Sistemas Informáticos
Curso 2007 - 2008

Extensión a multijugador de ViRPlay 3D, un juego educativo de enseñanza de orientación a objetos usando role-play

Pablo Fraile García
Guillermo Del Fresno Herena
Ricardo Gómez Miguélez

Dirigido por:
Marco Antonio Gómez Martín
Guillermo Jiménez Díaz

Departamento de Ingeniería del Software e Inteligencia Artificial

Facultad de Informática
Universidad Complutense de Madrid

CONTENIDO

0 *Resumen/Abstract*

0.1 Resumen

El proyecto ha consistido en la extensión de una herramienta educativa monousuario para la enseñanza de la programación orientada a objetos usando la técnica del role-play. Se han ampliado las funcionalidades de la aplicación con el objetivo de introducir el juego multiusuario en red. Para ello, se ha integrado una arquitectura cliente-servidor así como su protocolo correspondiente. A pesar de que este ha sido el trabajo de desarrollo principal, primero ha sido necesario completar las características principales de la versión de la que partimos.

0.2 Abstract

The Project has consisted in extending a single user learning tool for object oriented programming teaching using a role-play technique. New functionalities have been included in order to support multi user mode. Therefore, server-client architecture has been integrated as well as its protocol. Despite being the main development work, firstly it was necessary to complete the features of the starting version.

1 *Introducción*

Las simulaciones son una herramienta educativa habitual. Originalmente, las simulaciones se han utilizado como un entorno tolerante a los errores para proporcionar entrenamiento en el manejo de equipos caros o peligrosos de manipular. Sin embargo, la idea de aprender-haciendo en un entorno simulado se puede llevar más allá, y de hecho se está explorando su aplicabilidad en la enseñanza de, por ejemplo, habilidades sociales, ingeniería o matemáticas.

Los juegos son un tipo particular de simulación con el fin primordial de entretener. Una herramienta educativa disfrazada de juego puede conseguir fácilmente la motivación que requiere el proceso de aprendizaje. Además, cada vez más gente habla el lenguaje de los juegos, sobre todo, cada vez más gente joven.

Un ejemplo de este tipo de sistemas es ViRPlay 3D (VIRtual Role-PLAY 3D), que facilita la comprensión de las interacciones que se producen durante la ejecución de programas orientados a objetos. Para ello, combina tareas de aprendizaje activo con la visualización tanto de la arquitectura como del comportamiento dinámico del sistema. El método se basa en seguir una técnica tomada de la enseñanza de la orientación a objetos, el role-play, llevándolo a un entorno virtual interactivo en 3D.

1.1 ¿Qué es el role-play?

El role-play es principalmente la interpretación de diferentes roles o papeles que interactúan entre si dentro de un entorno y situación concretos, en nuestro caso con la finalidad de hacer una labor educativa. Cada uno de estos papeles es representado por los participantes o actores del role-play, que no solo participan en el desarrollo del mismo, sino que también observan y aprenden.

La orientación a objetos es un paradigma muy usado actualmente que facilita la programación a gran escala, sin embargo, no todo son ventajas. Transmitir este paradigma a estudiantes sin una base sólida, puede ser una tarea muy difícil.

La necesidad de facilitar esta labor de aprendizaje, de forma que los alumnos se vean inmersos en el programa, lleva a utilizar el role-play como método de enseñanza.

Ya que la programación orientada a objetos se caracteriza por los objetos y las interacciones que ocurren entre éstos dentro de la aplicación, el role-play es en mayor o menor medida adaptable para la representación de este paradigma. A partir de ahora cada vez que nos refiramos al role-play, nos estaremos refiriendo al role-play adaptado a la orientación a objetos.

Básicamente lo que se intenta conseguir con el role-play es representar el funcionamiento de una parte de la aplicación, es decir, qué objetos intervienen en ésta y como se comunican entre ellos. Esta comunicación no es más que las diferentes llamadas que se hacen a lo largo de la ejecución del programa.

Por tanto cada una de estas funcionalidades estaría representada por un escenario de role-play, donde los actores del mismo serían cada uno de los objetos que intervienen en la funcionalidad que se está representando. Además siguiendo el orden secuencial de los programas, donde la ejecución se encuentra en cada momento en un objeto concreto, podemos decir que solo uno de ellos tendrá el control del programa. Estos cambios del control entre los objetos serán las diferentes interacciones que se representen en el role-play. El cambio de control se produce cuando un objeto invoca la función de otro objeto, esto se puede ver como un objeto pidiéndole a otro objeto que realice alguna tarea a través del envío o paso de un mensaje. Además existirá un guión u orden concreto en el que se deben efectuar los pasos de mensaje, de acorde al escenario que se está simulando para representar fielmente el funcionamiento de la aplicación.

De esta forma, cada alumno interpreta un papel (rol) que representa a un objeto del escenario. Estos objetos se diferencian entre si ya que cada uno de éstos tendrán unos colaboradores y responsabilidades concretos. Dispondrán cada uno de ellos de la tarjeta CRC para su clase, y podrá comunicarse con sus colaboradores por medio del paso de mensajes siguiendo el orden de ejecución establecido por el funcionamiento simulado.

1.2 ViRPlay3D 2

ViRPlay3D es un juego desarrollado para simular las características principales del role-play. Tal y como se observa en la imagen, el jugador es capaz de trasladarse a un escenario donde se encuentran los actores para observar cómo se produce la comunicación entre éstos siguiendo el guión establecido para el escenario.

El juego está desarrollado en un entorno 3D donde los objetos están representados por figuras animadas. Además de los objetos, en el escenario también están representadas las tarjetas CRC de las clases con forma de caja detrás de cada uno de ellos. De esta forma, en el entorno de ViRPlay3D podemos considerar estas tarjetas CRC como una especie de "actor" ya que son las que van a proporcionar a los objetos la información sobre su comportamiento.

El jugador puede ver la ejecución del escenario con los diferentes pasos de mensaje que se intercambian los objetos uno detrás de otro. Éstos están representados como el lanzamiento de una pelota entre los objetos implicados.

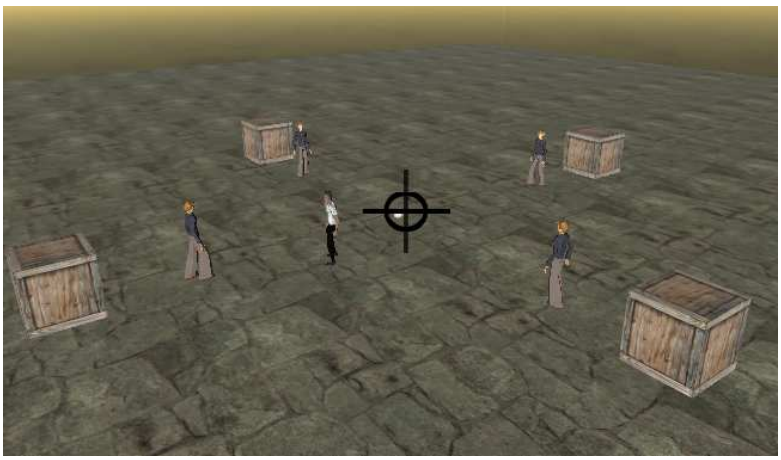

Figura 1. Representación de un escenario en ViRPlay3D

El principal problema que tiene ViRPlay3D es que está orientado a juego monojugador, desaprovechando una de las principales características del role-play, que es dar al jugador la posibilidad de participar en el escenario, no como un simple observador, sino como un objeto cuyas acciones influyen a la

hora de ejecutar correctamente el guión del escenario. De esta manera, el jugador aprenderá el funcionamiento de la parte de aplicación que se está representando cuando realice las acciones adecuadas.

Extendiendo el juego con un modo multijugador, no solo permitiendo al jugador manejar un objeto sino dando la posibilidad de que participen varias personas, completa el objetivo principal del role-play. Varias personas juntas colaborando entre sí para aprender un mismo escenario a la vez, donde todos los jugadores participan activamente en el desarrollo del mismo.

En el próximo capítulo, vamos a plantear la situación inicial de la que partió el proyecto, explicando más en detalle las principales funcionalidades que estaban implementadas en ViRPlay3D. De esta manera tendremos una base para definir cuáles fueron los objetivos de este proyecto a la hora de incluir el entorno multiusuario como un modo de juego. Una vez que se marquen los objetivos principales, se mostrara la planificación llevada a cabo durante el transcurso del proyecto.

2 *Situación Inicial de Partida*

En un primer momento disponíamos de una versión inicial de ViRPlay3D, orientada a jugador monousuario y con la arquitectura principal del juego definida. Lo primero que hicimos fue familiarizarnos con el código que ya había desarrollado, intentando centrar nuestros esfuerzos en encontrar las partes del programa que necesitaríamos conocer en profundidad para la inclusión del juego en red o multiusuario.

La parte más desarrollada en esta versión inicial, era la representación grafica de los diferentes aspectos de un único escenario de role-play, como son los objetos y las tarjetas CRC. También se contaba con una figura que representaba al jugador, y que por tanto, podía ser manejada por éste. Durante el juego, se tenía la posibilidad de simular el role-play del escenario, y ver las animaciones del paso de mensajes. La aplicación ya disponía de los aspectos principales de cualquier juego, como la navegación por menús para iniciar o terminar la partida, mover al jugador, o incluso cambiar de cámara.

Tras las primeras tomas de contacto nos dimos cuenta de que una parte del juego estaba ya implementada, sobre todo el funcionamiento global de la aplicación y la representación grafica. Tal como nos explicarían posteriormente los tutores, en nuestro trabajo de ampliación para introducir el juego multiusuario, no tendríamos que modificar prácticamente nada de la parte gráfica del juego. Nos centraríamos principalmente en la parte lógica, ya que, como se explicará más adelante, la arquitectura desarrollada separaba perfectamente los aspectos gráficos del juego de la lógica que controlaba la simulación.

2.1 Características funcionales de la versión inicial

2.1.1 Representación de escenarios

El juego sólo representaba un único escenario, pero éste era cargado directamente de un archivo, y según el contenido

de éste, se creaban los elementos gráficos que luego aparecían en pantalla. Así que, para disponer de nuevos escenarios, sólo necesitaríamos un archivo de texto con la información del escenario codificada en XML con una estructura previamente definida por los tutores y antiguos desarrolladores de ViRPlay3D. Por tanto la creación grafica y lógica de los actores del role-play estaba ya implementada, lo que implica que no tendríamos que preocuparnos de la manera en que se creaban las entidades 3D que los representaban en el escenario.

Los objetos que interactuaban en el escenario estaban representados por una figura humana, y las tarjetas CRC de cada uno de estos objetos detrás de ellos con la forma de una caja de madera. Los objetos tenían una animación permanente pero no cambiaban salvo que se vieran involucrados en un paso de mensaje establecido por el role-play que se estaba jugando. Se observaba el movimiento de los objetos pertinentes encarándose y lanzándose la pelota que representaba el mensaje ejecutado.

2.1.2 Movimiento y animación del jugador

Existía en el juego otra figura que representaba al jugador que en ese momento estaba observando la partida en curso. El usuario controlaba por tanto los movimientos de esta figura pudiendo desplazarse por todo el escenario. Además se le indicaba el nombre de los actores si estaba suficientemente cerca de ellos, y si lo deseaba podía consultar la información asociada al objeto.

Por tanto, para nosotros sería transparente el sistema de interacción entre el usuario y el programa, es decir como detectaba las diferentes pulsaciones del teclado y en consecuencia el jugador se movía por el escenario. Aunque en un futuro se podría manejar a cada uno de los objetos el sistema de movimiento de las entidades 3D estaba implementado.

La visión predeterminada por parte del jugador era en primera persona, aunque existía la posibilidad de cambiar la cámara a un modo libre donde se podría ver al jugador desde cualquier punto andando por el escenario. Por tanto el manejo de la cámara podía quedar fuera de nuestras responsabilidades.

2.1.3 Pasos de mensaje

El jugador era el encargado de ejecutar los diferentes pasos de mensaje del escenario cuando pulsaba una tecla determinada. Aunque se podían ver las animaciones de dos pasos de ejecución en el escenario de la versión inicial, no ocurría lo mismo que con los escenarios, estos pasos no eran cargados desde un archivo. Las diferentes acciones estaban programadas directamente en el código para funcionar exclusivamente con el único escenario del que por ahora se disponía, y por tanto el único en el que se podría jugar.

Este es uno de los aspectos en el que deberíamos trabajar para dotar al juego de la mayor flexibilidad a la hora de ejecutar cualquier guión de role-play que se quisiera sobre el escenario que tuviera asociado.

2.1.4 Información sobre los actores

Otra posibilidad que se le ofrecía al jugador, era la de consultar la información que tenia asociada cada uno de los actores del role-play, como por ejemplo las responsabilidades y los colaboradores de una clase establecidas en su tarjeta CRC. Sin embargo, la versión inicial solo tenía una interfaz provisional que mostraba siempre la misma información, ya que no había sido implementado un sistema que obtuviera la información para los actores del escenario al iniciar la partida.

Debíamos por tanto añadir al juego la carga, almacenamiento y consulta de la información asociada a los diferentes actores del escenario que se estuviera jugando.

2.2 Características Tecnológicas

2.2.1 Arquitectura

En este apartado se detallará como era la arquitectura del proyecto en la situación inicial, justo antes de cogerlo nosotros. En la figura 2 podemos ver un esquema de la

arquitectura del proyecto. La arquitectura se divide principalmente en tres módulos:

- **Aplicación**: Se podría ver como el motor del juego. Es una máquina de estados que se encuentra continuamente en un bucle actualizando toda la partida.
- **Lógica**: Contiene todo el control de las reglas del juego. Es un mundo basado en casillas en dos dimensiones donde se encuentran los actores del escenario. Las actores pueden ser de dos tipos: inmóviles llamadas clases, o con posibilidad de movimiento, llamadas avatares. Los avatares a su vez pueden ser objetos o el jugador. La lógica se comunica con la aplicación por medio de comandos, y con la parte gráfica por medio de observers.
- **GUI** (Graphical User Interface): Este módulo es responsable tanto de la gestión del dibujado de la escena en 3D como de la interfaz de usuario en 2D. Para el primero, existe un gestor de GUI encargado de crear, almacenar, dibujar y eliminar todas las entidades visuales de la escena. Existirá una entidad gráfica por cada una de las entidades lógicas y aquéllas permanecerán a la escucha de los eventos provocados por las entidades lógicas para actualizar convenientemente la escena en 3D. Para la gestión de la interfaz de usuario en 2D, se ha usado Nebula 2, un motor gráfico que proporciona una interfaz de scripting en Tcl. Esta interfaz es bidireccional, de modo que el módulo de Aplicación puede modificar los menús e interfaces de usuario y ésta puede comunicar al módulo de Aplicación las acciones realizadas por el usuario.

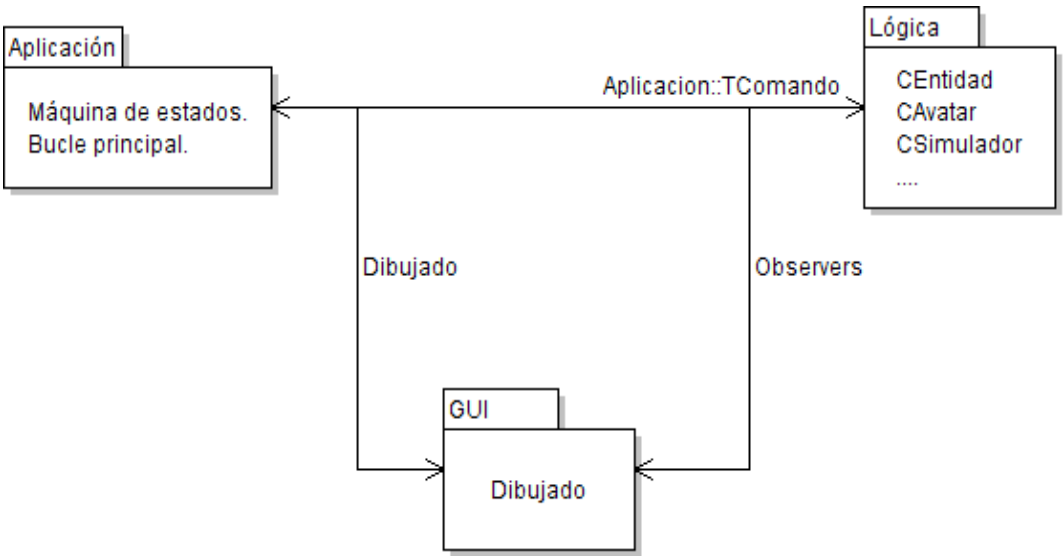

Figura 2. Esquema de paquetes de ViRPlay3D2 en la situación inicial

Ahora vamos a entrar un poco más en detalle en cada paquete y poder comprende el funcionamiento de ViRPlay3D2.

Paquete aplicación:

Como se expuso anteriormente, la aplicación es una máquina de estados jerárquica, gestionada mediante un patrón State, de la cual heredan los principales estados de la aplicación, tales como los menús principales o el estado de juego. Este estado merece mención especial, ya que de él dependen la mayoría de las cosas que ocurren durante la partida, por eso este estado es a su vez otra máquina de estados, de la cual heredan los estados internos del juego, tales como el estado para ver un inventario, el estado donde los personajes se mueven etc...

En la figura 3 podemos ver un diagrama UML de la máquina de estados de la aplicación. En la figura 4 tenemos el diagrama de estados.

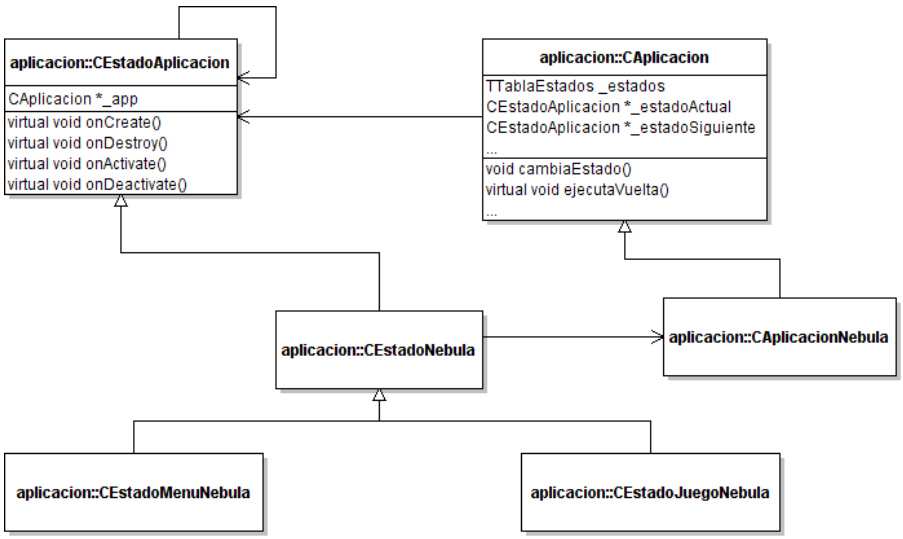

Figura 3. UML resumido de la máquina de estados del paquete de aplicación.

