
#define QSTATE_H
#include <string>
class QStateMachine;
class QState {
public:
    // Constructor y destructor ----------------------------------------
    QState(QStateMachine* pStateMachine = nullptr,
           std::string name = "State"):
        _pStateMachine(pStateMachine),
        _name(name) {}
    virtual ~QState() {}
    // Funciones estado ------------------------------------------------
    virtual void input(std::string event) = 0;
    virtual void enter() = 0;
    virtual void pause() = 0;
    virtual void resume() = 0;
    virtual void exit() = 0;
   // Funciones maquina estados ---------------------------------------
    virtual void changeState(QState* pState);
    virtual void pushState(QState* pState);
    virtual void popState();
    // Accesores -------------------------------------------------------
    std::string name() { return _name; }
protected:
    QStateMachine* _pStateMachine; // Puntero a QStateMachine
    std::string _name;             // Nombre del estado
};
#endif
```

Listado H.4: Código del archivo `QState.h`.

```
#include "QState.h"
#include "QStateMachine.h"
// Funciones maquina estados ---------------------------------------
void
QState::changeState(QState* pState) {
    if (_pStateMachine != nullptr)
        _pStateMachine->changeState(pState);
}
void
QState::pushState(QState* pState) {
    if (_pStateMachine != nullptr)
        _pStateMachine->pushState(pState);
}
void
QState::popState() {
    if (_pStateMachine != nullptr)
        _pStateMachine->popState();
}
```

Listado H.5: Código del archivo `QState.cpp`.

```
#ifndef QSTATEMACHINE_H
#define QSTATEMACHINE_H
#include <string>
#include <stack>
class QState;
class QStateMachine {
public:
    // Constructor y destructor ----------------------------------------
    QStateMachine();
    virtual ~QStateMachine();
    // Funciones maquina estados ---------------------------------------
    virtual void changeState(QState* pState);
    virtual void pushState(QState* pState);
    virtual void popState();
    virtual void clearStates();
    // Funciones estado ------------------------------------------------
    virtual void input(std::string event);
    std::string name();
protected:
    std::stack<QState*> _states; // Pila de estados
};
#endif
```

Listado H.6: Código del archivo `QStateMachine.h`.

