# El juego seguro de la Oca

**Daniel Celemín García**
Ingeniería Técnica en Informática de Sistemas (ETIS)

**Antoni Martínez Balleste**

17 de junio de 2004

*Este trabajo y lo que significa*
*esta dedicado a mi mujer, Emma*
*por estar siempre a mi lado*
*durante el tiempo de estudio*
*en los momentos buenos y en los malos*

# 1 Resumen

El objetivo de este proyecto, es el desarrollo de una aplicación que implemente la funcionalidad del ***juego de la oca*** de forma segura, con n jugadores distribuidos en diferentes ordenadores con direcciones IP y puertos diferentes conectados a través de una red, como puede ser Internet o una Intranet o red corporativa, entendiendo la jugabilidad de forma segura como el envío y recepción de mensajes entre estaciones sin la posibilidad de intromisiones ajenas a los jugadores y la cooperación fiable en el lanzamiento de los dados necesarios para la actualización de las posiciones de los jugadores durante el juego, todo ello gracias al uso de métodos para la comunicación entre estaciones que se basan en la utilización del protocolo de comunicación criptográfico SSL sobre librerías de sockets, la utilización de claves simétricas tipo DES para el cifrado de datos y claves de sesión, o las técnicas de compromiso de BIT para el lanzamiento del dado. El ámbito de aplicación de este proyecto se engloba en el marco de la ***seguridad en la comunicación*** entre procesos, así como la interacción de operaciones y ***cálculos conjuntos*** sin posibilidad de "engaño" por parte de los participantes y de los ***juegos de azar*** por Internet. El resultado es una aplicación punto-a-punto distribuida capaz de mantener una información fiable de la posición global de juego desde cada jugador (sincronización del estado de la partida), así como proporcionar una interfaz amigable para el propósito del proyecto, el *juego de la oca*.

Palabras clave:

SSL, compromiso de BIT, comunicación punto-a-punto, dirección IP, puerto, Intranet, socket, clave simétrica DES, clave de sesión, interfaz.

# 2 Índice General


**1 RESUMEN ..... 3**

**2 ÍNDICE GENERAL ..... 4**

**3 ÍNDICE DE FIGURAS ..... 7**

**4 ÍNDICE DE TABLAS ..... 8**

**5 INTRODUCCIÓN ..... 9**

5.1 CONTEXTO ..... 9
5.2 OBJETIVOS ..... 9
5.3 METODOLOGÍA ..... 10
5.4 PLANIFICACIÓN ..... 11
5.5 MOTIVACIÓN ..... 12
5.6 PRODUCTO OBTENIDO ..... 12
5.7 ESTRUCTURA DE LA MEMORIA ..... 12

**6 ESTADO DEL ARTE ..... 13**

6.1 LOS SOCKETS ..... 13
*6.1.1 Un poco de historia ..... 13*
*6.1.2 Comunicaciones TCP y UDP. ..... 13*
6.2 EL PROTOCOLO SSL ..... 14
*6.2.1 Introducción ..... 14*
*6.2.2 Servicios proporcionados ..... 14*
6.2.2.1 Confidencialidad ..... 14
6.2.2.2 Autenticidad ..... 15
6.2.2.3 Integridad ..... 15
*6.2.3 Descripción del protocolo de negociación ..... 15*
*6.2.4 Análisis de seguridad. ..... 16*
6.3 EL ESTÁNDAR DE CIFRADO DE BLOQUE DES ..... 17
*6.3.1 Introducción ..... 17*
*6.3.2 Funcionamiento ..... 17*
6.4 SITUACIONES DE DESCONFIANZA MUTUA. EL COMPROMISO DE BIT ..... 19

**7 HERRAMIENTAS UTILIZADAS ..... 21**

7.1 EL LENGUAJE DE PROGRAMACIÓN. JAVA. ..... 21
*7.1.1 Introducción ..... 21*
*7.1.2 Portabilidad ..... 21*
*7.1.3 Orientación a objetos ..... 21*
7.2 LAS COMUNICACIONES Y LA SEGURIDAD, TODO EN UNO. LA ARQUITECTURA JCA DE JAVA. 23
*7.2.1 La arquitectura criptográfica de Java. ..... 23*
7.2.1.1 JSSE ..... 23
7.2.1.2 criptografía de clave pública ..... 23
7.2.1.3 Certificados y autoridad de certificación ..... 24
7.2.1.4 SSL y TLS ..... 24
7.2.1.5 El protocolo de negociación de SSL ..... 24
7.2.1.6 Utilizando los sockets JSSE ..... 24
7.3 LA INTERFAZ PARA JAVA. AWT. ..... 26
*7.3.1 ¿Que es una interfaz de usuario? ..... 26*
*7.3.2 Componentes y contenedores ..... 26*
*7.3.3 Tipos de componentes ..... 27*

7.3.4 Tipos de contenedores ... 27
7.3.5 Creando un contenedor ... 27
7.3.6 Las decisiones de comportamiento de los componentes ... 27

**8 ANÁLISIS DE LA APLICACIÓN ... 28**

8.1 IDEA INICIAL. LA FIGURA DEL INICIADOR DE PARTIDA ... 28
8.2 ARQUITECTURA PUNTO A PUNTO. PASO DE MENSAJES DE CONEXIÓN ... 28
8.3 FASES DE JUEGO ... 29
8.3.1 Fase 1. Inicialización. ... 29
8.3.2 Fase 2. Conexión. ... 29
8.3.3 Fase 3. Juego ... 30
8.3.3.1 El lanzamiento de los dados ... 30
8.4 CLASES DEL DOMINIO. ... 32
8.5 DIAGRAMA DE ESTADO ... 33
8.6 CASOS DE USO. ... 34
8.7 DOCUMENTACIÓN DE LOS CASOS DE USO. ... 35
8.7.1 Caso de uso. Inicialización de una partida ... 35
8.7.2 Caso de uso. Enrolarse a una partida ya iniciada ... 35
8.7.3 Caso de uso. Esperar jugadores entrantes ... 36
8.7.4 Caso de uso. Comenzar partida. ... 36
8.7.5 Caso de uso. Jugar turno. ... 37
8.7.6 Caso de uso. Lanzar dados. ... 37