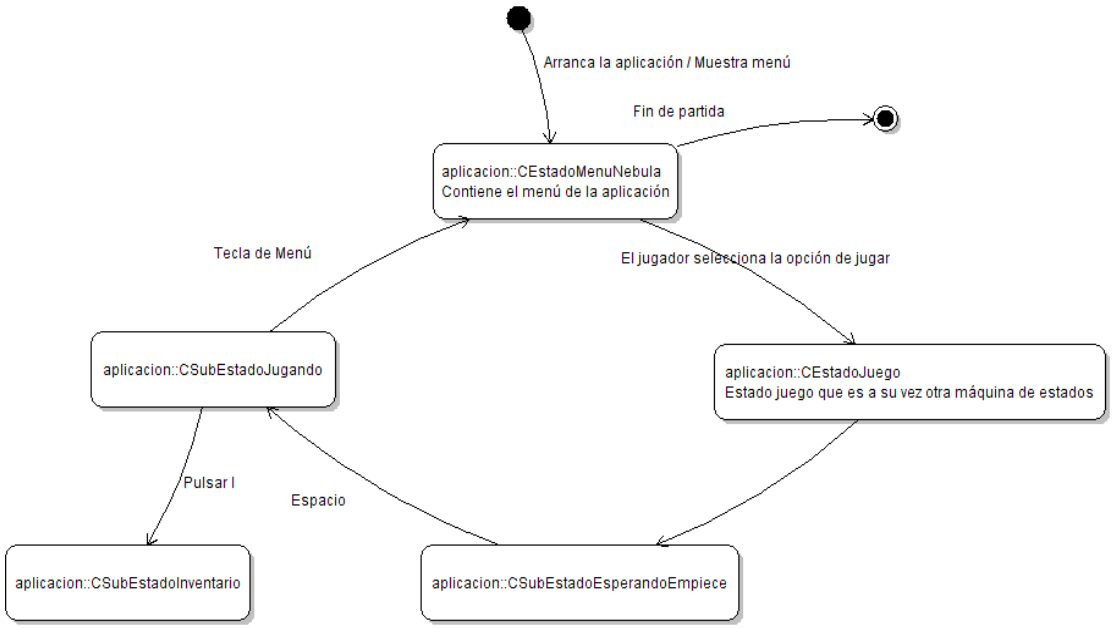

Figura 4. Máquina de estados simplificada gestionada en el paquete aplicación

Paquete lógica:

En él se encuentran todas las entidades del juego. Hay que tener en cuenta que aunque el juego es en 3D toda la parte lógica se gestiona en 2D, es el GUI el encargado de calcular la tercera coordenada. En el juego, se distinguen como entidades lógicas principalmente: el jugador, las tarjetas CRC o clases y los

objetos, también tienen mucha importancia las clases simulador y mundo, que son las que reciben los comandos de la GUI y los procesan para modificar la lógica de la manera oportuna. La clase mundo agrupa la información de todas las entidades de la partida, es como una clase contenedora de ellas.

En la figura 5 se puede ver un esquema detallado de las clases del paquete lógica.

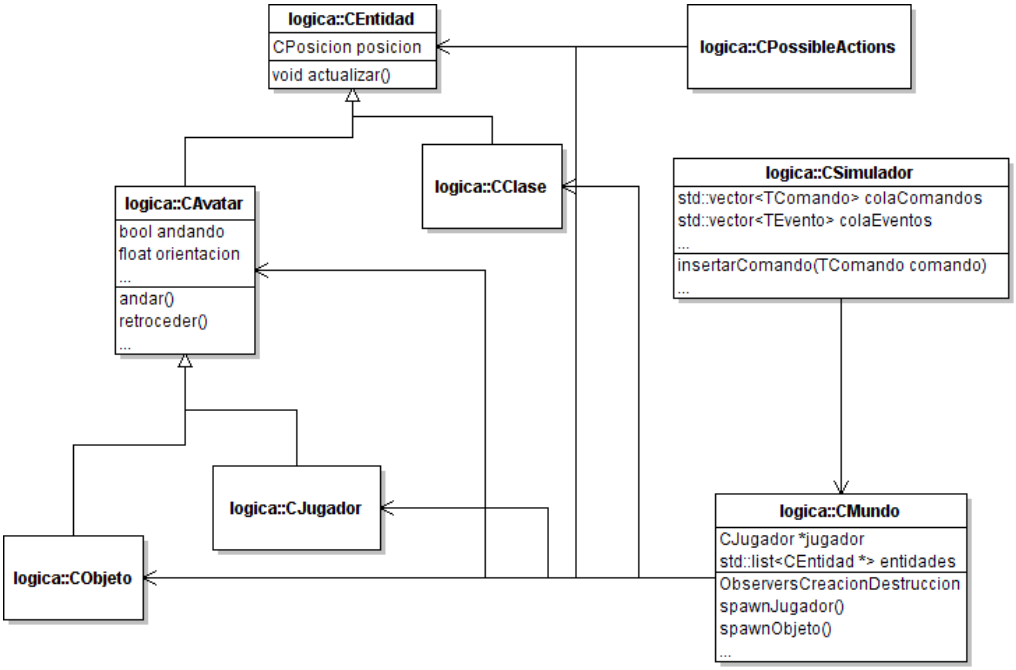

Figura 5. UML resumido del paquete lógica

Paquete GUI:

Se distingue principalmente la clase NentidadGUI, que representa la parte gráfica de las entidades de la lógica, de ella se distinguen dos tipos principales, uno para las entidades sin movimiento, como las clases, y otro para las entidades animadas, como los avatares o el jugador. Se observa también como se comunica con la lógica por medio de los observers.

El GestorAnim era una clase que cableaba el lanzamiento gráfico de la bola, en la ampliación esta clase ha desaparecido, ya que la pelota ahora es una entidad lógica más y se la trata como a todas.

En la figura 6 se muestra un esquema de las clases del paquete GUI.

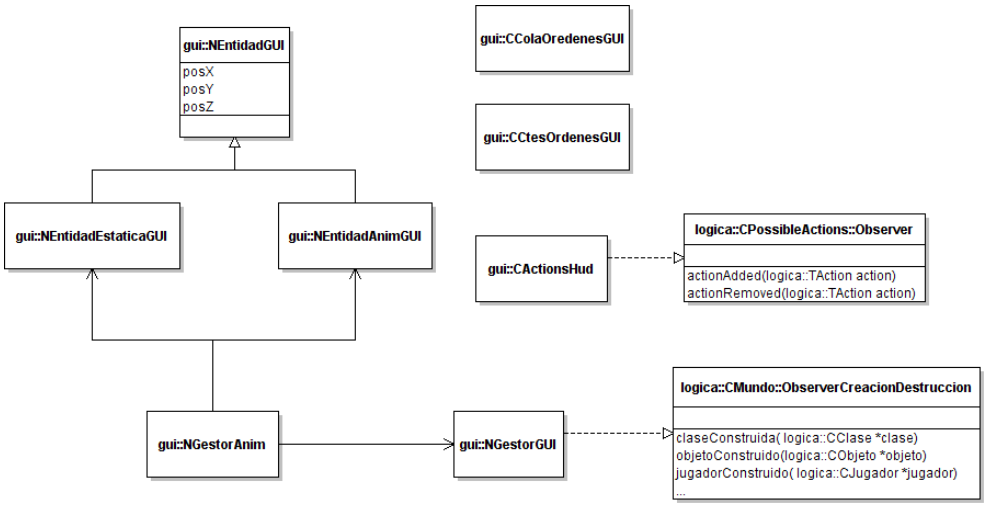

Figura 6. UML resumido del paquete GUI

- ***Coordinación GUI-Lógica: Patrón Observer***

Una parte muy importante del proyecto y que posteriormente fue vital para el desarrollo del juego multijugador es el mecanismo de observers que está montado en la arquitectura, como su propio nombre indican, hacen uso del patrón observer (Ver Figura7) para coordinar la lógica y la gráfica del juego.

El patrón Observador también conocido como "spider" define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, el observador se encarga de notificar este cambio a todos los otros dependientes. El objetivo de este patrón es desacoplar la clase de los objetos clientes del objeto.

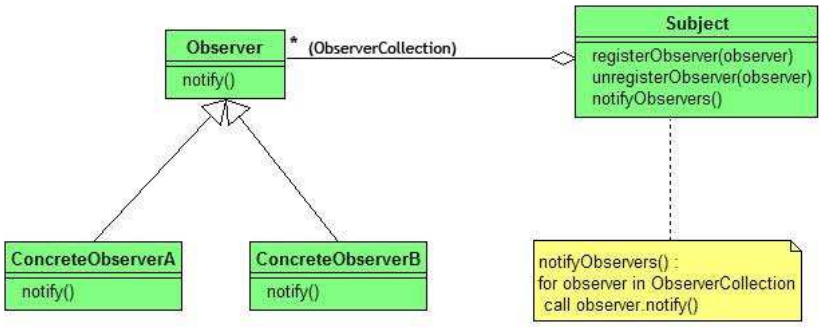

Figura 7. Patrón Observer

Para implementarlo, todas las clases de la lógica que quieran notificar de sus cambios, tendrán una clase interna observer, de forma que cuando se modifiquen, informarán a todas las clases que hayan implementado su observer interno. A continuación mostramos una imagen del mecanismo de observers que hay montado:

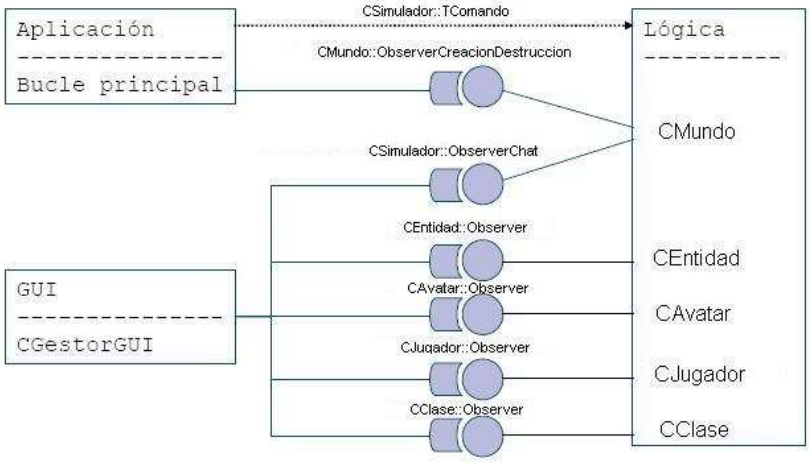

Figura 8. Arquitectura de observers

- ***Librerías graficas***

ViRPlay3D2 usa varías herramientas para su implementación. Éstas son, entre otras, DirectX, Nebula 2 y TCL. Se usan para la parte gráfica de modo que se tenga un motor de desarrollo para los personajes y las animaciones del juego. Las interfaces gráficas, como los menús e inventarios, están

desarrolladas en el lenguaje TCL de forma que junto con Nebula 2 se crea todo el entorno que vemos al ejecutar el juego.

3 *Objetivos*

El trabajo que se ha realizado durante este proyecto ha sido el de incluir en el juego la posibilidad de ejecutar el role-play en un entorno multijugador, donde cada uno de los jugadores puede participar activamente en el desarrollo del escenario.

Por otra parte, aunque a primera vista parecía que no quedaba mucho por hacer en la versión monousuario de ViRPlay3D, ya que se podía ver la ejecución de un escenario, no todas las funcionalidades estaban terminadas por completo.

3.1 Necesidades en el juego monojugador

Una vez que conocíamos las características principales así como el funcionamiento global de la versión inicial de ViRPlay3D de la que partimos, surgieron una serie de funcionalidades que debían ser, o bien completadas, o bien incluidas antes de la implementación del juego multiusuario en nuestro programa. Estas funcionalidades están enfocadas a permitir la ejecución de diferentes escenarios de role-play, así como la ejecución de guiones para el escenario.

3.1.1 Carga y almacenamiento de información

Durante la ejecución de un guión de role-play, es importante conocer que tarea desempeña cada objeto, a través de sus responsabilidades y colaboradores descritos en la tarjeta CRC correspondiente. También se debe poder consultar el estado del propio objeto tras producirse un paso de mensaje en la simulación del escenario. Es decir, el jugador del escenario debe poder consultar esta información en cualquier momento de la partida. En la versión inicial de ViRPlay3D, cuando se realizaba la acción de mirar a cualquier actor del role-play, se mostraba una interfaz con un ejemplo de lo que podía ser la información de una tarjeta CRC. Esta información era siempre la misma, y no correspondía con el actor (objeto o tarjeta CRC) del que se estaba consultando la información.

Se necesitará por tanto añadir al juego un sistema que recupere, almacene y permita la consulta de la información de los diferentes actores que participen en el escenario que se juega, y permitir su consulta durante el transcurso del mismo. Para esto, dispondremos de una serie de ficheros que contendrán esta información codificada en XML y se cargará durante la creación del escenario.

Además habrá que tener en cuenta que la información varía para cada uno de los actores del juego, así que la información se presentará de una manera u otra adaptándose al contenido de la información que se almacena de cada tipo de actor. En la figura 7 se puede ver el contenido de las informaciones de una tarjeta CRC y un objeto.

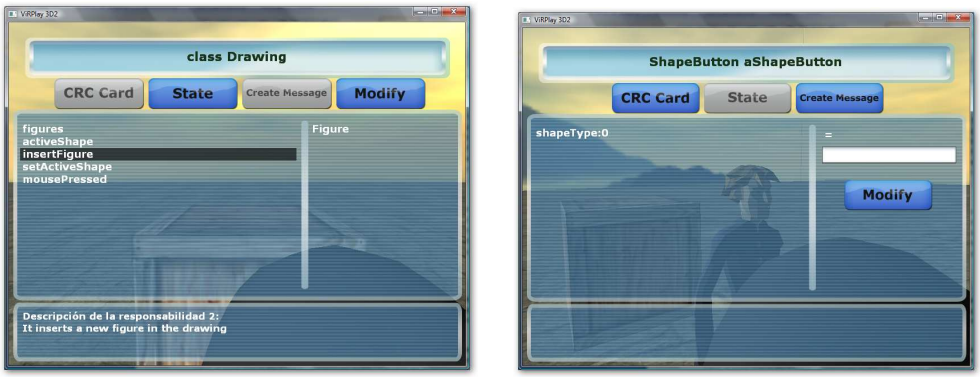

Figura 9. Representación de la información de los actores.

3.1.2 Ejecución de diferentes escenarios y guiones de role-play

Se dará al jugador la posibilidad de jugar en diferentes escenarios definidos previamente. Ya que la versión inicial de ViRPlay3D cargaba el escenario desde un archivo, también representada en formato XML, para disponer de nuevos escenarios, solo se necesitarían los archivos con la información codificada de la manera adecuada.

Un aspecto fundamental en la simulación del role-play es el guión que deben ejecutar los objetos que participan en el escenario. En este guión se definen los diferentes pasos de mensaje que se producen a lo largo de la ejecución del role-

play, permitiendo al usuario comprender como es el comportamiento del sistema que se está simulando.

Información sobre el role-play

En la versión inicial con la que empezamos el proyecto solo se podía ejecutar un guión concreto para el escenario por defecto. Es decir, no se cargaba ninguna información sobre el guión que se iba a seguir, simplemente existía un modulo en el programa que se encargaba de mover las figuras 3D para que se produjera un paso de mensajes entre dos objetos. La siguiente figura representa un paso de mensaje.

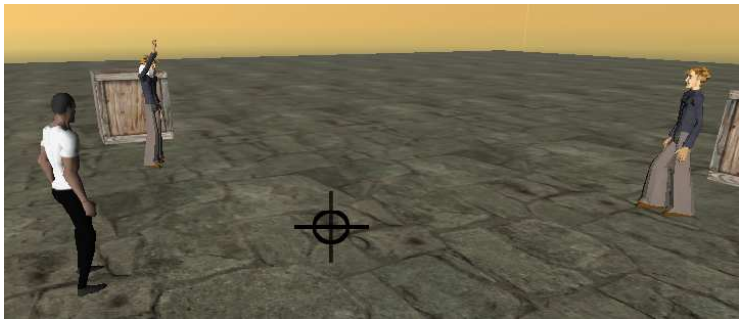

Figura 10. Paso de mensaje entre dos objetos.

Sin duda, será necesario añadir a nuestro sistema la posibilidad de cargar, ya no solo cualquier escenario, sino cualquier guión que defina el comportamiento de los objetos dentro del escenario. Esto dará a los usuarios de ViRPlay3D 2 la posibilidad de crear un escenario y un guión para representar la interacción que se produce entre los objetos de su sistema y así poder comprender y explicar mejor el funcionamiento de ciertas partes de éste.

Para ello se tendrá otro fichero con la información sobre el guion que se debe ejecutar en el escenario que se vaya a jugar. Al igual que el resto de ficheros la información estará codificada en XML. Ésta se cargará durante la creación del escenario, y marcara el orden de los pasos de mensaje que se produzcan durante la ejecución del mismo.

Gestión y ejecución de los mensajes

Una vez que el juego pueda tener la información sobre el role-play almacenado, se necesitara implementar la posibilidad de ejecutar estos pasos de mensaje de manera ordenada con las animaciones adecuadas representando el movimiento correspondiente.

Una cuestión importante a tener en cuenta es que los diferentes mensajes que se producen a lo largo de la simulación, pueden variar el estado de cualquiera de los objetos del juego. Por tanto lo más interesante es que el jugador decida en qué momento se ejecutan, ya que lo más probable es que quiera consultar la información de un objeto para conocer los cambios que se han producido tras lanzar el último mensaje.

Además se dará al usuario la posibilidad de consultar la información del último paso del role-play para saber por ejemplo quien fue el que lanzo el mensaje al objeto que esté en la posesión de la pelota o cual fue el mensaje que se le lanzó.

Como alternativa a la carga de la información del role-play, se permitirá al usuario crear mensajes mientras está jugando a través de una pantalla en la que definirá el contenido del mensaje a ejecutar.

Para finalizar, se almacenará en un fichero la información sobre el último guión ejecutado.

Representación grafica del paso de mensajes

Este punto será uno de los pocos en los que tengamos que desarrollar funcionalidades del apartado grafico del juego. Sera necesario encontrar una manera para representar la ejecución de los pasos de mensaje del role-play de tal modo que se pueda lanzar un mensaje desde un objeto a cualquier otro, diferenciar si el mensaje que se pasa es de llamada o de retorno, y mostrar en pantalla la descripción de éste.

3.2 Adaptación al juego en red

Una de las principales características de ViRPlay3D2 es su adaptación al modo multijugador. Con esto se consigue explotar el potencial de ViRPlay3D de forma que se permite la interacción entre varios usuarios en tiempo real y desde distintas máquinas.

Todas las funcionalidades descritas anteriormente para el modo monojugador, se extienden ahora al modo en red. De esta forma se aumenta la jugabilidad y el sentido del juego ya que los objetos del escenario pasan a ser ahora usuarios que actúan con libertad propia.

Con esto se consigue que los usuarios realmente puedan participar en el juego, interviniendo de una forma directa en el role-play, de modo que cada usuario tiene que representar un objeto de los que componen el escenario. De esta forma, el usuario no es un mero observador del juego, si no que interactúa con él, de modo que sus acciones influyen sobre el desarrollo de éste.