```
#include "QStateMachine.h"
#include "QState.h"
// Constructor y destructor ------------------------------------------
QStateMachine::QStateMachine() {}
QStateMachine::~QStateMachine() {}
// Funciones maquina estados -----------------------------------------
void
QStateMachine::changeState(QState* pState) {
    if (!_states.empty()) {
        _states.top()->exit();
        _states.pop();
    }
    _states.push(pState);
    _states.top()->enter();
}
void
QStateMachine::pushState(QState* pState) {
    if (!_states.empty()) {
        _states.top()->pause();
    }
    _states.push(pState);
    _states.top()->enter();
}
void
QStateMachine::popState() {
    if (!_states.empty()) {
        _states.top()->exit();
        _states.pop();
    }
    if (!_states.empty()) {
        _states.top()->resume();
    }
}
void
QStateMachine::clearStates() {
    while (!_states.empty()) {
        _states.top()->exit();
        _states.pop();
    }
}
// Funciones estado --------------------------------------------------
void
QStateMachine::input(std::string event) {
    if (!_states.empty()) {
        _states.top()->input(event);
    }
}
std::string
QStateMachine::name() {
    if (!_states.empty()) {
        return _states.top()->name();
    }
    return "";
}
```

Listado H.7: Código del archivo `QStateMachine.cpp`.

APÉNDICE I. CÓDIGO FUENTE

Dada la extensión del código fuente de la plataforma, en total más de 10.000 líneas, éste no se ha incluido impreso en ningún apéndice. Sin embargo, puede consultarse en su totalidad en el DVD adjunto al presente documento.

A continuación se resume brevemente la estructura de los directorios que se encuentran en el DVD adjunto:

- **deb/:** Contiene el archivo `.deb` para la instalación del *framework OGRE 3D* en distribuciones *GNU/Linux* basadas en *Debian*.
- **doc/:** Incluye el código fuente y las imagenes con las que se generó el presente documento.
- **src/:** En sus subdirectorios se incluye el código fuente de las aplicaciones de la plataforma.
  - **button/:** Contiene el código fuente de la aplicación para los pulsadores y el del microcontrolador *Arduino*.
  - **cakephp/:** Contiene el *framework CakePHP* y el código fuente de la aplicación Web para las preguntas.
  - **chroma/:** Contiene el código fuente de la aplicación para el *chroma*
  - **control/:** Contiene el código fuente de la aplicación Web de control.
  - **parser/:** Contiene el código fuente del reproductor de eventos.
  - **phone/:** Contiene el código fuente de la aplicación Web auxiliar para el público.
  - **player/:** Contiene el código fuente de la aplicación Web para el público.
  - **score/:** Contiene el código fuente de la aplicación para los marcadores.
  - **server/:** Contiene el código fuente del servidor *QuQuSI*.
- **mov/:** Contiene vídeos de las pruebas funcionales del sistema y de la fase final de las VII Olimpiadas Informáticas.

BIBLIOGRAFÍA

[1] Kantar Media
http://www.kantarmedia1.es/

[2] Obliqua Medios
http://www.obliqua.es/

[3] Antena 3
http://www.antena3.com/

[4] Telecinco
http://www.telecinco.es/

[5] Jugar a *«Atrapa un millón»* en directo
http://antena3.v2.playalong.med.playinstar.com/

[6] BAFTA Games
http://www.bafta.org/games/

[7] Google Play
https://play.google.com/

[8] PIPOCLUB
https://www.pipoclub.com/

[9] VGChartz
http://www.vgchartz.com/

[10] About.com
http://linux.about.com/

[11] Netcraft
http://www.netcraft.com/

[12] OpenLogic
http://www.openlogic.com/

[13] GStreamer
http://gstreamer.freedesktop.org/

[14] Ogre3D
http://www.ogre3d.org/

[15] Buzz!
http://www.buzzthegame.com/

[16] Nintendo
http://www.nintendo.com/

[17] Trivial Pursuit - EA Games
http://www.ea.com/es/trivial-pursuit/

[18] Bison - GNU Parser Generator
http://www.gnu.org/software/bison/

[19] Flex - The Fast Lexical Analizer
http://flex.sourceforge.net/

[20] Boost C++ Libraries
http://www.boost.org/

[21] DroidYourSite
http://droidyoursite.com/

[22] Francisco Moya Fernández, Carlos González Morcillo, David Villa Alises, Sergio Pérez Camacho, Miguel A. Redondo Duque, César Mora Castro, Félix J. Villanueva Molina, Miguel García Corchero. *Desarrollo de Videojuegos: Técnicas Avanzadas (1ª Edición)*. Bubok, 2012.

[23] J. L. González. *Jugabilidad: Caracterización de la Experiencia del Jugador en Videojuegos*. Tesis Doctoral, Universidad de Granada, 2010.

[24] Mª del Milagro Fernández Carrobles. *FTPack: Framework de programación de sistemas automatizados educativos basado en Eclipse*. Proyecto Fin de Carrera, Universidad de Castilla-La Mancha, 2010.

[25] ISO/IEC 9241. *ISO/IEC 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability*. ISO, 1998.

[26] Shari Lawrence Pfleeger. *Ingeniería de software: Teoría y práctica*. Prentice Hall, 2002.

[27] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, 2005.

[28] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Patrones de Diseño*. Addison-Wesley, 1995.

[29] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[30] Gerti Kappel, Birgit Pröll, Siegfried Reich, Werner Retschitzegger. *Web Engineering: The Discipline of Systematic Development of Web Applications*. John Wiley & Sons, 2003.

[31] Thomas Connolly, Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management (5th Edition)*. Addison-Wesley, 1998.

[32] Michael J. Donahoo, Kenneth L. Calvert. *The Pocket Guide to TCP/IP Sockets: C Version*. Morgan-Kaufmann, 2001.

[33] Barry W. Boehm. "Software engineering - as it is". *IEE Proceedings*, pp. 11-21, 1976.

[34] G. Brassard, P. Bratley. *Fundamentos de Algoritmia (1ª Edición)*. Prentice Hall, 1997.

[35] Brian W. Evans. *Arduino Programming Notebook*. Publicado en la Web, 2007.

[36] Tomas Akenine-Möller, Eric Haines, Naty Hoffman. *Real-Time Rendering (3rd Edition)*. A K Peters, 2008.

[37] Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*, Addison-Wesley Professional, 2005.

[38] Dickheiser, Michael J. *C++ for Game Programmers (2nd Edition)*. Charles River Media, 2007.

[39] Wim Taymans, Steve Baker, Andy Wingo, Rondald S. Bultje, Stefan Kost. *GStreamer Application Development Manual (1.0.3)*. Publicado en la Web, 2013.