**9 DISEÑO DE LA APLICACIÓN ... 38**

9.1 DIAGRAMAS DE SECUENCIAS ... 38
9.1.1 Diagrama del caso de uso Inicialización de partida ... 38
9.1.2 Diagrama del caso de uso enrolarse a una partida. ... 39
9.1.3 Diagrama de secuencia esperar jugadores entrantes ... 40
9.1.4 Diagrama de secuencia comenzar partida ... 40
9.1.5 Diagrama de secuencia jugar turno ... 41
9.2 DISEÑO DE LAS CLASES DEL DOMINIO ... 42
9.2.1 Clase Interfaz ... 42
9.2.2 Clase Partida ... 42
9.2.3 Clase Jugador ... 42
9.2.4 Clase Dado ... 42
9.2.5 Clase ImagenDado ... 43
9.2.6 Clase ImagenTablero ... 43
9.2.7 Clase ImagenFicha ... 43
9.2.8 Clase Mensajes ... 43
9.2.9 Clase StreamEntrada ... 43
9.2.10 Clase StreamSalida ... 43

**10 IMPLEMENTACIÓN ... 44**

10.1 LAS COMUNICACIONES SEGURAS Y EL PROTOCOLO DE ALTA DE JUGADORES ... 44
10.1.1 Iniciar una partida ... 45
10.1.2 Enrolarse a una partida ... 45
10.1.3 Esperar conexiones de jugadores ... 47
10.2 EL PROTOCOLO DEL LANZAMIENTO DEL DADO ... 48
10.3 LA INTERFAZ DEL JUEGO ... 49

**11 PRUEBAS ... 51**

**12 CONCLUSIONES ... 51**

**13 GLOSARIO ... 52**

**14 BIBLIOGRAFÍA ..... 53**

**15 ANEXOS ..... 54**

15.1 PAQUETE DE DISTRIBUCIÓN DE LA APLICACIÓN ..... 54
15.2 MANUAL DEL USUARIO ..... 54
*15.2.1 Requerimientos del sistema. ..... 54*
*15.2.2 Instalación. ..... 54*
*15.2.3 Comenzar la aplicación: argumentos de entrada. ..... 54*
*15.2.4 Jugando al juego de la Oca ..... 56*
*15.2.5 Errores más frecuentes ..... 57*
15.3 EL FICHERO GENERARKEYSTORES.BAT ..... 57
*15.3.1 Generando una pareja de claves ..... 58*
*15.3.2 Par de claves cliente/servidor ..... 58*
*15.3.3 Extraer las claves publicas ..... 59*
15.4 LOS FICHEROS ALMACENES DE CLAVES: *ALMACENPRIVADO.JKS* Y *ALMACENPUBLICO.JKS* ..... 59
15.5 LOS FICHEROS GRÁFICOS. EL GRAFICO DEL TABLERO Y DE LAS FICHAS. ..... 60
15.6 CÓDIGO FUENTE COMENTADO ..... 61
*15.6.1 Clase Base64. ..... 61*
*15.6.2 Clase Dado. ..... 68*
*15.6.3 Clase ImagenDado. ..... 71*
*15.6.4 Clase ImagenFicha. ..... 74*
*15.6.5 Clase ImagenTablero. ..... 77*
*15.6.6 Clase Interfaz. ..... 85*
*15.6.7 Clase Juego. ..... 92*
*15.6.8 Clase Jugador. ..... 99*
*15.6.9 Clase Mensajes. ..... 105*
*15.6.10 Clase Partida. ..... 106*
*15.6.11 Clase StreamEntrada. ..... 117*
*15.6.12 Clase StreamSalida. ..... 118*

# 3 Índice de figuras

*Figura 1. Metáfora del juego* ..... 9
*Figura 3. Llamadas a la librería SSL* ..... 14
*Figura 4. Esquema DES* ..... 18
*Figura 5. Esquema de la arquitectura criptográfica de Java (JCA)* ..... 23
*Figura 6. árbol de componentes de la interfaz de usuario* ..... 26
*Figura 7. Las relaciones de herencia de AWT* ..... 27
*Figura 9. Visión de los jugadores activos desde los diferentes procesos* ..... 29
*Figura 10. Calculo del dado de forma segura. utilización de la técnica de compromiso de BIT* 31
*Figura 12. Diagrama de clases del dominio* ..... 32
*Figura 13. Diagrama de estados de la aplicación* ..... 33
*Figura 14. Diagrama de casos de uso* ..... 34
*Figura 15. Diagrama de secuencias. Inicialización de partida* ..... 38
*Figura 16. Diagrama de secuencias. Enrolarse en una partida* ..... 39
*Figura 17. Diagrama de secuencias. Esperar jugadores.* ..... 40
*Figura 18. Diagrama de secuencias. Comenzar partida.* ..... 40
*Figura 19. Diagrama de secuencias. Jugar turno.* ..... 41
*Figura 20. Diagrama de clases del dominio definitiva* ..... 42
*Figura 31. Interfaz del juego de la Oca* ..... 50
*Figura 32. Interfaz del juego de la Oca (2)* ..... 56
*Figura 33.Grafico Tablero.png* ..... 60
*Figura 34. Gráficos de las fichas. Ficha1.png, ficha2.png, ficha3.png y ficha4.png* ..... 60
*Figura 35. Gráficos del dado. Dado1.png, dado2.png, dado3.png y dado4.png* ..... 60

# 4 Índice de tablas

*Tabla 1. Planificación del proyecto ..... 11*
*Tabla 2. Tabla de permutaciones σ:..... 17*
*Tabla 3. Tabla de permutaciones $\sigma^{-1}$: ..... 17*
*Tabla 4. opciones del software criptográfico keytool ..... 58*

# 5 Introducción

## 5.1 Contexto

El proyecto del juego de la oca podría englobarse dentro del marco de la seguridad de aplicaciones distribuidas con arquitectura punto a punto, pero a su vez se podría tomar como una solución práctica a diferentes protocolos y técnicas criptográficas, como por ejemplo, el compromiso de BIT, enfocado al lanzamiento de dados en los juegos de azar cooperativos a través de Internet.

Dentro de este contexto, se ha de trabajar con protocolos de seguridad que proporcionen autenticación de jugadores, integridad de los datos que circulan por la red insegura y la encriptación de dichos datos, así como herramientas de programación que permitan:

- La conexión de procesos identificados por diferentes direcciones IP y puertos de comunicación.
- La creación de una interfaz válida para el juego de la Oca.
- La generación de claves simétricas/asimétricas y su utilización

**Pero la máxima de este proyecto reside en el lanzamiento de los dados**: la implementación de algoritmos y protocolos capaces de lanzar un dado de forma conjunta y segura, sin la posibilidad de que ningún jugador pueda hacer "trampas" o cambiar el valor del dado calculado conjuntamente por todos y cada uno de los jugadores.