Para adaptar el juego al modo multijugador se pensaron en dos tipos de de arquitectura para comunicación en red: Arquitectura Cliente-Servidor y Arquitectura Peer-to-Peer (P2P). Ambas tienen sus ventajas e inconvenientes que se detallan a continuación:

Cliente-Servidor	Peer-to-Peer
Mayor Latencia	Menor Latencia
Cliente: Menos ancho de banda	Cliente: Más ancho de banda
Mayor escalabilidad	Menor escalabilidad
Más seguridad anticheat	Menos seguridad anticheat
Imposibilidad de Host migration	Posibilidad de Host migration
Necesario servidor más potente	Servidor más sencillo
Arquitectura algo más compleja	Arquitectura algo más simple
Menos problemas de NAT	Problemas de NAT

En el caso de ViRPlay3D2 se ha optado por una arquitectura Cliente-Servidor con lógica centralizada, esto implica que:

- El servidor es el único que ejecuta la lógica del juego.
- Los clientes solo deben mandar comandos (como por ejemplo de movimiento) para comunicarse con el servidor.
- Los clientes solo pintan. Para ello, el cliente debe tener localmente reflejado lo que contiene la parte lógica del servidor, sin embargo, sus entidades lógicas no las controlará el cliente, sino que recibirá por parte del servidor las órdenes necesarias para actualizarlas. Este tipo de lógica se conoce como lógica puppet (marioneta).
- Sincronización y coherencia elevada.

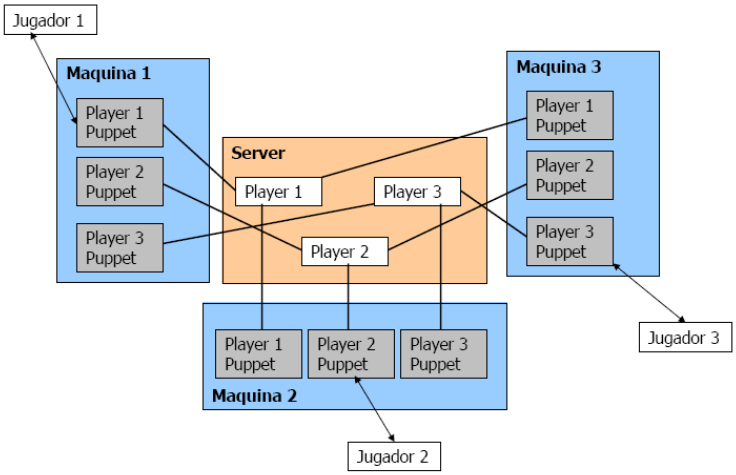

Figura 11. Arquitectura Cliente-Servidor usando lógicas marioneta

Como se puede ver en la figura 11, los clientes se conectan a una máquina servidora que es la que gestiona toda la lógica. Se tienen tantos usuarios como objetos tenga el escenario de modo que cada uno de estos tiene su información propia, es decir, la información del avatar que lo representa dentro del mundo. Esta información se puede visualizar en el juego, al igual que se hacía en monojugador, mediante el sistema de inventarios.

Los usuarios además de consultar la información que deseen, pueden comunicarse entre sí mediante una herramienta de comunicación o chat. Con esta herramienta se permite que cada usuario esté en contacto con el resto de jugadores permitiéndole comunicarse con ellos en cualquier momento. Cada vez que la partida llegue a su fin, el chat quedará

guardado en un archivo con formato XML para poder así consultar el historial de mensajes de la partida.

4 *Planificación*

Debido a que ViRPlay3D es un proyecto que ya estaba empezado, inicialmente hubo una fase previa de familiarización con la herramienta. La principal funcionalidad que se quería integrar era el modo multijugador, por lo que había que documentarse sobre ello. Para simplificar esta tarea lo primero fue documentarse con juegos de menor envergadura que funcionaban en red. Esta fase inicial fue lenta ya que apenas teníamos experiencia en el manejo de juegos en red. Una vez aprendido el funcionamiento de la librería de red que íbamos a usar (eNet), y ya conociendo más en profundidad el diseño de clases de ViRPlay3D se comenzó el desarrollo e implementación del juego. Para ello se han usado varias estrategias:

- Scrum.
- Reuniones semanales.
- Control de versiones.

4.1 Scrum

Como se ha detallado en el objetivo del proyecto no ha consistido sólo en introducir la red, si no que ha habido una serie de funcionalidades extra que se han ido integrando al juego. Parte de estas funcionalidades se fueron realizando mediante una metodología ágil de trabajo, conocida como Scrum.

Scrum es una metodología para la gestión de proyectos. Consiste en definir una serie de funcionalidades que se piden a una aplicación. Estas funcionalidades se dividen en tareas más sencillas que forman lo que se conoce con el término sprint o iteración. Un sprint consiste en marcar ciertos objetivos cada cierto tiempo, de forma que cada vez que se cumplan los plazos (habitualmente un mes), los objetivos marcados tienen que estar completados. Alguna de las funcionalidades finales de la aplicación se han realizado independientemente de sprints, pero la mayoría de ellas se han llevado a cabo siguiendo un orden preestablecido.

EL proyecto se ha desarrollado en cuatro fases:

La primera fase fue de primer contacto con el entorno. Partíamos de un prototipo ya hecho, ninguno teníamos experiencia en el tema de programación de videojuegos, y tuvimos que emplear un par de meses en entender bien cómo funcionaba todo para más adelante poder implementar sobre él. Como esta fase era de toma de contacto, en ella no se realizó ningún sprint.

En la segunda ya se empezó con implementación, comenzando por extender la funcionalidad en juego monojugador, con la intención de ir añadiendo características para que más tarde pudieran aprovecharse en la versión multijugador. Esta fase se corresponde con los sprints primero y segundo.

En la tercera fase (sprint 3), se cambió la arquitectura de la aplicación, de forma que todo lo que antes valía para un sólo jugador, quedara extendido para que muchos jugadores pudieran jugar a través de red local.

En la cuarta (sprint 4), se corrigieron todos los fallos y bugs que fueron quedando de los sprints anteriores, añadiendo en ocasiones pequeños detalles de implementación. También se dedicó a la realización de esta memoria de trabajo.

Todas las ampliaciones se han desarrollado en cuatro sprints que son:

4.1.1 Sprint 1:

REQ 1. Lanzar y recoger la bola

1.a. Descripción:

Se pretende que el objeto que sostiene la bola sea capaz de lanzar la bola de acuerdo a la información contenida en un mensaje.

Un mensaje contiene información sobre el actor que lanza la bola, llamado invocador, el actor que la recibe, llamado invocado, el nombre del método a invocar, la lista de parámetros, el valor devuelto (si existe) y el tipo del mensaje (invocación o retorno).

1.b. Actividades:

1. Crear la clase de los mensajes.
2. Crear la pelota como entidad lógica. La pelota contiene información sobre el mensaje que actualmente va a ser invocado, quién la sostiene (si la sostiene alguien), cuando es lanzada sabe cómo ir moviéndose hasta su destino, e informa a otras clases sobre: cuándo es lanzada, cuándo es recogida y cuándo cambia de posición.
3. Un objeto ha de saber lanzar una bola hacia otro objeto y ha de saber recoger la bola cuando está próxima a él. Para recoger la bola hay que comprobar que el objeto que la pretende recoger está lo suficientemente cerca para poder recogerla.
4. Es necesario coordinar las acciones a nivel lógico con las acciones visuales. Por tanto habrá que crea un nuevo gestor de animaciones que se encargue de esta tarea.

REQ 2. Ejecutar de manera automática un escenario.

2.a. Descripción:

A partir de la información contenida en un XML, se va a simular la información de un escenario completo.

2.b. Actividades:

1. Carga del XML. El escenario se almacena como un archivo XML que contiene los pasos de ejecución y las modificaciones producidas en el estado de los objetos tras dicho paso. Cada paso ha de traducirse a un mensaje que será usado para lanzar la pelota.
2. Crear un gestor para el role-play. El gestor se encuentra en la parte de la lógica y es el responsable de decidir cuál es el siguiente paso de ejecución de la simulación. Guarda la lista de todos los pasos de la ejecución (obtenida a partir del XML) y actualiza el estado de los objetos.

3. Proporcionar una manera para que el usuario pueda ir ejecutando paso a paso la simulación.
4. Comunicar al usuario cuándo ha concluido la ejecución de un escenario, presentando un mensaje por pantalla.

REQ 3. Visualizar el contenido de la pelota

3.a. Descripción:

Si se mira a la pelota se puede ver el contenido de su inventario. Éste contendrá la información referente al mensaje actual que está siendo invocado o al último invocado.

3.b. Actividades:

1. Crear la interfaz gráfica del inventario de la pelota y proporcionar los métodos para cargar la información de la misma.
2. Ampliar el sistema de inventarios para que se pueda ver la información de la bola.
3. Modificar la clase que representa la pelota en la lógica para que se permita su visualización.
4. Modificar la ejecución de la acción Mirar a. Actualmente solo se puede mirar simultáneamente a una entidad. Cuando un objeto mire a otro que sostiene la bola, se verán ambas entidades. Actualmente, en la pantalla aparecen las acciones numeradas. Usar este número para invocar la acción correspondiente.

REQ 4. Visualizar el estado de un objeto

4.a. Descripción:

A través del inventario del objeto, podemos ver el estado del mismo.

4.b. Actividades:

1. Hacer que los objetos puedan contener información sobre su estado o crear un repositorio de información sobre el estado de los objetos.
2. Crear la interfaz gráfica para ver a través del inventario del

objeto, el estado del mismo. Además, generar el evento de la GUI que permita pasar a este nuevo inventario. Para este inventario también hay que proporcionar los métodos que permitan incluir la información del mismo.

3. Ampliar el sistema de inventarios para que se pueda ver la información del objeto.
4. Inicializar los objetos con la información contenida en el XML del escenario.
5. Hacer que el gestor de role-play actualice la información de los objetos de acuerdo a la información contenida en el XML.

REQ 5. El usuario construye el contenido del mensaje de una bola

5.a. Descripción:

A través del inventario de un objeto, se construye el mensaje que va a ser invocado.

5.b. Actividades:

1. Crear la interfaz gráfica para la construcción de un mensaje. En ella el usuario selecciona el objeto al que va a invocar (el invocador ha de estar fijo porque es el que sostiene la bola), el método que va a invocar (ha de pertenecer al objeto invocado y se actualizará dependiendo de éste), el tipo del mensaje, los parámetros reales y el valor devuelto, si el mensaje es de tipo retorno. La interfaz puede solicitar información a la lógica para actualizar la información presentada al usuario. Tras aceptar la creación, la lógica será informada del mensaje que se va a crear.
2. Modificar el gestor role-play para que sea el responsable de ir guardando cada uno de los pasos de mensaje que se vayan produciendo. Tras cada mensaje creado se ha de mostrar la simulación completa.
3. Antes de terminar la aplicación hay que generar un XML que guarde lo que se ha simulado. El formato del XML es el mismo que el de la carga de escenarios.

4.1.2 Sprint 2:

REQ 6. Modificar el estado de un objeto

6.a. Descripción:

A través del inventario de un objeto, se puede modificar el estado de un objeto

6.b. Actividades:

1. Interpretar los comandos de cambio de estado que envía la interfaz del inventario del objeto y cambiar la información en el repositorio de información.
2. Actualizar la interfaz del inventario debido al cambio producido en la lógica. El repositorio de información ha de avisar de esos cambios al inventario lógico y éste ha de notificarlos al inventario GUI.

REQ 7. Selección de escenarios

7.a. Descripción:

Tras pulsar la opción de jugar aparecerá un nuevo menú en el que se podrá seleccionar el escenario que se desea cargar. Tras seleccionarlo, se cargará el escenario y se ejecutará dicho escenario.

7.b. Actividades:

1. Crear la interfaz gráfica para la selección del escenario.
2. Reestructurar la información de un escenario: Todos los escenarios se guardarán en la carpeta data/escenarios. Dentro de esta carpeta aparecerá una subcarpeta por cada escenario. El contenido de estas subcarpetas será el siguiente:
3. El archivo del escenario (XML)
4. La información de clases (XML)
5. La información de objetos y de role-play (XML)
6. Un fichero con la descripción larga del escenario que aparecerá al comienzo de la partida.

7. Un fichero de configuración, que de momento contendrá:
8. Nombre del escenario
9. Descripción corta del escenario
10. Fichero de descripción larga
11. Fichero de escenarios
12. Fichero de info de clases
13. Fichero de info de objetos y role-play
14. Modificar la clase Simulador para que cargue el escenario seleccionado en lugar de uno por defecto.
15. Modificar el script correspondiente para la presentación por pantalla de la descripción larga del escenario.

En este punto finaliza la ampliación monojugador, y comienza el cambio de arquitectura para adaptar el juego en red.

4.1.3 Sprint 3:

REQ 8. Extensión del juego a multijugador

8.a. Descripción:

Ampliar la arquitectura de forma que todas las funcionalidades disponibles en la versión monojugador, estén disponibles en la versión multijugador.

8.b. Actividades:

1. Conseguir establecer una conexión cliente servidor.
2. Modificar los menús gráficos para poder mostrar la opción de modo multijugador. Tendrá que haber una opción para inicializar los clientes y otra para el servidor.
3. Posibilidad de pasar comandos por red. Ha de ser posible mover a un solo jugador y realizar con él las mismas acciones que se podían realizar en la versión monojugador, es decir, poder moverse, ver inventarios etc...
4. Conseguir lo mismo que en el anterior punto pero cada jugador con un personaje distinto.

REQ 9. Selección de escenarios en modo multijugador

9.a. Descripción:

Extender la herramienta de selección de escenarios al modo multijugador.

9.b. Actividades:

1. Reutilizar la herramienta de selección de escenarios desde el menú de servidor.
2. Cada vez que se entra en modo servidor aparecerá un menú gráfico con la opción de seleccionar un escenario.
3. Una vez que en el servidor se haya seleccionado un escenario se tiene que comunicar su información por red a todos los clientes que se conecten.

REQ 10. Selección de roles

10.a. Descripción:

Diseñar una herramienta que permita la selección de roles de un escenario concreto previamente seleccionado.

10.b. Actividades:

1. Crear la interfaz gráfica para la selección de roles.
2. Cada vez que se entre en modo cliente aparecerá una interfaz gráfica para elegir el personaje con el que se va a jugar.
3. Esta Interfaz contendrá una serie de personajes según el escenario que se haya seleccionado en el servidor.
4. Gestión de los personajes: Cada vez que un cliente seleccione un personaje para jugar, se le comunica al servidor que es el que se encarga de gestionarlo todo. Si el personaje está libre se lo asigna al cliente, comunicándoles a todos los demás que el cliente ha sido seleccionado (Esto se indica mediante el símbolo "*" al lado del identificador del personaje).

5. Cuando todos los personajes se seleccionan se puede comenzar la partida.

REQ 11.	Herramienta de comunicación

11.a. Descripción:

Diseñar una herramienta que permita la comunicación a modo de chat entre los personajes que entren en juego en una partida.

11.b. Actividades:

1. Crear la interfaz gráfica para la herramienta de comunicación.
2. Cada vez que un cliente pulse la tecla TAB se entra en la herramienta de comunicación, donde puede escribir mensajes que se mandarán por red a todos los clientes conectados al servidor.
3. Cada vez que se abra la herramienta tendrá que aparecer el historial de mensajes almacenados.
4. Una vez que se termine la ejecución de la partida se guardará todo el historial de mensajes enviados en un archivo en formato XML.

4.1.4 Sprint 4

Este sprint ya no tiene requisitos detallados como los anteriores, ya que se dedicó principalmente a la corrección de fallos y bugs que iban quedando atrás de los anteriores sprints. Muchas de estas correcciones no eran fallos en sí, si no que era una nueva funcionalidad que no se había implementado en su totalidad, o que no funcionaba como se esperaba que lo hiciera.

Este sprint también se ha dedicado a la realización de esta memoria de trabajo en la que se detallan todas las ampliaciones del proyecto.

4.2 Reuniones semanales

Desde el inicio de ViRPlay3D se han ido sucediendo continuas reuniones con los tutores del proyecto, de forma que nos guiaran en el desarrollo de la implementación del juego.

Inicialmente las reuniones fueron para aportarnos documentación y ayudarnos en la comprensión tanto de ViRPlay3D como en la parte inicial de la adaptación de la red.

Después de estos inicios las reuniones se establecieron semanalmente de forma que así se podían consultar las posibles dudas que aparecieran en un corto periodo de tiempo y se podían examinar los resultados obtenidos. Estas reuniones han estado muy ligadas a los sprints ya que se han utilizado para comprobar que los requisitos se iban cumpliendo como se esperaba, en los plazos señalados.

4.3 Control de versiones

Debido a que el proyecto ha sido realizado por tres alumnos, era necesaria una forma para que todos pudiéramos trabajar sincronizados y en paralelo. Así empezamos a usar un sistema de control de versiones, de forma que cada cambio realizado por alguno de los alumnos se pudiera guardar, y que el resto pudieran actualizar su versión con la actual. Este método de trabajo nos ha ayudado mucho a repartir las tareas ya que cada uno podía trabajar en una tarea muy distinta del resto sin necesidad de un sistema de integración, ya que cada cambio se podía guardar y actualizar directamente en el repositorio de trabajo, y así siempre tener actualizada la versión más reciente.

4.4 Entorno de desarrollo

ViRPlay3D está implementado en C++, ya que se necesitaba un lenguaje que tuviera unas buenas prestaciones en tiempo de ejecución. Para la implementación se ha usado el entorno de desarrollo *Microsoft Visual Studio* en su versión *Visual Studio .NET 2003* que permite a los desarrolladores crear

aplicaciones, sitios y aplicaciones web, así como servicios web en cualquier entorno que soporte la plataforma .NET.

5 *Desarrollo del proyecto*

5.1 Implementación y diseño de las nuevas funcionalidades del juego monousuario

5.1.1 Carga y almacenamiento de la información

Una de las primeras funcionalidades se añadieron fue la carga en el sistema de la información de los actores implicados en el role-play. Esta información no es la misma para los objetos que para las clases. Además el jugador debe poder consultar esta información cuando se acerque al objeto o la clase y realice la acción de mirar.

Carga de la información

Esta información se encuentra almacenada en dos archivos XML diferentes. La información sobre los objetos se genera al mismo tiempo que el guión del role-play que se debe jugar.

A continuación se describe el contenido de estos dos archivos

Tarjetas CRC o Clases:

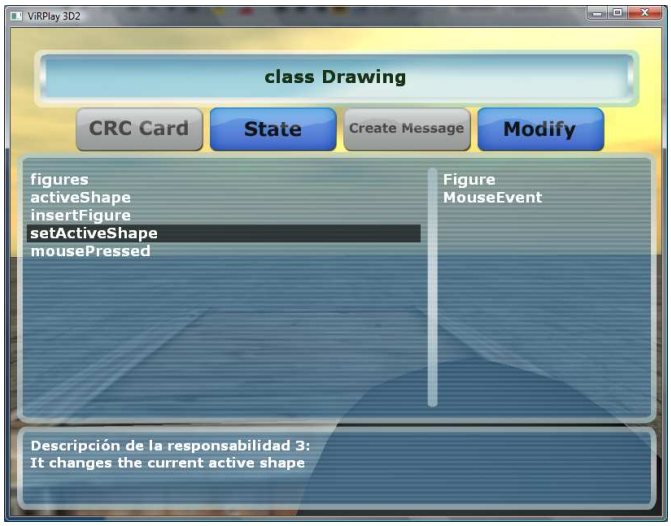

Figura 12. Información de las tarjetas CRC

- **Nombre**: de la clase
- **Descripción**: de la finalidad de la clase
- **Lista de responsabilidades** cada una con:
  - **Nombre**: de la responsabilidad
  - **Descripción**: de la responsabilidad
  - **Tipo**: del valor devuelto.
  - **Lista de parámetros** cada uno con:
    - **Nombre**: del parámetro
    - **Tipo**: del parámetro.
- **Lista de colaboradores** cada uno con:
  - **Nombre**: de la clase colaboradora.