## 5.2 Objetivos

Podríamos interpretar el funcionamiento de la aplicación objetivo de este proyecto observando el siguiente dibujo:

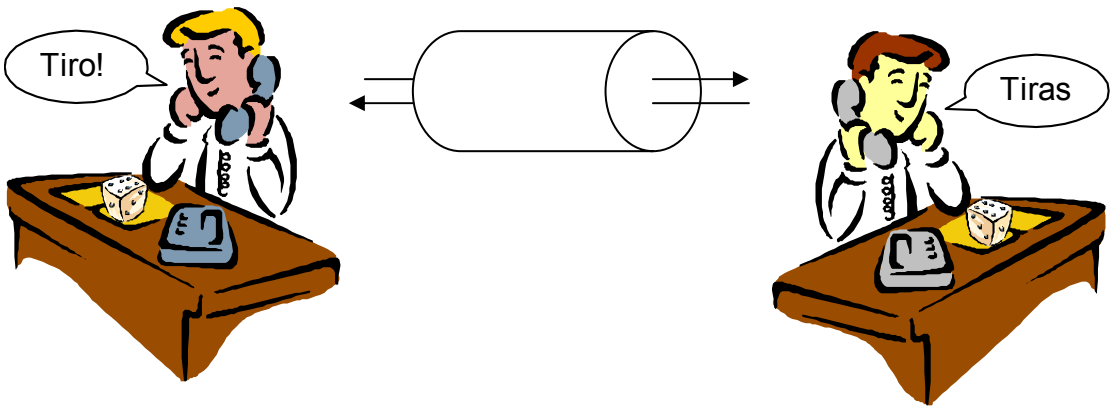


*Figura 1. Metáfora del juego*

Como se puede observar, las dos personas (***jugadores***) están jugando una ***partida*** del mismo ***juego***, el ***juego de la oca***. Las dos personas se comunican por teléfono (***sockets***), tienen un numero fijo (***par dirección IP-puerto***) y ambas tienen en su poder un ***dado*** que utilizan para mover su ***ficha*** a través del ***tablero*** del juego. Así mismo, han establecido un ***turno*** de juego mediante el cual pueden saber en cada momento quien tiene que mover su ***ficha***.

Siguiendo con la metáfora utilizada del dibujo, los objetivos de este proyecto son:

- Que ambas personas se comuniquen sin interrupciones, sin que nadie sepa que están jugando y sin que ninguna persona ajena al juego pueda hacerse pasar por uno de ellos.

- Que a la hora de lanzar el dado, puedan hacerlo de forma conjunta, sin que ninguno de los dos engañe al otro, estableciendo un protocolo que lo permita.

Mas formalmente, los objetivos del proyecto son implementar algoritmos y utilizar herramientas criptográficas y de interfaz para llevar a cabo la programación de la aplicación del juego de la oca teniendo en cuenta el contexto propuesto, es decir, utilizando herramientas de seguridad como por ejemplo el protocolo de comunicación SSL o claves criptográficas DES. También la comunicación entre n máquinas diferentes con n usuarios, direcciones IP y puertos distintos.

También es objetivo de este proyecto implementar una interfaz adecuada para permitir al jugador interactuar con la aplicación.

## 5.3 Metodología

El método utilizado para desarrollar el análisis, diseño e implementación de la aplicación se basa en los principios de la programación orientada a objetos.

De la misma forma que la programación orientada a objetos gira alrededor del concepto de clase, también giran el análisis de requisitos y el diseño. Por esta razón, el diagrama básico de estos métodos, el diagrama de clases y objetos, se utilizan tanto en el análisis como en el diseño; además, muchas de las clases descritas en el análisis de requisitos se implementan en los programas pasando por el diseño, cosa que hace el paso del análisis de requisitos al diseño sea más suave que en los métodos estructurados y también más sencillo y rápido.

Como dentro de una clase hay a la vez atributos y operaciones, es decir, datos y procesos, en el desarrollo orientado a objetos, a medida que se definen e implementan clases se avanza a la vez en estas dos dimensiones. El desarrollo no procede ni de manera *top-down* ni *bottom-up*, sino que se construyen grupos de clases interrelacionados, a menudo por niveles: Clases que gestionan la presentación de la información (interfaz), o bien las entradas por pantalla, o bien las lecturas y registros de la base de datos, o bien los algoritmos principales del programa, o bien según un ciclo de vida incremental.

El desarrollo orientado a objetos además de introducir técnicas nuevas, también aprovecha algunas técnicas y conceptos del desarrollo estructurado, como el diagrama de estados y transiciones, según veremos mas adelante.

Existen dos características del desarrollo orientado a objetos que probablemente han favorecido de manera decisiva su expansión hasta ahora:

- Permite la reutilización de programa en grado significativo, en forma de clases implementadas. En el caso de nuestro proyecto son ejemplos las librerías AWT para construir la interfaz, las librerías de sockets seguros utilizando el protocolo SSL o clases con licencia gratuita para codificar caracteres en base 64. Existen algunas más que veremos en el apartado de diseño de la aplicación, o bien,

- Facilita el desarrollo de herramientas informáticas de ayuda al desarrollo.

Para construir los diagramas de casos de uso, o de objetos se utiliza el modelo llamado UML, del cual es responsable la organización llamada OMG.

El Unified Modeling Language (UML) es un modelo para la construcción de programas orientados a objetos que han sido propuestos como estándar de ISO por la OMG. Consta de un conjunto de tipos de diagramas interrelacionados, dentro de los cuales se utilizan elementos del modelo, que sirven para describir diversos aspectos de la estructura y la dinámica del programa. UML es el resultado de una cierta unificación de los modelos utilizados en tres métodos preexistentes de desarrollo de programas orientados a objetos hechos por sus autores en colaboración.

## 5.4 Planificación

La planificación para la implementación de la aplicación viene explicada en la siguiente tabla, donde se indican las fechas y los objetivos de cada una de las etapas.

| Fechas | Objetivos |
|---|---|
| 1 – 7 marzo | Elaboración del plan de trabajo.<br>Búsqueda de información sobre lanzamiento de dados. |
| 8 – 14 marzo | Búsqueda de información sobre cifrado de comunicaciones, autenticación y sockets. |
| 15 – 21 marzo | Diseño del programa a nivel de procesos, comunicación e interacción. |
| 22 - 28 marzo | Diseño de la aplicación y decisión de funciones básicas. |
| 29 marzo - 4 abril | Desarrollo del programa en alto nivel. |
| 5 – 11 abril | Desarrollo del programa en alto nivel. |
| 12 – 18 abril | Implementación de la aplicación. |
| 19 – 25 abril | Implementación de la aplicación. |
| 26 abril – 2 mayo | Añadir el tablero y funciones multimedia. |
| 3 – 9 mayo | Memoria 1. Objetivo del proyecto, estado del arte y herramientas utilizadas. |
| 10 – 16 mayo | Memoria 1. Objetivo del proyecto, estado del arte y herramientas utilizadas. |
| 17 – 23 mayo | Memoria 2. Diseño del programa. |
| 24 – 30 mayo | Memoria 3. Pruebas, incidencias y conclusiones |
| 31 mayo – 6 junio | Memoria 4. Código fuente debidamente comentado |
| 7 – 13 junio | Presentación del proyecto con PowerPoint. |
| 14 – 28 junio | Presentación del proyecto con PowerPoint. |

*Tabla 1. Planificación del proyecto*

- **Elaboración del plan de trabajo**: Consiste en la redacción de esta planificación. Para ello se ha tenido que buscar información sobre el proyecto, dando una idea general de las materias, herramientas y demás aspectos que se desarrollan en este trabajo.

- **Búsqueda de información sobre cifrado de comunicaciones, autenticación y sockets**: En esta fase de búsqueda, se obtendrá la información relativa a comunicaciones (sockets) y seguridad (cifrado y autenticación). La librería de Java JSSE proporciona todos estos servicios.

- **Diseño del programa a nivel de procesos, comunicación e interacción**. En esta fase se analiza y se obtiene el diseño de la aplicación, sin implementación de la interfaz.

- **Desarrollo e implementación del programa**. Fase de programación siguiendo las pautas del análisis y el diseño.

- **Añadir tablero y multimedia**. Desarrollo del tablero de juego y programación mediante librerías Java para la inclusión de gráficos y su manejo.

- **Redacción de la memoria**. Redacción de este documento, reflejando todas las fases de análisis, diseño, implementación y pruebas relativo a la aplicación.

- **Redacción de la presentación**. Diseñar una presentación con los aspectos mas importantes y decisivos del diseño de la aplicación.

## 5.5 Motivación

La motivación en cuanto al desarrollo de un juego distribuido utilizando herramientas criptográficas se debe a lo atrayente del tema de videojuegos y a la vez al uso de la seguridad en dicho campo. Muchos son los juegos que podemos descargar de la red, pero muy pocos nos garantizan la seguridad de comunicación.

La seguridad y los métodos utilizados para garantizarla también son un foco de motivación por parte del autor de este proyecto, la criptografía siempre se ha envuelto de un misterio que ha atraído a muchas personas a lo largo de la historia.

Por otra parte el lanzamiento de los dados utilizando algoritmos de compromiso de BIT, es una materia poco explicada a través de medios de comunicación como Internet. Estos métodos son los utilizados por la mayoría de casinos on-line y juegos de azar jugados por ordenador, sin embargo cuesta mucho trabajo encontrar información al respecto.

## 5.6 Producto obtenido

El producto obtenido de este trabajo final de carrera consiste en una aplicación Java que reproduce el juego de la Oca. Desde la línea de comandos recibe varios parámetros que permiten comenzar una partida y esperar conexiones de jugadores o bien permiten “apuntarse” a una partida ya comenzada por otro usuario. También se pueden escoger las claves privadas y publicas que se utilizarán para la autenticación, es decir, para asegurar la conexión y la comunicación. Una vez la aplicación esta en marcha, se ha facilitado su interacción con el usuario ya que solamente se puede lanzar el dado cuando se tiene el turno y salir del juego. Durante el desarrollo de la partida, se pueden ver mensajes de estado de la partida, como por ejemplo, quien esta jugando, el resultado del dado o las bonificaciones (saltar de oca en oca o de puente a puente) y penalizaciones (caer en el pozo o en la prisión).

***NOTA: para la correcta ejecución de este proyecto, se recomienda el uso de la versión de Java 1.4 o superior, debido a la distribución de librerías criptográficas que estas versiones contienen (JSSE). El uso de otras versiones anteriores no garantiza que esta aplicación funcione correctamente.***

## 5.7 Estructura de la memoria

La estructura de esta memoria se compone de varios capítulos. El primero se dedica a comentar los productos utilizados y su estado a día de hoy. Seguidamente se citan las herramientas utilizadas para desarrollar la aplicación. Los capítulos siguientes constituyen el cuerpo de la memoria y se estructuran en análisis, diseño e implementación de la aplicación. Después se describen las pruebas realizadas, las conclusiones al desarrollo y finalmente los anexos, con información de recursos necesarios a la aplicación y el código fuente comentado.

# 6 Estado del arte

## 6.1 Los sockets

### 6.1.1 Un poco de historia

El sistema de entrada/salida de Unix persigue un paradigma usualmente conocido como *abrir-leer-escribir-cerrar*. Antes de que cualquier usuario pueda ejecutar operaciones E/S, debe utilizar la función *abrir* para especificar y obtener permisos del fichero o dispositivo a utilizar. Una vez el objeto a sido abierto, el proceso de usuario realiza una o varias llamadas a las funciones de *leer* o *escribir* datos. *Leer,* lee datos del objeto y lo transfiere al proceso de usuario, mientras que *escribir* transfiere información desde el proceso de usuario al objeto abierto. Después de que todas las operaciones hayan sido finalizadas, el usuario llama a la función *cerrar* para informar al sistema operativo que ha finalizado la utilización de dicho objeto.

Cuando los servicios para la comunicación ínter proceso (IPC) y de red se añadieron a Unix, la idea era realizar la interfaz lo mas parecida a la utilizada para el sistema de entrada/salida. En Unix, un proceso tiene una serie de descriptores de E/S de los cuales uno lee y el otro escribe. Esos descriptores se pueden referir a ficheros, dispositivos, o canales de comunicación (sockets). El tiempo de vida de un descriptor esta basado en tres fases: creación (abrir socket), leer o escribir (recibir o enviar al socket), y destrucción (cerrar socket).

La interfaz de la IPC, en versiones de Unix BSD esta implementada como una capa por encima de los protocolos de red TCP y UDP. Los destinos de los mensajes están especificados como direcciones de sockets; cada dirección de socket es un identificador de comunicaciones que consiste en un numero de puerto y una dirección de Internet (dirección IP).

Las operaciones de la IPC están basadas en pares de sockets, cada uno perteneciente a un proceso de comunicación. La IPC se hace intercambiando alguna información transmitiendo esos datos en un mensaje entre un socket en un proceso y otro socket en otro proceso. Cuando los mensajes se han enviado, el mensaje se pone en la cola en el socket emisor hasta que el subyacente protocolo de red lo ha transmitido. Cuando llega a destino, el mensaje se pone en cola en el socket receptor hasta que el proceso que recibe la información utiliza los mecanismos necesarios para recuperarla.