Objetos:

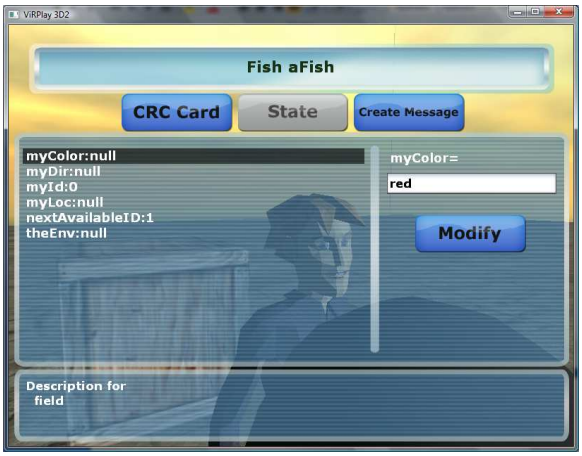

Figura 13. Información de un objeto

- **Nombre**: del objeto
- **Hashcode**: para identificar al objeto en el escenario
- **Tipo:** del objeto
- **Lista de atributos** cada uno con:
  - **Nombre**: del atributo
  - **Tipo**: del atributo
  - **Valor**: del atributo

Estos archivos son procesados y almacenados al principio de la partida durante la creación del escenario al comenzar la partida.

Almacenamiento de la información

Para almacenar esta información en el sistema, se crearon clases nuevas de almacenamiento acorde al contenido de cada una. Inicialmente se disponía de la clase CInfo, que almacenaba el nombre del objeto y de qué tipo de actor se trataba. Lo que se ha hecho es crear nuevas clases CObjectInfo y CClassInfo que heredan de ésta, pudiéndose almacenar independientemente del tipo que sean. De esto se encarga la clase CInfoRepositorio, formada por una lista de objetos CInfo. Será CSimulador quien tenga la referencia al repositorio como uno de sus atributos. Todo esto se refleja en diagrama de clases ce la figura 14.

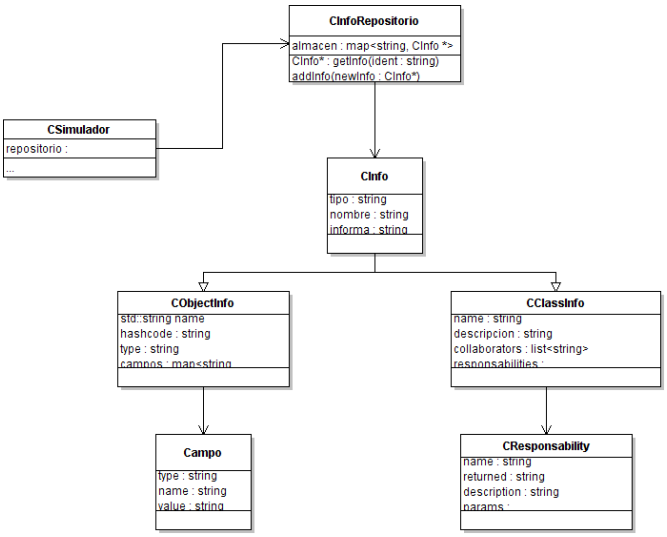

Figura 14. UML de las clases de almacenamiento de información

5.1.2 Visualización de la información

Uno de los aspectos gráficos a implementar, es la visualización en 2D de la diferente información que se tiene almacenada sobre los objetos y las tarjetas CRC. Además se deberá poder visualizar la información sobre el último mensaje ejecutado, o lo que es lo mismo la información del mensaje que hay en la pelota.

Creación de las interfaces

Para mostrar interfaces en nuestro programa la manera de crearlas será a través del servidor de scripts de TCL. Habrá un script para cada tipo de información que se quiera visualizar, y deberán ser cargados y descargados en el momento adecuado. Nuestra labor para implementar esta función, consiste por tanto en cargar el script adecuado para cada una de estas informaciones y llamar a una serie de funciones del script que rellenaran ciertos campos a mostrar, estos campos también variarán según el actor que se esté mirando.

Siguiendo la arquitectura de estados del juego, hay creado un estado que se encarga de mostrar la información del objeto que se está mirando. Este estado tiene como atributos un objeto de la clase *CInfo* que contiene la información que se está

mostrando, éste se establece cada vez que se solicita ver la información. Posee además un controlador de la interfaz (*CControladorMenuInventario*) que es el encargado de comunicarse con TCL para cargar el script y manejar los datos de la información activa. De esta manera se separa la gestión de los gráficos en 2D de la aplicación en sí, reduciéndolo a una serie de llamadas muy sencillas al servidor de scripts.

Inicialmente el CControladorMenuInventario cargaba siempre el mismo script y datos con lo que la información que se veía por pantalla era la misma. Lo primero que debemos hacer por tanto es definir esta clase como virtual y crear una interfaz de esta clase para cada tipo de interfaz 2D que se quiera crear. Las interfaces de esta clase deben poder por lo menos:

- **Dibujar la interfaz**: Se harán las llamadas correspondientes a través del servidor de scripts de la aplicación para comunicarle el valor de los campos a mostrar.
- **Destruir el interfaz**: Se solicitará la destrucción de la interfaz que se ha creado para visualizar la información.
- **Procesar orden**: por medio de este método se informará al controlador de la pulsación de un elemento de la interfaz. Normalmente tendrá que volver a enviar cierta información al servidor de scripts para actualizar la visualización.

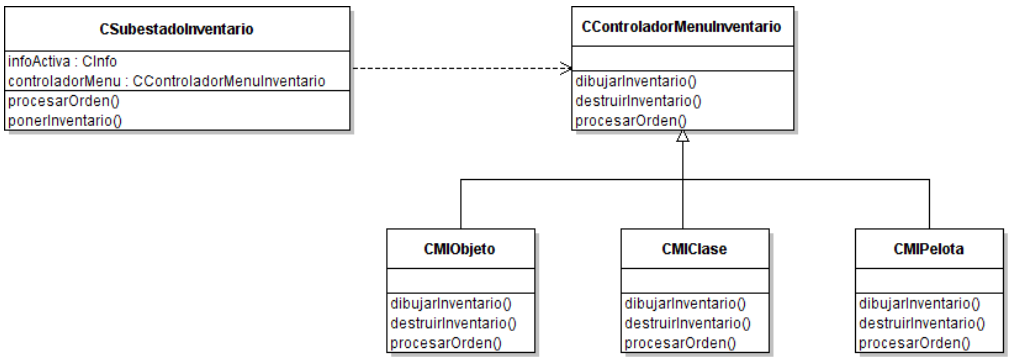

Figura 15. UML de las clases encargadas de manejar la interfaz

En la figura anterior se observa el diagrama de las clases que intervienen en el proceso de visualización de las informaciones. Como se ve tenemos una clase por cada uno de los tipos de información que se puede observar por pantalla. Este diseño es bastante importante, puesto que si quisiéramos añadir

otros tipos de interfaces 2D al juego solo habría que añadir otra clase que implementara las funciones necesarias.

Para visualizar la información de los objetos, el simulador dispone de un objeto de la clase CInventario, que es la que almacena la información activa. Siguiendo el sistema de observers de la aplicación, esta clase avisara a sus propios observers de las modificaciones que se produzcan sobre la información que se visualiza, como puede ser el propio establecimiento de la información o los cambios que se pueden producir sobre un objeto.

La clase que está a la escucha de la clase CInventario siendo observer de ésta, es el CSubestadoInventario, estado activo del juego cuando se muestran los inventarios, y que como ya se ha comentado antes, dispone de un CControladorMenuInvenario para manejar la interfaz. En el siguiente diagrama de clases se ilustra cómo están relacionadas las clases de almacenamiento de la información con cada uno de los controladores que las representan por pantalla.

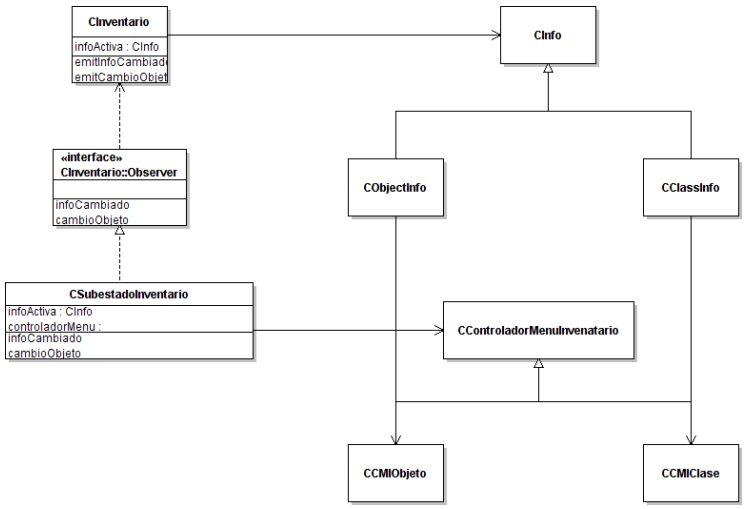

Figura 16. UML de las clases encargadas de la visualización

A continuación se muestra el diagrama de secuencia desde que el simulador detecta la orden de mostrar la información hasta que se lanza el script que muestra la información por pantalla.

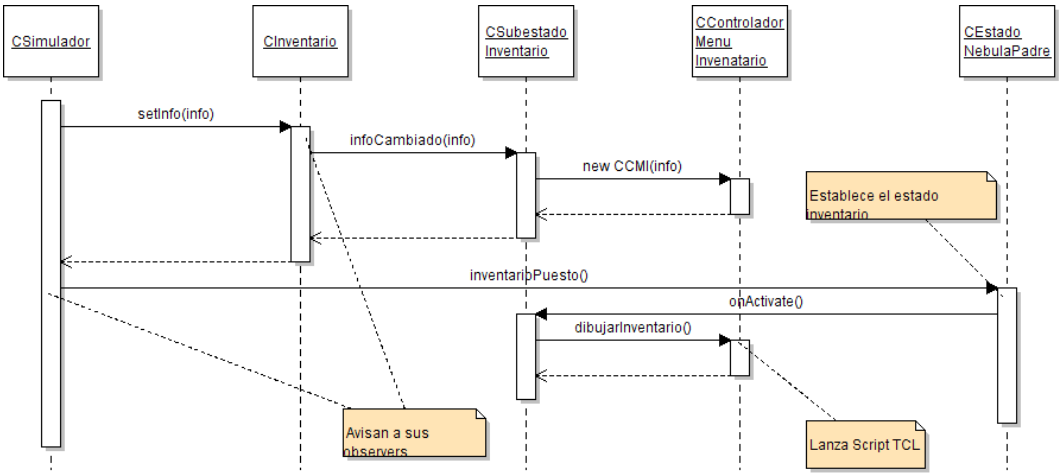

Figura 17. Diagrama de Secuencia de la visualización de la información

Como se ve el simulador lo que hace es avisar al inventario que se debe cambiar la información a mostrar. A continuación éste avisa a sus observes, entre ellos al subestado del inventario que establece la información a mostrar CSubestadoinventario. Éste crea el CControladorMenuInventario adecuado en función del tipo de información a mostrar, tal como se muestra en la figura 18. En este momento se devuelve el control al simulador que avisa a sus observers de que se tiene que pasar al estado del inventario. El observer del simulador es el estado principal del juego CEstadoNebulaPadre, que establece el subestado del inventario. En este momento CSubestadoInventario manda a su controlador dibujar el menú.

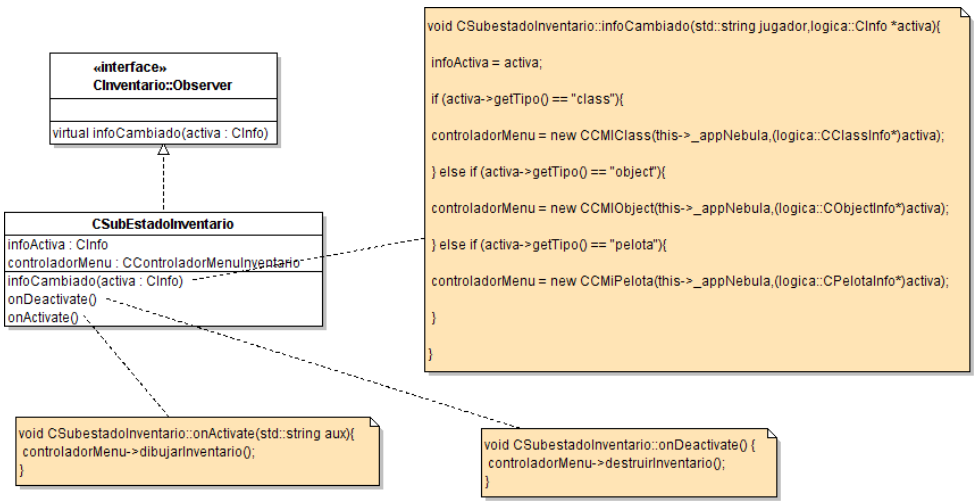

Figura 18. Manejo de la interfaz

Modificación de la información

Cuando el simulador recibe un comando para cambiar los campos de un objeto, tras la modificación de la información en el repositorio, se debe pedir al inventario que avise de este cambio a sus observers, si hay alguna información mostrándose y corresponde con la del objeto cambiado, deben actualizarse los datos en pantalla. La secuencia de ejecución ocurre de forma parecida a la visualización de la información.

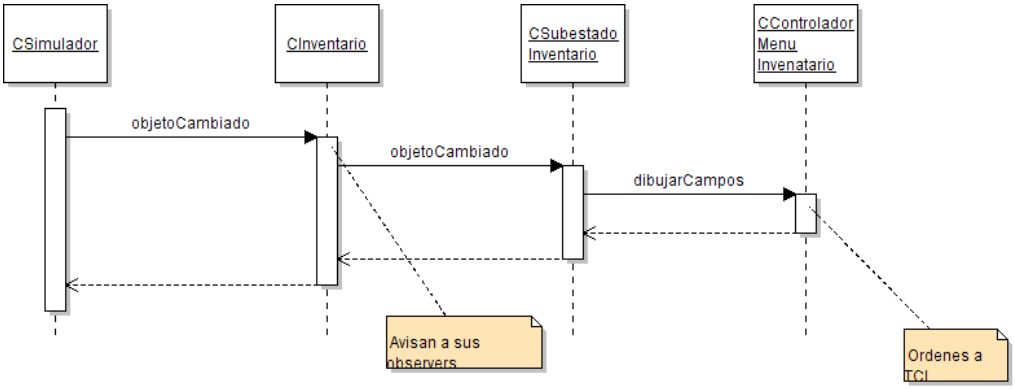

Figura 19. Secuencia de la modificación de la información

Interacción del jugador con la interfaz

Cuando el jugador está consultando la información de uno de los actores del role-play, podrá normalmente interactuar con ella. Por ejemplo para consultar la descripción de la responsabilidad que ha seleccionado de una clase, o cambiar el valor de un campo de un objeto.

La forma de tratar estas acciones, es a través de una cola de órdenes que se consulta en cada ciclo y en la que se van depositando las ordenes provenientes del GUI y. La aplicación hará llegar estas órdenes hasta el estado actual, en este caso CSubestadoInventario. Éste a su vez le pasará al CControladorMenuInventario las órdenes que considere oportunas. Estas órdenes contendrán la información necesaria para tratar la acción del usuario. Por ejemplo, cuando se cambie el valor de un atributo en un objeto, este cambio se comunicará al CControladorMenuInventario a través de una de estas órdenes, y este a su vez se lo comunicara al simulador para actualizar la información del objeto con el cambio que se ha

producido. Si por ejemplo la orden es la de cerrar el inventario, el que la tratará será el CSubestadoInventario para destruir el controlador del menú.

En este punto cabe mencionar la importancia del sistema de observadores de la aplicación. Cuando un controlador detecta que se ha producido un cambio no actualiza la interfaz inmediatamente, sino que espera a que se le avise desde el simulador.

5.1.3 Gestión del paso de mensajes

Como ya se ha explicado, el paso de los mensajes es quizá la parte fundamental en la ejecución de cualquier escenario de role-play. Es por tanto necesario incluir en nuestro programa una serie de elementos que se encarguen de esto. Las tareas por tanto que deben realizarse, son, la carga y almacenamiento del guión del role-play, separando cada paso de ejecución, para poder ser ejecutados independientemente y en el orden adecuado.

Carga de la información

Al igual que con la información de los objetos y las tarjetas CRC, se usa un archivo en XML codificado con el guión que debe ejecutarse en el escenario. De hecho se encuentra en el mismo archivo que la información de los objetos, ya que los pasos de ejecución están en su mayor medida condicionados por los objetos que estén presentes.

La información que contiene el archivo aparte de la definición de los objetos, es una lista con los diferentes pasos, donde cada uno está representado por las siguientes características:

- **Nombre**: del método ejecutado en el mensaje
- **Tipo de mensaje**: indicando si es de llamada o de retorno.
- **Invocador**: código hash del objeto que hace la llamada

- **Invocado**: código hash del objeto que se recibe el mensaje.
- **Tipo y valor de retorno**: si es un mensaje de retorno.
- **Lista de parámetros** con:
  - **Valor**: del parámetro
- **Lista de cambios:** que se producen en los campos de los objetos cada uno con:
  - **Tipo**: del campo a modificar
  - **Nombre**: del campo
  - **Valor**: nuevo valor del campo

Gestión de los mensajes

La parte más importante a desarrollar es una clase que se encargue del almacenamiento y manejo de los mensajes. Esta clase es CGestorRolePlay, y es la encargada de proporcionar al simulador los métodos necesarios para la ejecución del guión paso a paso.

Además el CGestorRolePlay debe ser el encargo de completar estos pasos de ejecución con la información necesaria para que la parte grafica de la ejecución se pueda realizar, como pueden ser las entidades 3D que representan a los objetos que participan en el paso de mensaje. El almacenamiento y la gestión de los mensajes se representan en la siguiente figura:

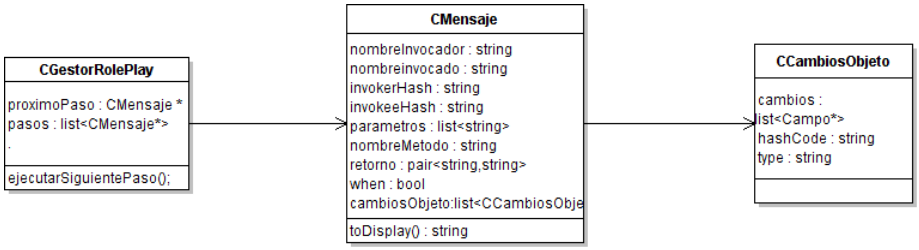

Figura 20. Clases de almacenamiento del guión de role-play

Creación de mensajes

Para completar el manejo de los mensajes, y aunque no sea una de las funcionalidades importantes, se le ofrece al jugador la posibilidad de crear mensajes durante el transcurso de la partida. Estos mensajes, además de ejecutarse, se salvaran en un archivo de texto en XML con el mismo formato que hemos explicado arriba para la carga de la información.

Para construir los mensajes, el jugador se acerca al objeto que tiene la pelota en ese momento, para establecerle como invocador del próximo mensaje. Una vez que se tiene establecido el objeto que lanzara el mensaje, se pasan a definir las características de éste. Se distinguen los siguientes campos:

- ***Invoker***: El invocador, es el objeto que tiene intención de comunicarse y el que lance la pelota.
- **Invokee**: El invocado, es el objeto que recibirá el mensaje, es decir el que recogerá la pelota.
- ***Message****:* El mensaje que quieres pasar, será el nombre del método a usar.
- ***Return****:* Marcar para indicar que el mensaje es de retorno. Los mensajes de retorno se representa gráficamente por un tiro raso, mientras que los otros se representan por un tiro parabólico.
- ***Return Value****:* El valor de retorno.
- ***Parameteres****:* Parámetros del método y su valor asociado.