### 6.1.2 Comunicaciones TCP y UDP.

Existen dos protocolos de comunicaciones que pueden ser utilizados a la hora de programar sockets: comunicación mediante datagramas y mediante flujo de comunicación.

- **Comunicación con datagramas:** El protocolo de comunicación mediante datagramas conocido como UDP (User Datagram Protocol), es un protocolo no orientado a conexión, es decir, cada vez que se envían datagramas, se necesita enviar tanto el descriptor del socket local como la dirección del socket receptor, la información adicional se ha de enviar cada vez que los procesos se quieren comunicar.

- **Flujo de comunicación:** Conocido como TCP (Transfer Control Protocol). Al contrario que UDP, TCP es un protocolo orientado a la conexión. A fin de establecer una comunicación sobre el protocolo TCP, se debe primero establecer una conexión entre el par de sockets. Mientras que uno de ellos escucha una posible petición de conexión (servidor), el otro realiza una petición de conexión (cliente). Una vez los sockets se han conectado, se pueden utilizar para transmitir datos en una u otra dirección.

Mas adelante se verán la implementación de los sockets utilizando la librería de sockets seguros con Java (JSSE).

## 6.2 EL protocolo SSL

### 6.2.1 Introducción

Inicialmente, Netscape Comunications desarrolló el protocolo SecureSocket Layer (SSL) con el objetivo de añadir privacidad y autenticidad a las comunicaciones basadas en un protocolo fiable, como por ejemplo, TCP.

El diseño del protocolo SSL se hizo de manera suficientemente general como para que se pudiera utilizar en cualquier aplicación y no solamente con HTTP. Así, un protocolo de nivel superior que utilice TCP también puede utilizar SSL.

De esta manera, para hacer segura una aplicación solamente haría falta sustituir las llamadas de *sockets* originales por las de la librería SSL, como ilustra la figura siguiente. De aquí proviene el nombre del protocolo.

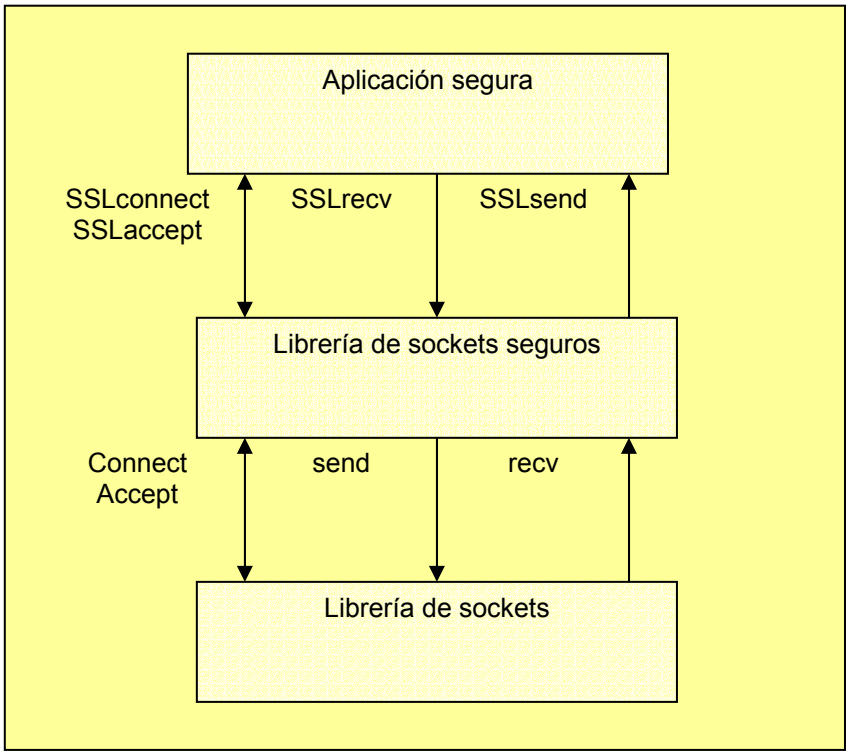


*Figura 3. Llamadas a la librería SSL*

La primera versión del protocolo SSL fue la versión 2.0, desarrollada a mediados de los años noventa. Poco después, Netscape publicó la versión 3.0, con muchos cambios respecto a la anterior.

### 6.2.2 Servicios proporcionados

#### *6.2.2.1 Confidencialidad*

El flujo de información normal en una conexión SSL consiste en intercambiar paquetes con datos encriptados mediante claves simétricas (por motivos de eficiencia y rapidez). Al inicio de cada sesión, cliente y servidor se ponen de acuerdo en las claves a utilizar para cifrar los datos. Siempre se utilizan dos claves diferentes, una para los paquetes enviados desde el cliente al servidor, y la otra para los paquetes enviados en sentido contrario.

Para evitar que cualquier intruso pueda interceptar las claves de sesión enviadas al principio de la negociación, se sigue un método de intercambio de claves:

- En SSL v2, existe un único método definido: cada servidor tiene que tener asociado un certificado X.509, que se envía al cliente al inicio de la sesión, y el cliente lo utiliza para enviar una cadena de bits llamada **secreto maestro**, cifrados con la clave publica del servidor. A partir del secreto maestro se obtienen las claves que se utilizarán en la comunicación cifrada.

- EN SSL v3, en cambio, el método de intercambio se negocia durante el establecimiento de la conexión.

#### *6.2.2.2 Autenticidad*

Como acabamos de apuntar, en SSL v2 el servidor envía su certificado al cliente y éste lo puede utilizar para el intercambio inicial de claves y también para verificar que se ha conectado al servidor autentico. El proceso de verificación de autenticidad consta de dos partes:

- El cliente valida el certificado comprobando el nombre, la autoridad que lo ha firmado, la caducidad, la firma, que no este revocado, etc.

- El protocolo proporciona un método de desafío-respuesta para que el servidor demuestre que tiene la clave privada correspondiente y, por lo tanto, el certificado realmente es suyo. Lo primero que tiene que hacer el cliente cuando comienza una nueva sesión es enviar al servidor una cadena de bits aleatorios o *desafío*, y una vez se han establecido las claves de cifrado el servidor en primer lugar ha de enviar el desafío encriptado con la clave acordada. Si el servidor hubiera enviado un certificado que no era el suyo, no sería capaz de obtener la clave necesaria para encriptar el desafío, tal y como el cliente espera.

El protocolo SSL v2 también contempla la autenticación del cliente al servidor (en el proyecto no se ha utilizado ya que la autenticación de la parte servidor es suficiente y de esta forma la aplicación aumenta en velocidad de ejecución).

#### *6.2.2.3 Integridad*

Una vez establecidas las claves a utilizar en la sesión, además de encriptar los paquetes se les incorpora un código de autenticación (MAC) que permite verificar la integridad de la información. Para calcular este código se utilizan otras claves secretas derivadas del secreto maestro.

A parte de los servicios vistos, el protocolo proporciona herramientas para aumentar la eficiencia en la comunicación, permitiendo la reutilización de los parámetros de una primera sesión para otras posteriores.

### 6.2.3 Descripción del protocolo de negociación.

El protocolo tiene la finalidad de autenticar al cliente y/o al servidor, y acordar los algoritmos y claves que utilizaran de una manera segura, es decir, de manera que se garantiza la confidencialidad y la integridad de la negociación.

1. Cuando de establece una conexión, el servidor normalmente espera que el cliente inicie la negociación. También puede optar por enviar un mensaje de petición de saludo. El mensaje de petición de saludo indica al cliente que esta preparado para comenzar. Durante la sesión, el servidor también puede enviar un mensaje de petición de saludo al cliente para indicar que quiere iniciar una renegociación.

2. El cliente envía un mensaje de saludo de cliente al inicio de la conexión o como respuesta a una petición de saludo.

3. A continuación, el servidor responde con un mensaje de saludo de servidor. Si se ha decidido continuar con una sesión anterior, cliente y servidor ya pueden comenzar a utilizar los algoritmos y claves previamente acordados, se saltan los mensajes que vienen a continuación, y pasan directamente a los de finalización de la negociación.

4. Si el cliente puede autenticar al servidor, este envía el mensaje de certificado. El mensaje de certificado normalmente contendrá el certificado X.509 del servidor o una cadena de certificados. Si el servidor no tiene certificado, o se ha acordado un método de intercambio de claves que no utiliza certificados, tiene que enviar un mensaje de intercambio de claves de servidor. El mensaje de intercambio de claves de servidor contiene los parámetros necesarios para el método que se tiene que seguir.

5. En el caso que se tenga que autenticar también al cliente, el servidor le envía un mensaje de petición de certificado, que contiene una lista de los tipos posibles de certificados que el servidor puede admitir, por orden de preferencia, y una lista de los DN de las autoridades de certificación que el servidor reconoce.

6. Para finalizar la primera fase del dialogo, el servidor envía un mensaje de ***fin de saludo de servidor***.

7. Cuando acaba el saludo del servidor, el cliente ya sabe como tiene que continuar el protocolo de negociación. En primer lugar, si el servidor le ha pedido un certificado, el cliente le envía un mensaje de certificado si tiene alguno de las características solicitadas.

8. El cliente envía un mensaje de intercambio de claves de cliente. El mensaje de intercambio de claves de cliente tiene un contenido que depende del método de intercambio acordado. EL caso de seguir el método RSA, en este mensaje esta el resultado de encriptar una cadena de 48 bytes aleatorios con la clave publica del servidor. Esta cadena será utilizada como secreto premaestro. En este punto, cliente y servidor ya pueden calcular el secreto maestro. Este secreto es una cadena de 48 bytes que se obtiene aplicando funciones de resumen varias veces y de diversas maneras a una combinación del secreto premaestro y las cadenas aleatorias que enviaron cliente y servidor en sus mensajes de saludo.

9. El cliente ya se puede autenticar si ha enviado un certificado como respuesta a una petición de certificado. Para esa autenticación tiene que demostrar que tiene la clave privada correspondiente mediante un mensaje de verificación de certificado, que contiene una firma, generada con la clave privada del cliente, una cadena de bytes obtenida a partir de la concatenación de todos los mensajes de negociación intercambiados hasta ahora, desde el saludo del cliente hasta el intercambio de claves de cliente.

10. Cuando se llega a este punto ya se pueden enviar los mensajes con las claves y los algoritmos de cifrado, MAC y compresión antes acordados. Cada parte envía al otro una notificación de cambio de cifrado seguida de un mensaje de finalización.

### 6.2.4 Análisis de seguridad.

Con el uso del protocolo se pueden evitar los ataques siguientes:

- **Escucha de los paquetes enviados por cliente y servidor**: Cuando los datos de envían encriptados, un intruso que pueda leer los paquetes se enfrenta al problema de romper el cifrado si quiere interpretar el contenido.

- **Suplantación de cliente o servidor**: Cuando se hace la autenticación de servidor o cliente, el certificado debidamente firmado sirve como prueba de identidad.

- **Alteración de los paquetes**: Un atacante puede modificar los paquetes para lleguen con un contenido diferente al original. El receptor detectara el ataque porque el código de integridad MAC será incorrecto.

## 6.3 El estándar de cifrado de bloque DES.

### 6.3.1 Introducción

En el año 1977, el National Bureau of Standards (NBS), una sección del Departamento de Defensa de los EUA, publicó un criptosistema estándar creado con la finalidad de proteger cualquier tipo de datos: el DES. En su desarrollo participo la empresa IBM y la National Security Agency (NSA). El criptosistema DES es un criptosistemas de cifrado de bloque que cifra bloques de datos de 64 bits de longitud por medio de una clave de 56 bits. El DES consigue cumplir tanto el principio de confusión (inclusión de sustituciones para que la relación entre clave y texto cifrado sea tan complicada como se pueda) y el de difusión (utiliza transformaciones que disipan las propiedades estadísticas del texto en claro entre el texto cifrado), gracias a las acciones de las cajas S que contiene.

### 6.3.2 Funcionamiento

Como se puede observar en el gráfico de funcionamiento, se suministra al algoritmo un bloque de entrada, M, sobre el que se aplica una permutación inicial, σ, de donde se obtiene $T_o$ = σ(M). Cuando ya se han hecho las 16 iteraciones de la función f, el resultado se transpone por medio de la permutación de salida $\sigma^{-1}$. Las permutaciones σ y $\sigma^{-1}$ se recogen en dos tablas que cambian los bits de entrada desde una posición inicial hasta una final:

| Tabla de permutaciones σ | | | | | | | |
|---|---|---|---|---|---|---|---|
| 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 |
| 60 | 52 | 44 | 36 | 28 | 20 | 12 | 4 |
| 62 | 54 | 46 | 38 | 30 | 22 | 14 | 6 |
| 64 | 56 | 48 | 40 | 32 | 24 | 16 | 8 |
| 57 | 49 | 41 | 33 | 25 | 17 | 9 | 1 |
| 59 | 51 | 43 | 35 | 27 | 19 | 11 | 3 |
| 61 | 53 | 45 | 37 | 29 | 21 | 13 | 5 |
| 63 | 55 | 47 | 39 | 31 | 23 | 15 | 7 |