Figura 21. Inventario de creación de mensaje

El jugador construirá los mensajes a través de una interfaz los aspectos que se han especificado arriba. Esto ocurre en el estado de la aplicación designado a ello, el CSubestadoCracionMensaje. Una vez que el jugador haya definido el mensaje, basta decir que se enviarán los datos al simulador y éste se los comunicara al CGestorRolePlay para que construya y ejecute el nuevo mensaje.

Tras la configuración del menú, el mensaje quedará construido y pasará a ejecutarse. El Invoker lanzará la pelota por alto o bajo, dependiendo de si el mensaje es de retorno o no, al invocado con los valores configurados.

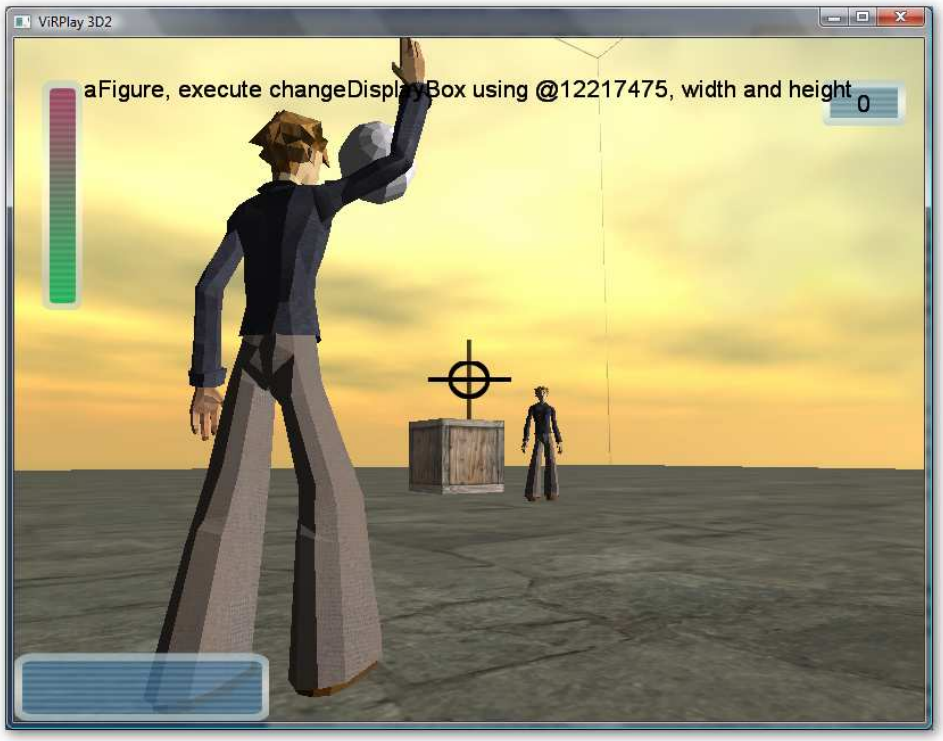

Figura 22. aDrawing manda un mensaje a aFigure para que ejecute el método "changeDisplayBox"

Para posteriormente salvar esta información, el CGestorRolePlay dispondrá de un almacén de mensajes que codificará en el archivo una vez que el jugador abandone la partida.

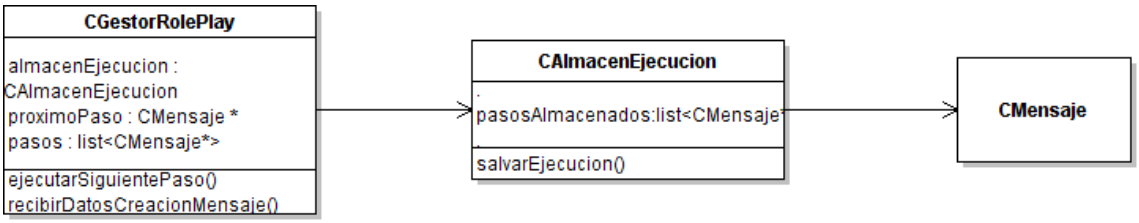

Figura 23. Clases de almacenamiento de pasos ejecutados

Visualización del contenido del mensaje

Para permitir al jugador seguir el desarrollo del guion del rol-play, se puede también consultar la información del último paso de mensaje ejecutado. Podrá conocer en todo momento quien fue el último objeto en pasar ese mensaje, o cual fue el valor de los argumentos del mismo. Se accede mirando a la

pelota de la misma manera que a los objetos o a las tarjetas CRC. Se puede decir que la pelota almacena la información del último mensaje.

Figura 24. Contenido del mensaje

El funcionamiento es por tanto el mismo que cuando se visualizaban las informaciones de los actores. Se tendrá una copia almacenada de la información que hay en la pelota en un objeto de la clase CInfoPelota que al igual que las clases de almacenamiento de los otros actores, heredará de la clase CInfo. Análogamente para representar esta información por pantalla, bastara con añadir otro controlador 2D que se comunique adecuadamente con el servidor de scripts de la misma manera que se representaban las otras informaciones.

La pelota actúa como mecanismo de comunicación entre los objetos. Para ello, la pelota tiene asociado un menú con el cual el jugador puede crear un mensaje. El aspecto del menú es el siguiente:

Animación de los pasos de mensaje

El paso de mensajes es la forma de comunicación principal que tienen los objetos de role-play. Teníamos que representar esta comunicación de forma gráfica. Se decidió, que cuando un objeto quisiera comunicarse con otro, se realizarían una serie de animaciones. El invocador lanzaría la pelota por alto, en el caso de que ejecutara un método que no fuera de retorno, o lanzaría la pelota por abajo en el caso contrario.

Todas estas animaciones había que coordinarlas en tiempo real, es decir, no se podía hacer que cuando un jugador quisiera pasar un mensaje, activar la animación y esperar a que terminase, ya que de esta forma se paraliza la lógica, y la idea es que mientras se está pasando un mensaje, el resto de jugadores puedan seguir haciendo sus actividades sin problemas.

Para coordinar las animaciones, se decidió que fuera la pelota la encargada de hacerlo. Se diseño con una máquina de estados interna, de forma que dependiendo de en qué estado estuviera, la GUI supiera como pintarla y que animación realizar en función de en qué estado se encontrara (Ver Figura 25). Así pues, el invocador simplemente tendría que rellenar la pelota con la información necesaria para pasar el mensaje, y ejecutar su método inicio(), y a partir de ahí la pelota se encargaría del resto. Al final, la coordinación quedó de la siguiente manera:

Parte lógica (estado de la pelota)	**Parte gráfica**
Estado Encarando	Animación en la que los objetos van cambiando su orientación hasta quedar enfrentados.
Estado Hablando	Los objetos realizan la animación de hablar y se imprime por pantalla el texto que intercambian.
Estado Lanzando	El objeto que posee la pelota comienza su animación de lanzamiento hasta que suelta la pelota. La animación puede ser un lanzamiento aéreo o por raso.
Estado Pelota Lanzada	La pelota va cambiando su posición en cada tick de juego y la parte gráfica la repinta como a una entidad más.
Estado Pelota Cogiendo	El objeto destino comienza la animación de recoger la bola, ya sea por arriba o por abajo, dependiendo desde donde venga la pelota.

Estado Pelota Reposo	La GUI libera los recursos necesarios utilizados para la animación de la pelota

Hemos visto como se ha coordinado desde el punto de vista lógico, pero ¿qué hay desde el punto de vista gráfico? Para que la parte gráfica pudiera actualizarse de manera acorde con la lógica, se creó la clase pelotaGUI, que sería la encargada de esta labor. Nos basamos en el sistema de observers de ViRPlay3D, creando una clase observer interna de la pelota que estaría a la escucha de cada cambio de estado, para poder pintar por pantalla lo que fuera necesario (Ver Figura 26).

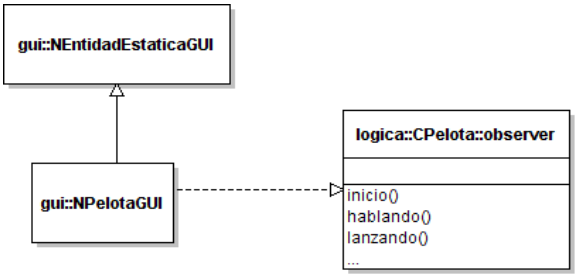

Figura 25. Coordinación de la parte gráfica y lógica de la pelota mediante observers

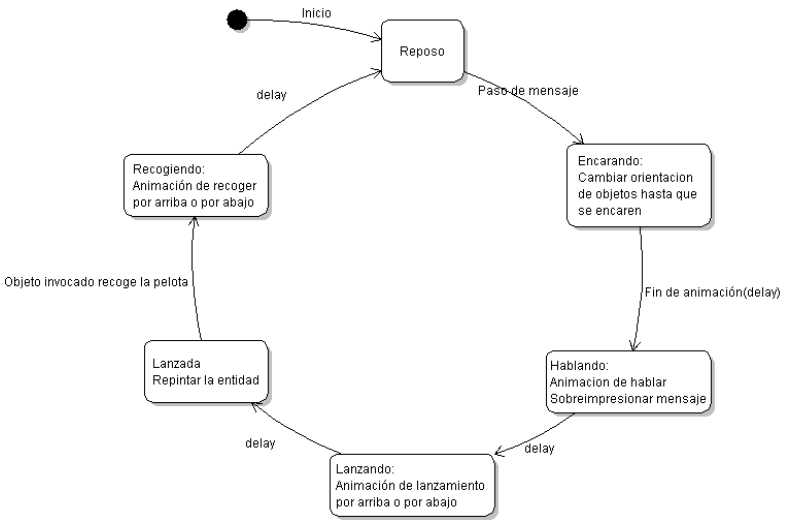

Figura 26. Diagrama de estados de la pelota en la parte gráfica

5.1.4 Escenarios

Todo el desarrollo del juego de ViRPlay3D se basa en escenarios. Un escenario no es más que un conjunto de información. Esta información tiene que ser completa. Cada escenario tiene que describir íntegramente cada una de las partes del juego:

- ***Mapa**:* Para cada escenario hay que describir que entidades (clases, objetos, avatar….) hay en el mundo y cuál es su ubicación inicial dentro de él.
- ***Información estática:*** Contiene la información relacionada con las clases de objetos que participan en la simulación y está accesible desde los inventarios de los objetos y las clases.
- ***Información de ejecución:*** Describe cómo se van a comportar los objetos que intervienen en la simulación a medida que el alumno solicita el siguiente paso de ejecución o decide interpretar el papel del objeto activo. Además, esta información se usa para ir actualizando el estado de los objetos tras cada paso de ejecución.

Está información tiene un formato determinado y se almacena en el sistema mediante archivos XML.

ViRPlay3D2 ofrece un sistema extensible de escenarios. A los escenarios inicialmente creados, se le pueden añadir cuantos se deseen siguiendo el formato correcto para ellos.

- ***Sistema de escenarios***

ViRPlay3D2 ofrece un sistema extensible de escenarios. De esta forma cada vez que se añade un escenario nuevo (siguiendo el formato correcto), la aplicación lo reconocerá y procesará de forma que pase a estar disponible para su uso.

Todos estos escenarios se almacenan en una carpeta "escenarios" de forma que cada vez que se quiera añadir uno nuevo, se introduce mediante una capeta con el nombre del escenario y que contiene todos los archivos necesarios. Estos archivos indispensables son los que contienen toda la información necesaria para la partida y tienen una notación y

formato específicos. Los archivos que tiene que contener cada nuevo escenario son los siguientes:

- ***ClassInfo***: Archivo contiene toda la información sobre las clases.
- ***Config.***xml: Este archivo tiene un nombre preestablecido que no se puede modificar. Es un archivo de configuración donde se describen los nombres y las rutas del resto de archivos necesarios para la creación y simulación de un escenario.
- ***escenario***: Archivo que describe toda la información sobre el mapa. Contiene la distribución de las entidades al inicio de la partida.
- ***LongDescription***: Archivo con la descripción larga del escenario.
- ***RolePlay***: Archivo con toda la información de ejecución del escenario.

Cada archivo descrito tiene una notación y formatos propios que se deben seguir para el correcto funcionamiento. Todos los archivos a excepción de Config.xml pueden tener el nombre que se quiera, de forma que aparezca la ruta del archivo con ese nombre en el archivo de configuración.

La aplicación tiene un módulo encargado del análisis y mostrado de los escenarios. Esto se realiza en el estado selescenario de forma que cada vez que se entra en este estado se busca en la carpeta externa "escenario" todos los escenarios almacenados. De estos se extrae el fichero Config.xml que es el que contiene todas las rutas de los ficheros de información. Cuando se tienen todos los escenarios disponibles se muestran por pantalla, permitiendo al usuario seleccionar el que más desee.

Con esto conseguimos saber cuántos escenarios tenemos disponibles pero no parseamos la información completa sobre cada uno de ellos. El parseado global de toda la información se hace al iniciar la lógica mediante un sistema de analizadores léxicos, cada uno de los con una determinada función.

- ***Secuencia de ejecución***

Cuando el usuario juega en modo monojugador o hace de servidor, la primera pantalla que aparece es la selección de escenarios. En esta pantalla se muestran todos los escenarios disponibles para el juego, de forma que cada vez que se seleccione uno, aparecerá en la parte inferior de la pantalla una breve descripción sobre este. Cuando se pulsa la tecla *ok* se valida la selección del escenario y se pasa al siguiente estado de juego.

Las rutas de los archivos de información quedan almacenadas de forma que cuando se necesite rescatar la información se buscan mediante estas rutas. Hay que tener en cuenta que el orden de parseo importa. Primero hay que tener todo el sistema de observers preparado, para que en el momento de actualizar la lógica con la información de estos archivos, se muestren los cambios en la interfaz gráfica o se notifique por red, si estamos en multijugador, a todos los clientes.

Ejemplo de interfaz gráfico del menú de selección de escenarios:

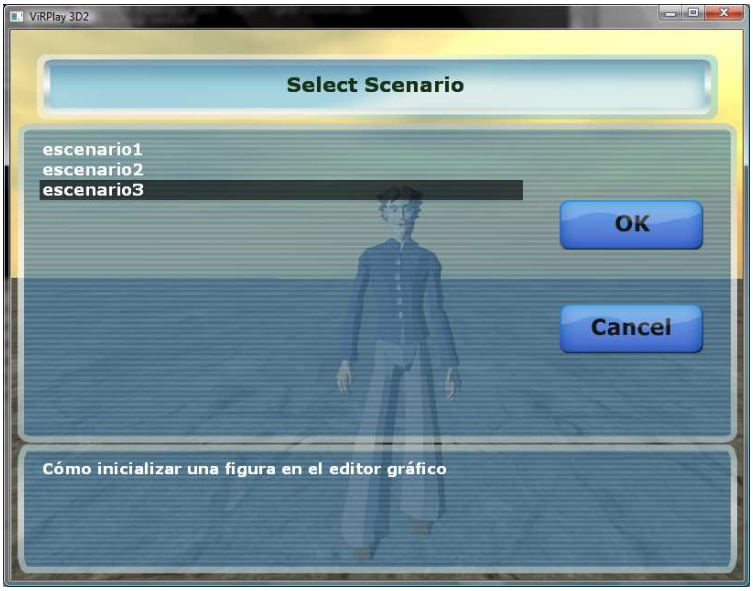

Figura 27. Interfaz gráfico del menú selección de escenarios

5.2 Red

5.2.1 Definición de la arquitectura

Como se ha comentado en la sección de objetivos, se ha optado por una arquitectura Cliente-Servidor con lógica centralizada. De esta forma la aplicación se divide en dos partes bien diferenciadas como se puede ver en la figura 28:

- El servidor.
- Los clientes.

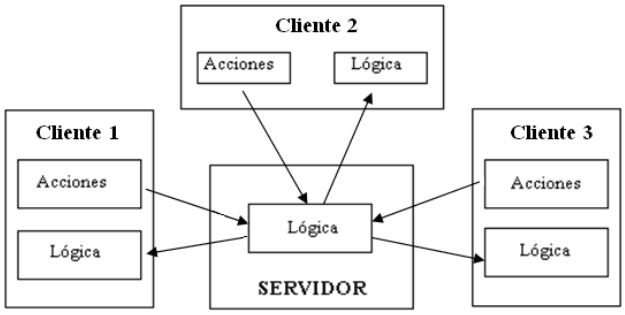

Figura 28. Esquema de conexión entre servidor y clientes

El servidor es la máquina encargada del inicio del juego. Una vez seleccionado, se escoge el escenario en el que se va a jugar, comunicando a los clientes que se conecten a él la información necesaria para iniciar el juego.

Los clientes se conectan a la máquina servidora, por lo tanto está tiene que estar en funcionamiento a la hora de ejecutar el cliente. Una vez conectado al servidor, el cliente elige su rol en el escenario, y una vez que todos los objetos han sido seleccionados se comienza el juego.

Todas las acciones realizadas por los clientes no se reflejan directamente en la partida, si no que primero se comunican a la máquina servidora. Ésta primero actualiza la lógica central comunicando cada cambio a todos los clientes que actualizan su lógica marioneta, actualizando la interfaz gráfica según corresponda.

Para implementar esta nueva funcionalidad se ha hecho uso de una librería extra diseñada para la comunicación en red. Esta librería es Enet.

5.2.2 Librería Enet

ENet es una librería genérica de red implementada en C y portable a Linux y Windows, cuya finalidad es proporcionar una relativamente pequeña, sencilla y potente red de comunicación en la capa superior de UDP (User Datagram Protocol). Hace uso de dos protocolos de transferencia de datos, TCP y UDP, de forma que crea un control de paquetes básico (una visión reducida de TCP) pero utilizando UDP.

Las características básicas de eNet son:

- Gestión de la conexión: Define una interfaz muy simple para la conexión entre hosts.
- Secuenciamiento de paquetes: Garantía en el orden de llegada.
- Canales: A través de la misma conexión se puedes crear varios canales de datos.
- Muy útil cuando se necesita separar el control de la transferencia de información.
- Fiabilidad: eNet puede garantizar la entrega de paquetes.
- Fragmentación y Reensamblado de los paquetes.
- Configurable: Es posible configurar varios parámetros como el tamaño de la ventana, time-out, etc.

Las clases principales de eNet son:

- ***Servidor***: representa la clase que inicia la red para que se conecten a ella.
- ***Cliente***: representa la clase para conectarse a un servidor.
- ***Conexión***: representa una conexión entre un cliente y un servidor.
- ***Paquete***: son los datos que se envían en una conexión.

La forma de comunicarse es mediante el método **Service**() que es el que se encarga de procesar los paquetes recibidos.

5.2.3 Sistema de servidor y clientes

Debido a que el sistema de servidor y clientes se ha implementado siguiendo un diseño de lógica centralizada, se consigue que solo sea el servidor el que cambie la lógica, impidiendo así que se produzcan incoherencias y consiguiendo una sincronización muy alta. Las características principales de esta técnica son:

- Los clientes solo envían los movimientos o acciones que realizan los jugadores.
- El servidor recibe estos movimientos y actualiza su lógica, reenviando estos nuevos cambios a los clientes para que actualicen su lógica "marioneta" según corresponda.

Para ejecutar el juego en modo multijugador van a hacer falta dos tipos de aplicaciones: una que funcione en modo servidor, y el resto en modo cliente.

Tanto el servidor como el cliente disponen de una herramienta, diseñada mediante sendos menús que se pueden ver en la figura 29, para cambiar las opciones básicas para la conexión de red. Estas opciones vienen establecidas por defecto a unos valores modificables. Son, para el servidor:

- ***Max. Connections:*** Para establecer el número máximo de clientes que permitimos que se conecten al servidor.
- ***Port:*** Puerto para establecer la conexión.

Las opciones configurables desde el punto de vista del cliente son:

- ***Ip Address:*** Dirección ip donde se encuentre el servidor.
- ***Port:*** Puerto del servidor.

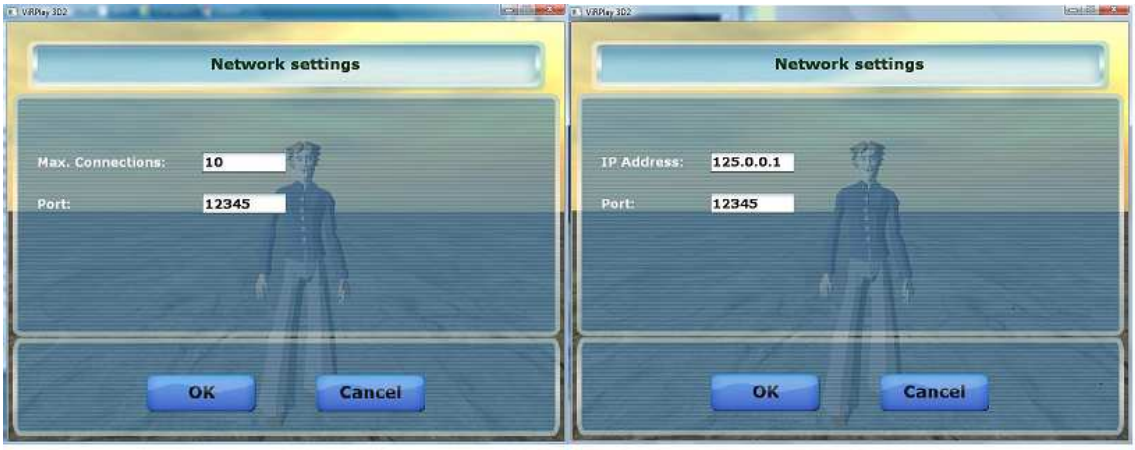

Figura 29. Menús de configuración de las opciones de red. A la izquierda las opciones del servidor, a la derecha las del cliente.

5.2.4 Servidor

- *Arquitectura del servidor*

El servidor es la aplicación que se encarga de iniciar la red para el modo multijugador. Ha sido desarrollado de forma estructurada, para que cada módulo se encargue de una tarea concreta. Desde el menú principal pulsando el botón del servidor (3) se pasa a la inicialización del servidor que está formado por un conjunto de clases que forman una máquina de estados:

- **selescenarioRed:** este estado es el encargado de seleccionar el escenario en el que se va a desarrollar el juego. Para esto se recorren los ficheros de escenarios para determinar cuales hay disponibles. Una vez obtenido el conjunto de escenarios se muestra el menú gráfico, donde aparte del nombre del escenario aparece una pequeña descripción de cada uno de ellos. Una vez seleccionado un escenario se pasa al siguiente estado, selpersonaje.
- **selpersonaje:** en este estado se inicializa el servidor mediante la librería eNet, con el puerto y el número de clientes máximo deseado. Este estado tiene dos partes: la primera es establecer la conexión con cada cliente que se le conecte; la segunda parte consiste en la gestión de la información que se transmiten entre el servidor y el cliente. Inicialmente el servidor manda a los clientes la información del escenario seleccionado, esto es, la lista con los posibles roles que se pueden escoger y la situación de cada uno de ellos

(seleccionado/no seleccionado). Los clientes al seleccionar o deseleccionar un personaje envían paquetes al servidor con la información sobre la selección del rol. Finalmente cuando todos los roles se han seleccionado, la partida comienza, pasando el servidor a su estado juegoservidor.

- **JugandoServidor:** en este estado se inicializa la lógica, se crea la proxyLogica del servidor para gestionar la comunicación con los clientes y se envía el paquete de empezar juego a todos los clientes. Una vez llegado a este estado el servidor se queda a la espera de eventos, actualizando la lógica y la proxyLogica cada cierto tiempo. Cuando se termina la ejecución se sale del juego. Desde este estado se puede salir de la partida y volver al menú principal.

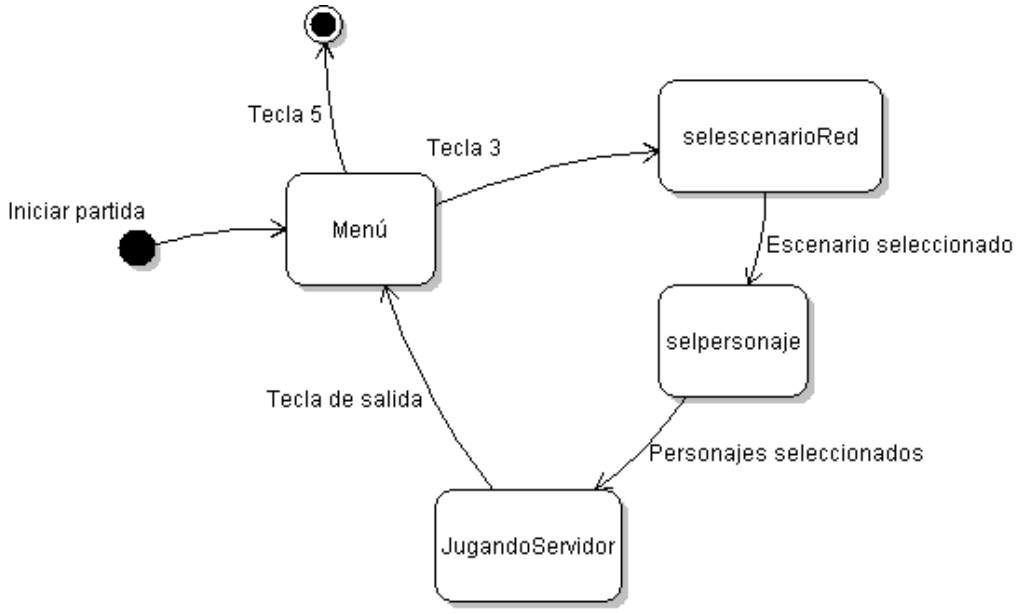

Figura 30. Diagrama de estados del servidor

- *Secuencia de ejecución*

La secuencia de ejecución del modo multijugador desde la parte del servidor seguiría el siguiente desarrollo. Al iniciar el juego en red, se selecciona el escenario en el que se va a jugar, y se crea el servidor. Una vez inicializado el servidor, y ya conociendo el número de jugadores que pueden entrar en juego, el servidor inicia un modo de espera de clientes, de modo que para cada conexión que recibe, le envía toda la información del escenario necesaria para la partida. Una vez que todos los clientes han seleccionado su rol, el servidor pasa a un estado de espera activa. Ya que el diseño de la red se basa en

una lógica centralizada, cada cambio producido en algún cliente, es comunicado al servidor mediante un mecanismo de *proxys*. Cada cambio producido en la lógica del servidor se comunica a los clientes que estén involucrados en este cambio.

5.2.5 Cliente

- *Arquitectura del cliente*

Los clientes son las aplicaciones que se conectan al servidor. Estos son los que verdaderamente realizan el juego ya que son los que se mueven y realizan acciones en la partida. Desde el menú principal pulsando el botón de cliente (2) se pasa a la inicialización del cliente que está formado por un conjunto de clases que forman una máquina de estados:

- **ConectaServer:** En esta clase se crea el cliente y la conexión con el servidor. Cuando se recibe un paquete con la confirmación de la conexión se pasa al estado selpersonajecliente.
- **SelPersonajeCliente:** En este estado es donde se realiza la selección del personaje. Inicialmente se carga la interfaz gráfica con la correspondiente información del escenario necesaria. En este momento se puede seleccionar o deseleccionar un personaje, mostrándose en la parte inferior de la pantalla las acciones realizadas tanto por sí mismo, como por otros clientes. Cada vez que un cliente es seleccionado se muestra en su nombre el símbolo "*". Una vez que todos los clientes han aceptado su rol el servidor manda un aviso para que empiece el juego y se pasa al estado juegoCliente.
- **JuegoCliente:** Este estado es a su vez una máquina de estados donde se realiza todo el juego por parte del cliente. Se prepara la comunicación con el servidor y se inicializa la máquina de estados al igual que se hace en el modo monojugador.

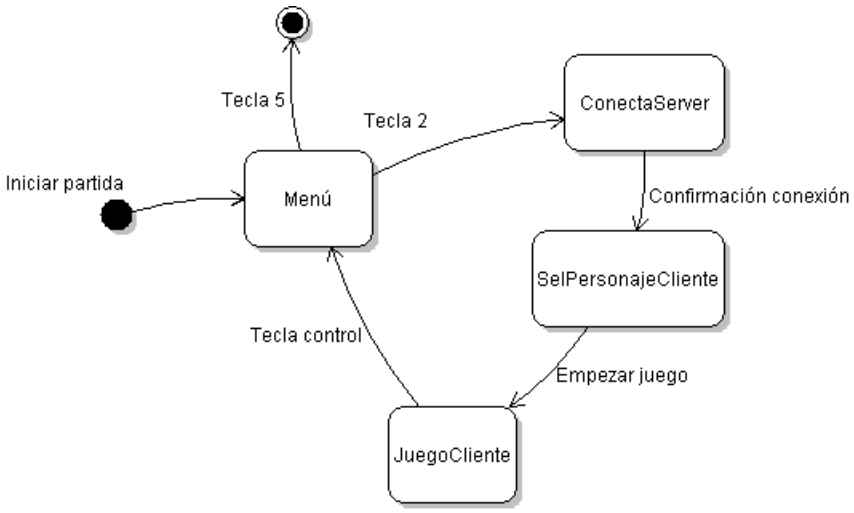

Figura 31. Diagrama de estados del cliente

Una vez realizadas las inicializaciones de los estados y añadidos los observers necesarios se pasa al estado esperandoempiece:

- **EsperandoEmpiece:** En este estado se muestra una interfaz gráfica con la descripción larga del escenario seleccionado para el juego. Al pulsar la tecla *espacio* se pasa al estado JugandoCliente.
- **JugandoCliente:** Este es el estado normal de juego. En este estado es donde nos podemos mover por el mundo y realizar acciones. Se diferencia del modo monojugador en que cada vez que realizamos una acción que repercute en la lógica, no la cambiamos directamente, sino que se informa al servidor que es el que actualiza la lógica global. De aquí se puede pasar a varios estados dependiendo de la acción realizada:
    - Tecla *tabulación*: se pasa al estado chat (También se pasa a este estado si otro cliente manda un mensaje de chat).
    - Teclas *numéricas* (0 al 5): Si hay alguna acción posible a realizar en alguna de estas teclas se pasa al estado inventario (Este paso se realiza de forma indirecta mediante los observers y las proxys).
    - Tecla *control*: Se vuelve al menú inicial.
- **ChatCliente:** En este estado se muestra la interfaz gráfica del chat para mandar mensajes a otros clientes. Inicialmente se carga la información del chat para que se muestre el historial

de la conversación. Al pulsar la tecla *tabulación* se vuelve al estado jugando de red.

- **InventarioCliente:** En este estado se muestra la información de la entidad seleccionada desde el estado jugando. Según el tipo de información que se vaya a mostrar se llama a un controlador diferente. Dependiendo de la información que visualizamos podremos realizar unas acciones u otras. Al visualizar el inventario de un objeto podemos desear crear un mensaje pasando al estado creacionmensaje. Si pulsamos la tecla *espacio* volvemos al estado jugando de red.
- **CreacionMensajeCliente:** En este estado se pueden crear mensajes con una información deseada. Una vez creada esta información o pulsando la tecla *espacio* volvemos al estado jugando de red.

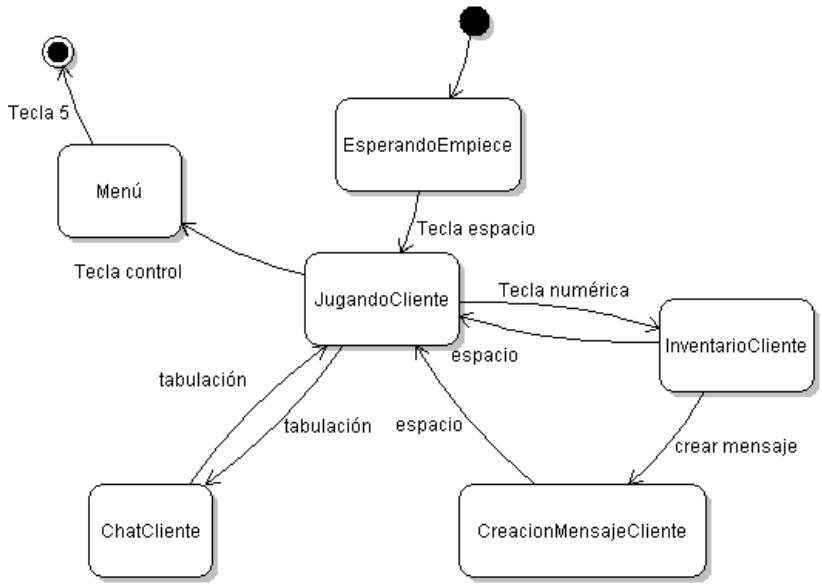

Figura 32. Diagrama de estados del estado juegoCliente

- *Secuencia de ejecución*

La secuencia de ejecución del modo multijugador desde la parte del cliente seguiría el siguiente desarrollo. Al iniciar el juego en modo cliente se envía una petición al servidor. El servidor primero contesta aceptando la conexión. Cuando la conexión está establecida se pasa a la selección del personaje. El cliente recibirá un paquete con la información necesaria sobre el escenario donde se va a desarrollar la partida. Con esta información el cliente puede seleccionar el rol que va a

desempeñar en el juego, comunicando cada cambio que realice al servidor (selección/deselección de roles).

Una vez que todos los clientes han sido seleccionados se muestra la pantalla de inicio del escenario con la información detallada. Al pulsar la tecla espacio se pasa a la parte más vistosa del juego, donde el cliente se puede mover libremente por el mundo o realizar acciones tales como consultar información de entidades o enviar mensajes. Cada vez que el jugador realice alguna de estas acciones, se le enviará por red al servidor, mediante el mecanismo de *proxys*, la acción realizada para que el servidor realice los cambios oportunos. Con la lógica global ya actualizada, se envían estos nuevos cambios a los clientes, que actualizan su lógica "marioneta" para que se muestren mediante la interfaz gráfica y se visualicen por pantalla.

5.2.6 Comunicación cliente-servidor mediante el mecanismo de Proxys

Para llevar a cabo la comunicación cliente-servidor, se ha optado por un mecanismo de proxys. Es decir, habrá dos proxys principales, uno en la parte del servidor, llamado proxylógica y otro en la parte del cliente llamado proxyappgui. Para mantener la sincronización, se aprovecha el mecanismo de observers que ya estaba montado y se añaden a los proxys como oyentes. De esta forma, cada vez que se produzca un cambio en la lógica, se comunicará a todos los oyentes, una vez llega al proxylógica, mandará el paquete correspondiente por red, el cual será recibido por el proxyappgui que se encargará de mandarlo al cliente de forma legible.

En la parte de red se distinguen 2 paquetes:

- red: Contiene las clases de más bajo nivel para trabajar con la red, entre las que destacan la clase cliente y la clase servidor. La clase conexión contiene las propiedades básicas de una comunicación: puerto, IP, etc... Se hace uso de una clase factoría para facilitar la construcción del cliente y el servidor.
- redAplicacion: Contiene las clases de más alto nivel, todos los proxys están contenidos aquí.

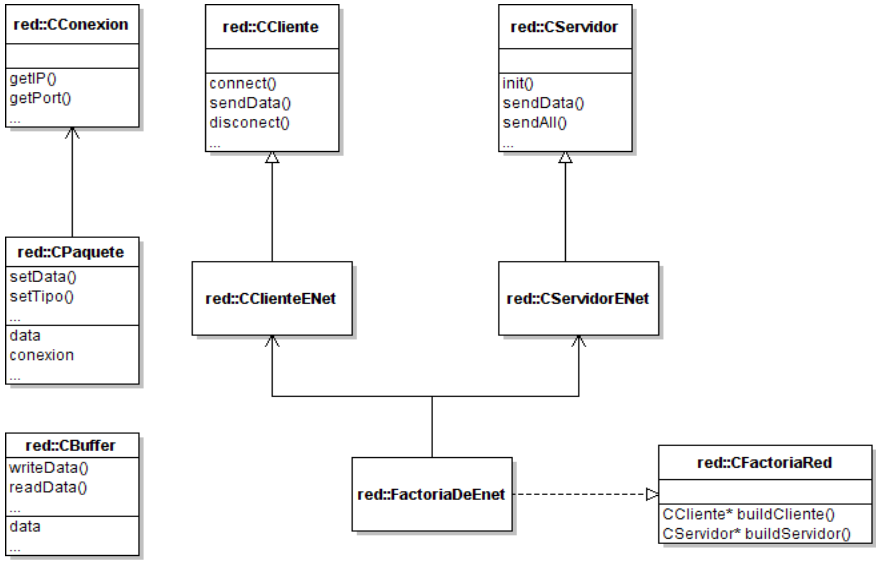

Figura 33. UML del paquete red

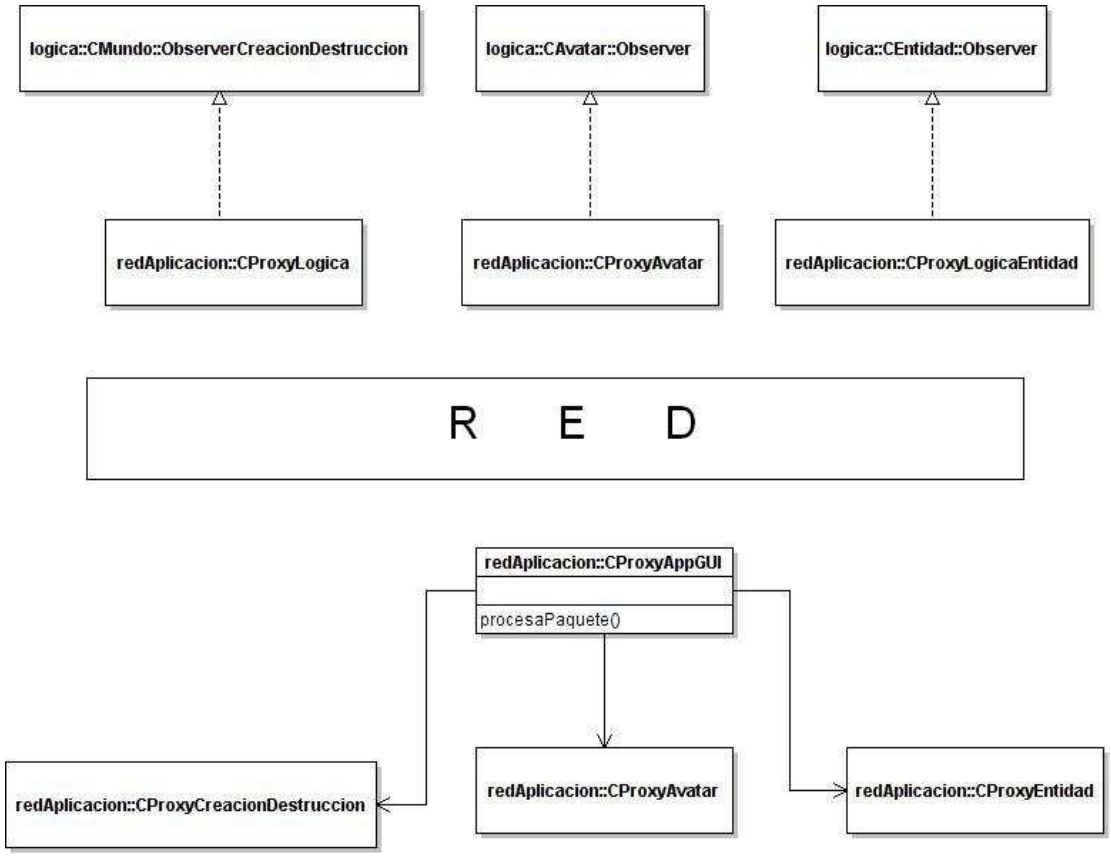

Figura 34. UML de parte del mecanismo de proxys

En este último esquema se observa la arquitectura de los proxys. En la parte de la lógica los proxys son oyentes de las clases que emiten eventos. Cada proxy de la lógica envía por red los paquetes de datos con las actualizaciones necesarias para los clientes. En el lado del cliente, la clase CproxyAppGUI recibe los paquetes, y los distribuye a las clases concretas para que cada una procese el paquete de la manera adecuada.