*Tabla 2. Tabla de permutaciones σ:*

| Tabla de permutaciones $\sigma^{-1}$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| 40 | 8 | 48 | 16 | 56 | 24 | 64 | 32 |
| 39 | 7 | 47 | 15 | 55 | 23 | 63 | 31 |
| 38 | 6 | 46 | 14 | 54 | 22 | 62 | 30 |
| 37 | 5 | 45 | 13 | 53 | 21 | 61 | 29 |
| 36 | 4 | 44 | 12 | 52 | 20 | 60 | 28 |
| 35 | 3 | 43 | 11 | 51 | 19 | 59 | 27 |
| 34 | 2 | 42 | 10 | 50 | 18 | 58 | 26 |
| 33 | 1 | 41 | 9 | 49 | 17 | 57 | 25 |

*Tabla 3. Tabla de permutaciones $\sigma^{-1}$:*

Esquema general de funcionamiento del DES:

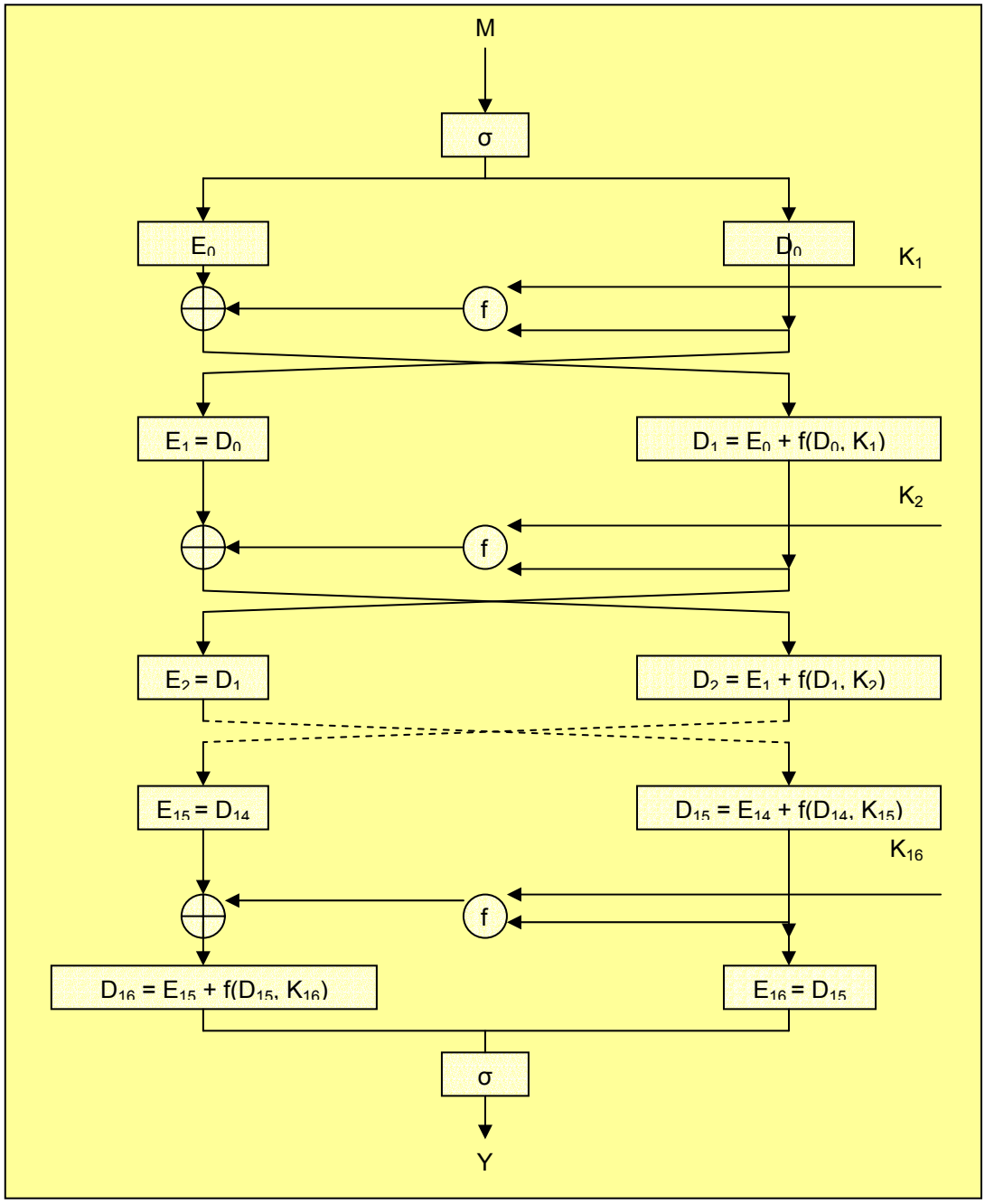


*Figura 4. Esquema DES*

Las funciones combinan sustituciones y transposiciones, y las claves con las que opera, 16 claves de 48 bits extraídas de una clave de 56 bits, se generan a partir del uso de tablas de permutaciones que generan bits de salida permutando y haciendo combinaciones a partir de los bits de entrada.

## 6.4 Situaciones de desconfianza mutua. El compromiso de BIT.

A y B se telefonean para ir al cine, pero no se ponen de acuerdo en la película que quieren ver. A ha escogido una película y B otra. Finalmente deciden escoger la película que verán, a cara o cruz. El problema es que como hablan por teléfono, obviamente, es muy fácil que B, que tira la moneda, haga trampa, ya que A no puede ver que resultado ha salido realmente en la moneda y B puede decir siempre que ha salido lo contrario.

Para solucionar este problema, lo que tiene que hacer A es comprometerse a un valor (cara o cruz) y enviar a B el compromiso del valor al cual se ha comprometido de manera que B no pueda ver cual es ese valor. B lanza una moneda y comunica el resultado a A. Entonces A abre su compromiso y comprueba si ha acertado o no. Evidentemente, para que A tampoco pueda hacer trampa, tendrá que asegurar que no ha podido cambiar el valor una vez se ha comprometido.

Traduciendo este esquema en un problema matemático y suponiendo que A se compromete al valor de un solo BIT, tenemos lo que se llama ***compromiso de BIT***.

Las características de este compromiso son las siguientes:

- B, a partir del compromiso de A, no puede obtener el valor b.
- A puede abrir el compromiso para mostrar que efectivamente b era el valor al que se había comprometido.
- A no puede abrir el mismo compromiso para mostrar otro valor que no sea b.