También hay que tener en cuenta que en ocasiones, el cliente tiene que comunicarse con el servidor, por ejemplo para mandar comandos como de movimiento o visualizar inventarios. La clase CproxyAppGUI tiene una doble función, aparte de recibir los paquetes de los proxys de la lógica, también recibe comandos que insertará en el proxyLogica para que esta los inserte en el simulador y puedan ser procesados correctamente.

A continuación se muestra una imagen con el esquema simplificado de esta arquitectura de proxys. En la práctica, no hay un solo proxylógica, sino que hay un Proxy diferente por cada clase que tiene observers e interesa notificar de cambios producidos en su lógica. Así mismo, tampoco hay un Proxy único en el cliente, sino que habrá uno por cada Proxy lógica que haya y cada uno se comunicará con el que le corresponda.

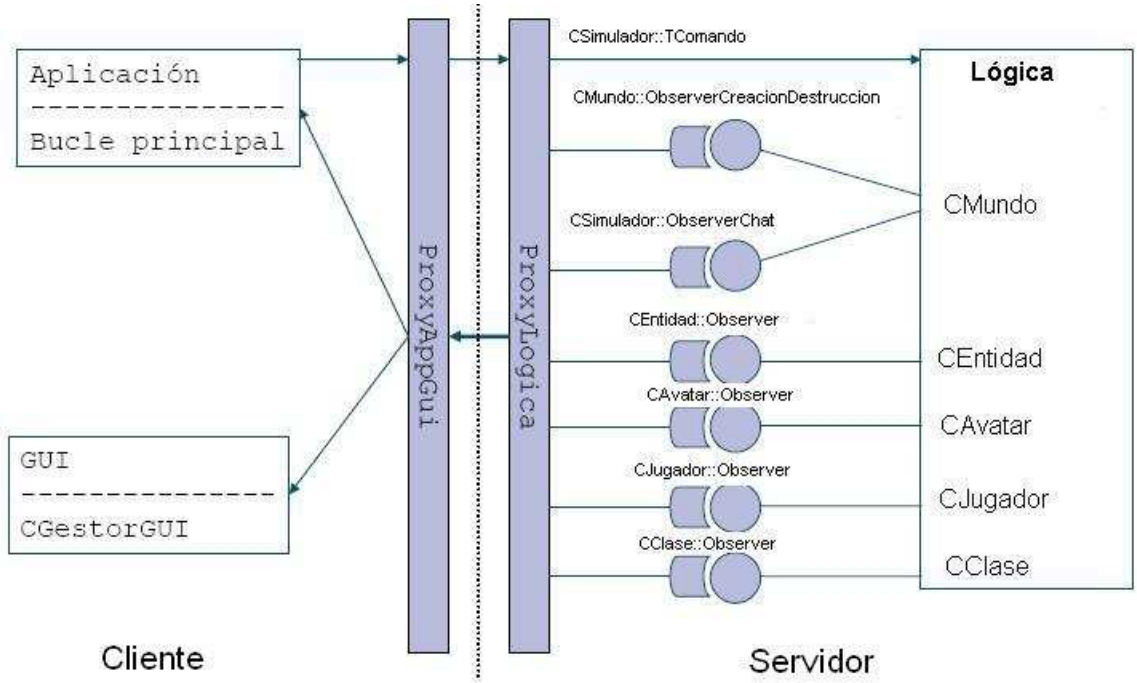

Figura 35. Esquema simplificado del mecanismo de proxys para la arquitectura de red

¿Cómo se intercambia la información?

Ya hemos visto como está montada la arquitectura, pero ¿qué estructura tienen los paquetes que se envían? La información se encapsula en paquetes que se enviarán a través de red. Esta información enviada ha de serializarse a fuerza bruta, ya que C++ no dispone de librerías estándar que permitan serializar de manera automática. Para ello se tiene una clase buffer, en la cual escribiremos los atributos a serializar byte a byte, este buffer será el campo de datos del paquete que viajará a través de la red.

Tipo Paquete	Mensaje	Dirección de Memoria de la clase	Atributo1	Atributo2	….	AtributoN

Figura 36. Esquema de paquete de datos

En la figura podemos ver en detalle el esquema de un paquete, distinguimos los campos:

Tipo Paquete: puede ser de datos o de control. Los de control se usarán para controlar estados de conexión.

Mensaje: Identifica el tipo de mensaje del paquete. Puede ser por ejemplo uno de selección de personaje, de movimiento, etc.

Dirección de la clase: Este atributo es útil, porque supongamos que un objeto se ha movido en la lógica del servidor. El servidor envía el puntero al cliente, sin embargo, en el cliente esa dirección de memoria no tiene sentido. Para evitar este problema, en el momento de creación de entidades, se manda la dirección de memoria de cada entidad, por otro lado, en el cliente se tendrá una tabla hash que mapee direcciones de memoria con las direcciones correspondientes en su lógica, de esta forma el cliente podrá identificar qué entidad hay que actualizar.

AtributoN: Serialización del atributo byte a byte. Los strings se serializan de forma especial, pasando primero su longitud para saber cuántos caracteres habrá que leer en el destino.

Comunicación y protocolo

- ***Comunicación cliente-servidor***

Una vez visto qué forma tienen los paquetes, veamos ejemplos concretos de los paquetes y más en concreto como se desarrolla el juego multijugador.

El intercambio de información por red se trata de dos formas diferentes, dependiendo si la ha partido ha comenzado, en cuyo caso se intercambiarán los diferentes comandos para realizar las acciones, o en la fase donde la partida aun no ha comenzado, en cuyo caso habrá que identificar las conexiones, seleccionar escenarios, roles, etc.

Se comenzará explicando el primer caso, que es el principal y es donde entra en juego la arquitectura de proxys previamente explicada.

- ***Comunicación desde el cliente***

El proxy del cliente (llamado ProxyAppGUI) se comunica con el servidor mediante la inserción de comandos:

```
void CProxyAppGui::insertarComando(logica::CSimulador::TComando comando) {
	red::Mensaje msg = redLaberinto::COMMAND;
	red::CBuffer buffer(500,100);
	buffer.write(serializarComando()); //Serializa el comando como corresponda
	clienteRed->sendData(conexionClienteRed, buffer.getbuffer(), buffer.getSize(),0,0);
}
```

Y para la comunicación del servidor con el cliente, se dispone de un método actualizar que está continuamente leyendo paquetes por red, lo identifica y lo gestiona con el proxy que corresponda.

```
void CProxyAppGui::actualizar(unsigned int dt)  {
	std::vector<red::CPaquete*> paquetes;
	clienteRed->service(paquetes);		//lee paquetes de red

	for(std::vector<red::CPaquete*>::iterator iter = paquetes.begin(); iter != paquetes.end(); ++iter){
			redLaberinto::Mensaje msg;
			red::CBuffer buffer(500,100);
			buffer.write(paquete->getData(),paquete->getDataLength());
			buffer.reset();
			buffer.read(&msg, sizeof(msg));
			if(msg == red::CLASE_DESTRUIDA) {...} //Trata el paquete
			else if(msg == redLaberinto::POSICION_CAMBIADA) {...}  //Trata el paquete
			....
	}
}
```

Aunque hay más de 30 tipos de paquetes de datos, se pueden clasificar en varios grupos, de forma que todos los paquetes de un mismo grupo se tratan de forma casi idéntica. Los grupos que podemos distinguir son:

- Paquetes de creación/destrucción: Cuando se crea/destruye una clase de la lógica en el servidor, se notifica a los clientes para que tengan una copia de esa clase, como ya se explicó anteriormente, estas clases son marionetas, ya que por sí solas no hacen nada, están ahí para que el cliente las vaya actualizando bajo las órdenes del servidor, y la GUI pueda usarlas para pintar.

- Paquetes de movimiento: Son los que se encargan de actualizar a las entidades cuando se desplazan o giran.
- Paquetes de inventario: Modifican el inventario para que se muestre por pantalla correctamente.
- Paquetes de pelota: Aunque el movimiento de la pelota ya está gestionado por los paquetes de movimiento. La bola dispone también de muchas otras animaciones, las cuales se gestionan aquí.
- Paquetes de chat: El intercambio de mensajes de los clientes.

Veamos ahora un ejemplo de cada tipo de paquete para que se entienda el esquema que se sigue:

- ***Paquetes de creación/destrucción***

Su funcionamiento es siempre, primero deserializar la clase la cual hay que construir/destruir, identificarla y crearla/destruirla. Se muestra a continuación de un ejemplo de cómo se crea un objeto en el escenario:

```
if(msg == red::OBJETO_CONSTRUIDO) {
	logica::CObjeto* objeto = new logica::CObjeto(simulador->getMundo());  //Creamos la clase
	logica::CObjeto* objetoremoto;
	buffer.read(&objetoremoto, sizeof(objetoremoto));
	objeto->deserialize(buffer);  //Se deserializa la clase y se identifica
	simulador->getMundo()->objetos.push_back(objeto);
	simulador->getMundo()->entidades.push_back(objeto);
	remote2local[objetoremoto]=objeto;  //Se mapea para tener su referencia
	simulador->getMundo()->emitObjetoNuevo(objeto);
}
```

- ***Paquetes de movimiento***

Su esquema puede ser un poco complicado, ya que el servidor pasa una referencia del objeto que quiere moverse, y ese puntero no tiene sentido en el cliente. Esto se resuelve en los paquetes de creación destrucción, que mapean las referencias para posteriormente poder identificarlos. Ponemos un ejemplo de cómo el cliente notifica los cambios de posición de las entidades del juego:

```
if(msg == red::DESPLAZAMIENTO_CAMBIADO) {
	logica::CEntidad* entidadremota;
	buffer.read(&entidadremota, sizeof(entidadremota));

	//Se identifica la entidad
	std::map<void*,void*>::iterator iterFind = remote2local.find(entidadremota);
	if( iterFind == remote2local.end() ) assert("objeto remoto no mapeado a local");
	logica::CEntidad* entidad = (logica::CEntidad*)iterFind->second;

	//Se deserializa obteniendo los atributos necesarios para pintarla
	entidad -> deserialize(buffer);
	bool andando, retrocediendo;
	buffer.read(&andando, sizeof(retrocediendo));
	buffer.read(&retrocediendo, sizeof(retrocediendo));


	//Se actualiza
	if(andando)
			((logica::CAvatar*)entidad)->andar();
	else if (retrocediendo)
			((logica::CAvatar*)entidad)->retroceder();
	else
			((logica::CAvatar*)entidad)->parar();
}
```

- ***Paquetes de inventario***

Se tratan de forma muy sencilla. Se deserializan sus atributos y se modifican, de forma que tras la modificación la GUI será notificada y lo actualizará como corresponda. Se muestra un ejemplo de cómo actualizar la lista de invocados en el inventario de crear un mensaje:

```
else if (msg == red::LISTA_INVOCADOS) {

	//Se deserializan los campos
	std::list<std::string> lista;
	std::string nombre = this->deserializeString(buffer);
	std::string personaje = aplicacion::CGestorConfig::getPersonaje();
	int nobj;
	buffer.read(&nobj,sizeof(nobj));
	std::string obj;
	for (int i = 0; i < nobj;i++){
			obj = this->deserializeString(buffer);
			lista.push_back(obj);
	}

	//Se comunica
	simulador->getInventario()->emitListaInvocados(nombre,lista);
}
```

- ***Paquetes de pelota***

Se gestionan de forma similar a los del inventario, se obtienen los atributos y se actualiza. Veamos el ejemplo en el que

hay que gestionar la animación del lanzamiento de la pelota, que puede ser por arriba o por abajo.

```
if(msg == red::LANZANDO_PELOTA) {
	logica::CPelota *pelota = simulador -> getMundo() -> getPelota();
	bool porArriba;
	buffer.read(&porArriba, sizeof(porArriba));  //Se deserializan los campos
	pelota -> getMensaje() -> setEsInvocado(porArriba);  //Se actualiza la pelota
	pelota->emitLanzando();  //Se comunica
}
```

- ***Paquetes de chat***

También son simples, se deserializa el texto a mostrar y se añade al simulador de forma que tras la modificación el GUI se actualice como corresponda. A continuación un ejemplo de cómo se gestionan los mensajes de chat principalmente:

```
else if(msg == red::CHAT) {
	std::string us,tex;
	/Se deserializan los strings
	us = this->deserializeString(buffer);
	tex = this->deserializeString(buffer);
	simulador->addEntradaChat(us,tex); /Se añaden
}
```

- ***Comunicación desde el servidor***

Acabamos de ver como se trataban los paquetes desde el lado del cliente. En la lógica es ligeramente más complicado: Cada cambio producido en alguna entidad, disparará su correspondiente observer. La idea es tener varios proxys, cada uno escuchando el observer que corresponda, de forma que cuando se modifique algo, ese proxy se entere y comunique inmediatamente la acción al cliente para que se actualice.

El proxy de lógica, aparte de tener como misión principal crear paquetes para los clientes, también debe saber recibir información del cliente, que se hace por medio de los comandos. Recordemos que el proxyAppGUI tenía un método para insertar los comandos, veamos ahora como se reciben en la lógica:

```
void CProxyLogica::actualizar(unsigned int dt) {
	std::vector<red::CPaquete*> paquetes;
	servidorRed->service(paquetes);

	for(std::vector<red::CPaquete*>::iterator iter = paquetes.begin(); iter != paquetes.end(); ++iter) {
			red::CPaquete* paquete = *iter;
			red::Mensaje* msg;
			msg = (red::Mensaje*)paquete->getData();
			if(*msg == red::COMMAND) {
					red::Mensaje msg2;
					red::CBuffer buffer(500,100);
					buffer.write(paquete->getData(),paquete->getDataLength());
					buffer.reset();
					buffer.read(&msg2, sizeof(msg2));
					logica::CSimulador::EIdComando id;
					buffer.read(&id, sizeof(id));
					bool activar;
					buffer.read(&activar, sizeof(activar));
					float radioGiro;
					buffer.read(&radioGiro, sizeof(radioGiro));
					std::string identificador = deserializeString(buffer);
					std::string campo = deserializeString(buffer);
					std::string valor = deserializeString(buffer);
					std::string texto = deserializeString(buffer);
					simulador->insertarComando(logica::CSimulador::TComando(id, activar,
							identificador, radioGiro, campo, valor, texto));
			}
	}
}
```

Los paquetes también pueden asociarse en grupos, de hecho se asocian de la misma forma que en el cliente, cosa lógica ya que cada proxy en la lógica tiene su correspondiente pareja en el cliente. Para ilustrarlo, vamos a poner los mismos ejemplos que se pusieron en el cliente, pero desde el otro lado.

- ***Paquetes de creación/destrucción***

Estos tienen una particularidad especial. Cuando se crea una entidad en el servidor, se le asociaran los proxys que se encargarán de gestionarlo. Veamos el ejemplo del objeto construido, observar que se le asocian 2 proxys, el de entidad, ya que un objeto sigue siendo una entidad y el proxy de objeto, que gestionará el movimiento y los cambios de orientación.

```
void CProxyLogica::objetoConstruido( const logica::CMundo *mundo, logica::CObjeto *objeto) {
	// Se añaden los proxys que gestionarán los objetos
	((logica::CAvatar*)objeto)->addObserver(proxyav);
	((logica::CEntidad*)objeto)->addObserver(proxyent);

	// Se crea el paquete correspondiente que luego el cliente deserializará en el orden adecuado
	red::Mensaje msg = redLaberinto::OBJETO_CONSTRUIDO;
	red::CBuffer buffer(500,100);
	buffer.write(&msg, sizeof(msg));
	buffer.write(&objeto, sizeof(objeto));
	((logica::CObjeto*)objeto)->serialize(buffer);

	// Se envía el paquete a todos los clientes
	servidorRed->sendAll(buffer.getbuffer(), buffer.getSize(),0,0);
}
```

- ***Paquetes de movimiento***

Estos paquetes serán creados por el proxy de entidad, que será el encargado de notificar cuando una entidad ha cambiado de posición, y el proxy de avatar, que se encargan de notificar cuando un objeto tiene intención de anda, girar, retroceder, etc. Veamos el ejemplo que vimos en el cliente con el desplazamiento cambiado. Observar, como antes de serializar los atributos, se serializa la dirección de memoria de la entidad. Esto se debe a lo que venimos contando, el cliente necesita saber que avatar ha cambiado su desplazamiento, por eso enviamos la dirección de memoria, de forma que luego el cliente la mapeará y podrá organizarse.

```
void CProxyLogicaAvatar::desplazamientoCambiado(const logica::CEntidad *entidad,  bool andando,bool
retrocediendo) {

	red::Mensaje msg = redLaberinto::DESPLAZAMIENTO_CAMBIADO;
	red::CBuffer buffer(500,100);
	buffer.write(&msg, sizeof(msg));
	buffer.write(&entidad, sizeof(entidad)); // Se serializa la referencia a la entidad
	((logica::CAvatar*)entidad)->serialize(buffer);
	buffer.write(&andando, sizeof(andando));
	buffer.write(&retrocediendo, sizeof(retrocediendo));

	// Se envía el paquete a todos los clientes
	servidorRed->sendAll(buffer.getbuffer(), buffer.getSize(),0,0);
}
```

- ***Paquetes de inventario***

Se gestionan por el proxyLogicaInventario. Su funcionamiento es similar a los vistos anteriormente, serializar lo

necesario y enviar el paquete. En este caso ya no es necesario serializar referencias, ya inventario solo hay uno. A continuación el ejemplo de la lista de invocados en el lado del servidor:

```
void CProxyLogicaInventario::listaInvocados(std::string jugador,std::list<std::string> listaObjetos) {

	// Se crea el paquete y se serializan los atributos
	red::Mensaje msg = redLaberinto::LISTA_INVOCADOS;
	red::CBuffer buffer(500,100);
	buffer.write(&msg, sizeof(msg));
	serializeString(buffer,jugador);
	serializarListaString(buffer,listaObjetos);

	// Se envía el paquete a todos los clientes
	servidorRed->sendAll(buffer.getbuffer(), buffer.getSize(),0,0);
}
```

- ***Paquetes de pelota***

Los gestiona el proxyPelota. Su esquema es como siempre. Veamos de nuevo el ejemplo de lanzando la pelota desde el lado del servidor.

```
void CProxyLogicaPelota::lanzando(const logica::CPelota *pelota, bool porArriba) {

	/ Se crea el paquete y se serializan los atributos
	red::Mensaje msg = red::LANZANDO_PELOTA;
	red::CBuffer buffer(500,100);
	buffer.write(&msg, sizeof(msg));
	buffer.write(&porArriba, sizeof(porArriba));

	// Se envía el paquete a todos los clientes
	servidorRed->sendAll(buffer.getbuffer(), buffer.getSize(),0,0);
}
```

Ya hemos visto como se realiza la comunicación en la fase de juego a través de proxys, sin embargo falta por especificar cómo se realiza la comunicación antes de entrar en juego. En esta parte no son necesarios los proxys, principalmente porque inicialmente no se saben las conexiones entrantes, ni cuantos jugadores habrá, etc. Por eso hay que tratarlo de forma ligeramente diferente.

Esta inicialización, se realiza a través de dos estados: El estado de seleccionar el escenario y el estado de seleccionar personaje.

El estado de escenarios es bastante simple y ya se ha explicado anteriormente. En este estado solo interviene el servidor.

El estado de selección de personaje está presente en el lado del cliente y en el lado del servidor. Sus principales funciones son:

- ***Selección de personaje en el servidor:***

La selección de personaje es un estado más difícil de lo que parecería a simple vista. Sus funciones son:

1. Gestionar la entrada de conexiones entrantes por parte de clientes, de forma que al llegar una conexión, almacenarla y notificar al cliente para que cambie al estado de seleccionar personaje.
2. Presentar una lista al cliente sobre los personajes a elegir
3. Gestionar la selección teniendo cuidado de los personajes que están disponibles o ya han sido elegidos e informar a todos los clientes de los cambios sufridos.
4. Lanzar la partida cuando corresponda. Para ello, se mandará un paquete de tipo EMPEZAR_JUEGO cuando todos los roles del escenario hayan sido seleccionados. Como excepción a esta regla, se comenzará el juego también cuando el instructor haya elegido role, independientemente de que el resto de roles hayan sido escogidos.

Veamos un poco en detalle su funcionamiento. Observemos la función principal de esta clase. Se leen paquetes continuamente y en función del tipo se realiza una de las funciones arriba expuestas.

```
void CestadoSelPersonaje::procesarEntradaUsuario() {
        std::vector<red::CPaquete*> paquetes;
       _appNebula->servidorRed->service(paquetes);

       for(std::vector<red::CPaquete*>::iterator iter = paquetes.begin();iter != paquetes.end();++iter) {
                    red::CPaquete* paquete = *iter;
                    if( paquete->getTipo() == red::CONEXION) {
                                 // Gestiona conexiones entrantes y manda el paquete para que el cliente
                                 // cambie al estado de selección de personajes
                                 ...
                    }
                    if( paquete->getTipo() == red::DATOS) {
                                 red::Mensaje msg;
                                 memcpy(&msg, paquete->getData(),sizeof(msg));
                                 if(msg == red:GET_PERSONAJES) {
                                              //  Envia la lista de personajes disponibles al cliente
                                              ...
                                 }
                                 else if(msg == red::SET_PERSONAJE) {
                                              // Marca el personaje seleccionado y envía confirmación al cliente
                                              ...
                                 }
                                 else if(msg == red::UNSET_PERSONAJE) {
                                              // Deja el personaje libre y envía confirmación al cliente
                                 }
                    }
       }

      // Si ya todos los personajes están seleccionados, manda paquete de comenzar partida
       if (remaining == 0) {
                     red::Mensaje msg = redLaberinto::EMPEZAR_JUEGO;
                                 _appNebula->servidorRed->sendAll(&msg, sizeof(msg),0,1);
                                 _app->estableceEstado("juegoservidor");
       }
 }
```

- ***Selección de personaje en el cliente:***

5. Mostrar el menú de selección de roles (Ver figura 37).
6. Mandar paquete de tipo GET_PERSONAJES solicitando la lista de personajes del escenario.
7. Mandar paquete de tipo SET_PERSONAJE solicitando un role concreto, para el cual, el servidor responderá con un ACK dependiendo de si el personaje está disponible o no.
8. El cliente también podrá modificar un personaje mientras la partida no haya finalizado.

Veamos ahora en un poco en detalle la función principal de seleccionar roles en el lado del cliente.

```
void CestadoSelPersonajeCliente::procesarEntradaUsuario() {
	red::CPaquete* paquete = _appNebula->clienteRed->readPacket();
	if(paquete !=NULL && paquete->getTipo() == red::DATOS) {
			redLaberinto::Mensaje msg;
			memcpy(&msg, paquete->getData(),sizeof(msg));
			if(msg == red::GET_PERSONAJES) {
				// Obtiene los personajes para mostrarlos por pantalla
				...
			}
			else if(msg == red::SET_PERSONAJE) {
				//Lee el paquete del servidor y comprueba si su personaje pudo ser elegido
				//notifica por pantalla la resolución
				...
			}
			else if(msg == red::UNSET_PERSONAJE) {
				//Lee el paquete del servidor y comprueba su pudo deseleccionar al
personaje
				...
			}
			else if(msg == redLaberinto::EMPEZAR_JUEGO) // El servidor da permiso para empezar
				_appNebula->estableceEstado("juegored");
			}
}
```

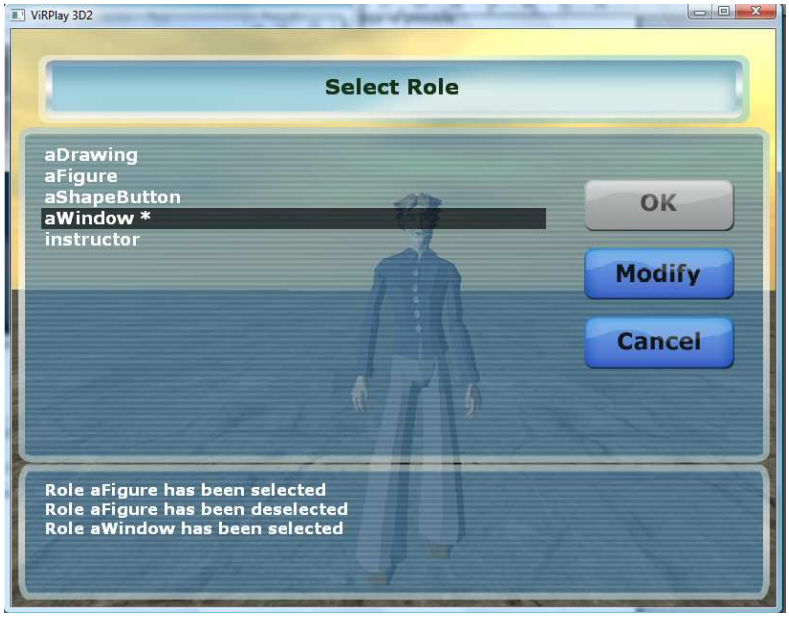

Figura 37. Menú de selección de role

Finalmente ponemos un diagrama de secuencia (Figura 38) para que se vea el funcionamiento de cliente y servidor a la vez, y ver como intercambian paquetes hasta comenzar la partida.

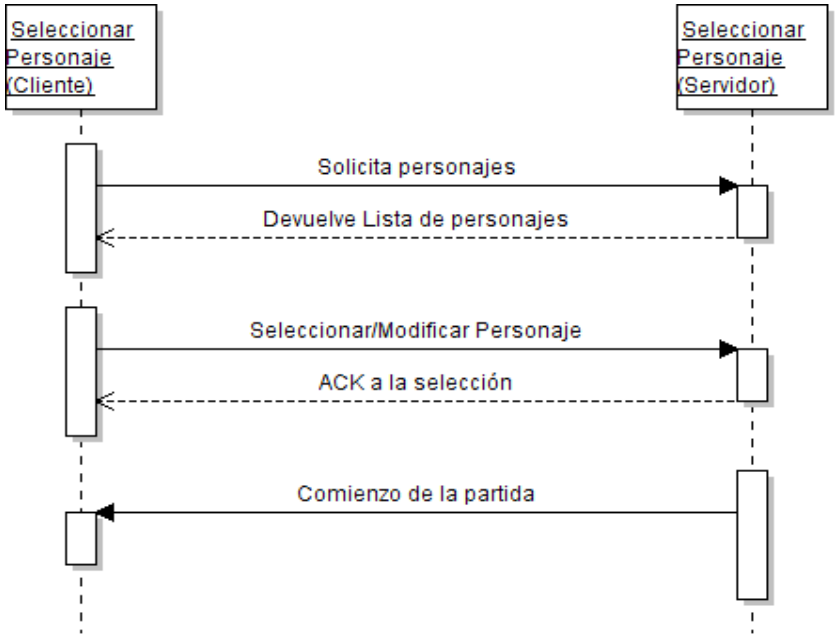

Figura 38. Diagrama de secuencia de la selección de personaje

5.2.7 Chat

Para permitir a los clientes comunicarse entre sí se ha desarrollado una herramienta de comunicación o chat. El chat no es más que una interfaz donde los usuarios pueden introducir mensajes en forma de cadena de texto y enviarlos a todos los clientes que estén conectados al servidor. Cada mensaje aparece identificado por el nombre del personaje en el juego. En cualquier momento del juego el cliente puede desplegar está herramienta y comunicarse con el resto de clientes. El mensaje enviado será entregado a todos los clientes y quedará almacenado. Como la lógica está centralizada en el servidor todos los mensajes que se deseen enviar tienen que pasar previamente por este, para gestionarlos y realizar los cambios oportunos en la lógica. Finalmente los clientes serán informados de estos cambios mediante el mecanismo de observers.

El aspecto de la interfaz gráfica del chat es el siguiente:

Figura 39. Interfaz gráfico del chat

- ***Arquitectura del chat***

El chat a nivel lógico está constituido por una lista de mensajes. Para gestionar de manera sincronizada todos los cambios por nuevos mensajes introducidos, se usa el patrón de diseño Observer. Con esto conseguimos que cada vez que alguien escriba algún mensaje nuevo, nos avise de este cambio y podamos mostrar el mensaje en todos los clientes.

En el juego la forma de acceder al chat es mediante un estado de juego. Al estar jugando si pulsamos la tecla *tabulación* pasamos al estado "chat" que es el que gestiona todo para que se muestre la interfaz gráfica y podamos escribir los mensajes. En este estado se realiza inicialmente la carga del chat almacenado, de forma que, cada vez que abramos la herramienta de comunicación, podamos ver el historial de mensajes enviados. Con el historial en pantalla el usuario puede introducir el mensaje y enviarlo. Si el juego se está desarrollando en modo multijugador este mensaje llegará a todos los clientes precedido del nombre que desempeña el usuario en el juego.

Para que se pueda llevar a cabo el mecanismo de comunicación en red, se hace uso de las proxys, tanto del lado del servidor como del cliente, para que gestionen correctamente la transferencia de la información. Para esto se usa el patrón Observer de forma que cada vez que haya un cambio, se notifique a la Proxy del servidor (CProxyLogicaInventario), para enviarlo por red. Una vez recibido en los clientes, por mediación

de su Proxy (CProxyAppGui), se notifica al subEstadoChat para que muestre por pantalla estos nuevos cambios introducidos.

Una vez que se cierre la aplicación se ha diseñado una herramienta que permita guardar el chat. De esta forma podemos examinar el historial de conversación de una partida desarrollada con anterioridad.

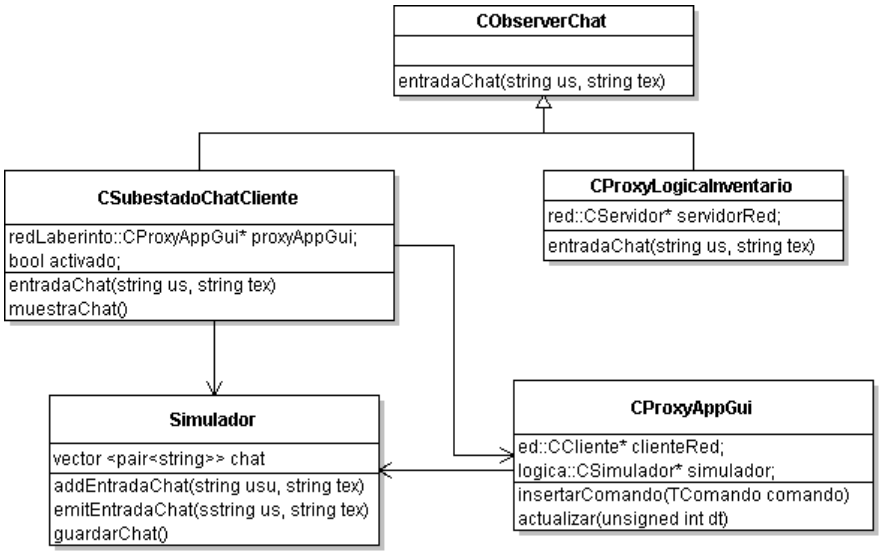

Figura 40. UML de la herramienta de comunicación

- ***Secuencia de ejecución***

Un ejemplo de ejecución para usar la herramienta de comunicaciones seguiría el siguiente desarrollo. El cliente que desea mandar un mensaje pulsa la tecla *tabulación* y pasa al estado chat. Desde este estado puede visualizar los mensajes ya enviados anteriormente. Para mandar un mensaje rellena el campo de texto en la parte inferior y pulsa la tecla *entrar* o pincha en el botón enviar. Automáticamente se manda mediante la proxyAppGui el comando chat, junto con la información necesaria como es el nombre del objeto y el mensaje enviado.

Cuando el servidor recibe este comando, mediante la proxyLogicaInventario, lo procesa, de forma que actualiza la información desde el simulador con esta nueva información. Al realizar esta actualización mediante el mecanismo de observers se comunica a la proxyLogica este cambio, de forma que se serializa por red la nueva entrada. Cada cliente recibe este

cambio mediante la proxyAppGui y actualiza su lógica en su simulador propio. Al actualizar la lógica en el cliente mediante un Observer se avisa a la interfaz gráfica para que muestre el nuevo mensaje. Una vez terminada la aplicación el chat queda guardado en el lado del servidor.

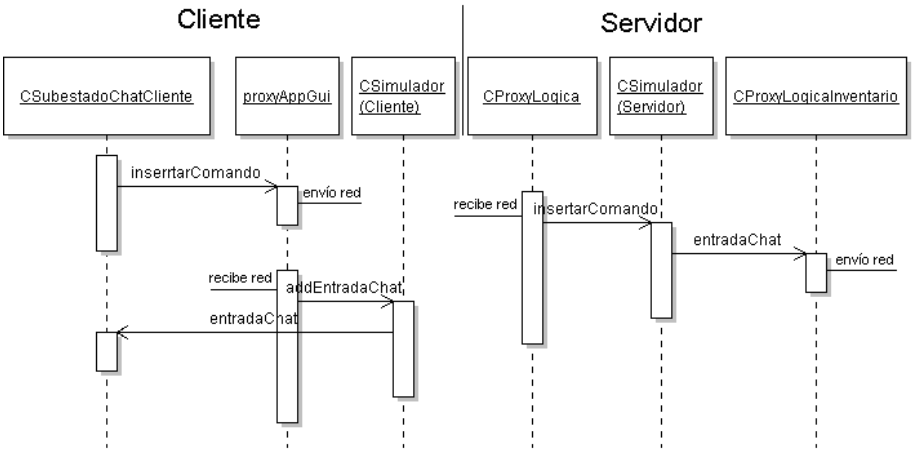

Figura 41. Diagrama de secuencia de la herramienta de comunicación

6 *Conclusiones*

ViRPlay3D es una herramienta educativa que surge para facilitar la comprensión de la programación orientada a objetos. Para conseguir esto, la aplicación hace uso de la técnica de role-play, que facilita la labor de aprendizaje activo, haciendo que los usuarios sean parte del desarrollo del programa.

ViRPlay3D2 utiliza esta idea de una forma más directa, ya que añade el desarrollo de partidas en modo multijugador, con lo que se permite explotar la técnica de role-play, de modo que varios usuarios puedan utilizar la herramienta al mismo tiempo y en la misma partida.

De esta forma la aplicación simula las sesiones de role-play similares a las clases presenciales en la enseñanza de patrones de diseño

A modo de resumen, las aportaciones principales de este proyecto a la versión anterior de ViRPlay3D2 son:

- ***Expansión del modo monojugador:*** Se han ampliado las funcionalidades existentes y se han añadido otras nuevas con el objetivo de preparar la aplicación para el modo multijugador. Algunas de las mejoras añadidas son: creación de una entidad lógica pelota, posibilidad de creación de mensajes, nuevos inventarios, etc.
- ***Integración del modo multijugador al juego:*** Se ha cambiado la arquitectura de forma que se pueda soportar modo multijugador. Se permiten que varios usuarios, desde distintas máquinas, puedan participar conjuntamente en un escenario del juego.
- ***Creación de un sistema complejo de inventarios:*** Se permite consultar la información de todas las entidades del mundo: clases, objetos y pelota.
- ***Simulación automática de un escenario:*** A partir de un fichero XML se permite la simulación de un escenario.
- ***Construcción de mensaje:*** Se permite la construcción de nuevos mensaje de ejecución.
- ***Selección de escenarios:*** Se permite la selección del escenario del juego al inicio de la partida.

- ***Extensión de escenarios***: Se permite la integración de nuevos escenarios en el formato adecuado.
- ***Selección de personajes***: Según el escenario seleccionado se pueden escoger los personajes para el juego.
- ***Herramienta de comunicación***: Se permite la comunicación con todos los clientes conectados mediante un chat.
- ***Selección de las opciones de red***: Mediante dos menús, uno para el servidor y otro para el cliente, se permiten modificar las opciones de conexión de red.

7 Bibliograía

RECIO GARCÍA J. A., Práctica básica de juegos en red.
Grupo de Aplicaciones de Inteligencia Artificial.
Universidad Complutense de Madrid

RAYA ROA I., Desarrollo de juegos multijugador en red.
Departamento de Sistemas Informáticos y Programación.
Universidad Complutense de Madrid. 2007.

JIMÉNEZ DÍAZ G., Entornos virtuales basados en técnicas de aprendizaje activo para la enseñanza de la orientación a objetos.
Departamento de Ingeniería del Software e Inteligencia Artificial.
Universidad Complutense de Madrid. 2008.

GAMMA, E; HELM, R; JOHNSON; VLISSIDES ,J.
Design Patterns: Elements of Reusable Object-Oriented Software

SALZAN, L., Documentación sobre la librería eNet.
[http://enet.bespin.org]

CPLUSPLUS.COM, Referencias sobre C++.
[http://www.cplusplus.com/reference/]

C++ REFERENCE, Referencias sobre C++.
[http://www.cppreference.com/]

RADON LABS GmbH, Documentación sobre Nebula.
[http://www.radonlabs.de/]

MICROSOFT CORPORATION, Documentación sobre Microsoft visual Studio
[http://msdn.microsoft.com/en-us/vstudio/default.aspx]

8 *Palabras Clave*

- Role-play
- Videojuego Multiusuario
- Aplicación educativa
- Enseñanza
- Orientacion a Objetos
- Patrones de Diseño

9 *Tabla de Figuras*

10 *Autorización*

Los abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Pablo Fraile García Guillermo del Fresno Herena Ricardo Gómez Miguélez

Madrid, a 04 de julio de 2008