Otro ejemplo, mas directamente relacionado con el lanzamiento de dados, o lanzamiento de monedas es el siguiente: Alice y Bob cada viernes por la noche juegan al juego de la oca, pero cuando Bob tiene que ausentarse durante un tiempo, acuerdan que se llamaran por teléfono para seguir jugando. Pero se les presenta algún problema. ¿Como, exactamente, se lanza un dado por teléfono, para que los dos sepan que el resultado del lanzamiento del dado a sido legal?.

Lo que han de hacer Alice y Bob es leer a M. Blum en su articulo “Coin Flipping By Telephone: A protocol for solving impossible problems” publicado en 1982, presentando un protocolo simple para el lanzamiento de una moneda.

Lo que el articulo trata es simple: Alice lanzara una moneda y le dirá a Bob el resultado, entonces Bob lanzara otra moneda y sumara (XOR) los dos resultados anunciando el resultado. Parece un protocolo razonable ya que solo con que uno de los dos lanzamientos fuera verdadero, el resultado obtenido sería también cierto. Pero los problemas vienen cuando los jugadores son adversarios, ya que cuando Bob lanza su moneda, sabe el resultado que ha obtenido Alice y por lo tanto, puede “trucar” su resultado anunciando un resultado favorable a sus intereses.

Joe Kilian, en su articulo “Uses of Randomness in Algorithms and Protocols”, de MIT press en 1990, va mas lejos hablando del uso de la criptografía para asegurar el lanzamiento aleatorio de una moneda.

En el nuevo protocolo, Alice y Bob determinan un mensaje como por ejemplo “Mi moneda ha resultado”. Entonces Alice lanza su moneda y añade al mensaje anterior su resultado, dando el mensaje “Mi moneda ha resultado cara”. Llamemos a ese mensaje A, acto seguido genera una clave aleatoria y calcula B encriptando A utilizando la clave generada, y transmite B a Bob. Bob hace lo mismo, enviando el mensaje cifrado con su clave hacia Alice.

Cuando Alice recibe el mensaje cifrado de Bob, envía su clave para que Bob pueda descifrar el mensaje y pueda ver que el lanzamiento de Alice resulto cara. Bob conoce el resultado de Alice, pero no puede variar su clave para que Alice descubra un valor que le favorezca, y por lo tanto tendrá que enviarle la clave que genero al principio, con lo cual Alice sabrá el resultado

que obtuvo Bob. Finalmente los dos suman los resultados (XOR) y calculan el lanzamiento final.

Enfocado al lanzamiento de los dados, que tienen seis posibles resultados, utilizando el cifrado de bloque DES, el funcionamiento es el siguiente, poniendo como ejemplo a dos jugadores, A y B, que se puede extender fácilmente a n jugadores:

- A y B generan un valor aleatorio que será su compromiso (el valor del dado lanzado individualmente).
- A y B generan una clave de sesión mediante el algoritmo de bloque DES. $C_a$ y $C_b$, y cifran el resultado del primer apartado.
- A y B envían el valor cifrado a todos los demás. A le envía a B el valor cifrado y B le envía a A su valor, también cifrado.
- Cuando todos los jugadores tienen el valor cifrado de los correspondientes compromisos, envían las respectivas claves utilizadas para cifrar ese valor a todos los jugadores. A le envía la clave a B y B se la envía a A.
- Cuando lo dos jugadores tienen las claves del otro, descifran los valores cifrados recibidos.
- Finalmente, suman todos los valores descifrados incluido el suyo, y generan el valor calculado colectivamente haciendo un modulo 6.

De esa forma se calcula el valor del dado lanzado colectivamente.

# 7 Herramientas utilizadas

## 7.1 El lenguaje de programación. JAVA.

Es preciso comentar en este apartado, aunque en la introducción ya lo hicimos, que para la correcta ejecución de este proyecto, se recomienda el uso de la versión de Java 1.4 o superior (Java2), debido al uso por parte de la aplicación de librerías criptográficas (JSSE) que esta distribución contiene.

### 7.1.1 Introducción

Java es un lenguaje desarrollado por Sun con la intención de competir con Microsoft en el mercado de la red. Sin embargo, su historia se remonta a la creación de una filial de Sun (FirstPerson) enfocada al desarrollo de aplicaciones para electrodomésticos, microondas, lavaplatos, televisiones... Esta filial desapareció tras un par de éxitos de laboratorio y ningún desarrollo comercial.

Sin embargo, para el desarrollo en el laboratorio, uno de los trabajadores de FirstPerson, James Gosling, desarrolló un lenguaje derivado de C++ que intentaba eliminar las deficiencias del mismo. Llamó a ese lenguaje Oak. Cuando Sun abandonó el proyecto de FirstPerson, se encontró con este lenguaje y, tras varias modificaciones (entre ellas la del nombre), decidió lanzarlo al mercado en verano de 1995.

El éxito de Java reside en varias de sus características. Java es un lenguaje sencillo, o todo lo sencillo que puede ser un lenguaje orientado a objetos, eliminando la mayor parte de los problemas de C++, que aportó su granito de arena a los problemas de C. Es un lenguaje independiente de plataforma, por lo que un programa hecho en Java se ejecutará igual en un PC con Windows que en una estación de trabajo basada en Unix. También hay que destacar su seguridad, desarrollar programas que accedan ilegalmente a la memoria o realizar caballos de Troya es una tarea propia de titanes.

Cabe mencionar también su capacidad multihilo, su robustez o lo integrado que tiene el protocolo TCP/IP, lo que lo hace un lenguaje ideal para Internet. Pero es su sencillez, portabilidad y seguridad lo que le han hecho un lenguaje de tanta importancia.

### 7.1.2 Portabilidad

La portabilidad se consigue haciendo de Java un lenguaje medio interpretado medio compilado. ¿Cómo se come esto? Pues se coge el código fuente, se compila a un lenguaje intermedio cercano al lenguaje máquina pero independiente del ordenador y el sistema operativo en que se ejecuta (llamado en el mundo Java bytecode) y, finalmente, se interpreta ese lenguaje intermedio por medio de un programa denominado máquina virtual de Java.

Este esquema lo han seguido otros lenguajes, como por ejemplo Visual Basic. Sin embargo, nunca se había empleado como punto de partida a un lenguaje multiplataforma ni se había hecho de manera tan eficiente. Cuando Java apareció en el mercado se hablaba de que era entre 10 y 30 veces más lento que C++. Ahora, con los compiladores JIT (Just in Time) se habla de tiempos entre 2 y 5 veces más lentos. Con la potencia de las máquinas actuales, esa lentitud es un precio que se puede pagar sin problemas contemplando las ventajas de un lenguaje portable.

### 7.1.3 Orientación a objetos

Dado que Java es un lenguaje orientado a objetos, es imprescindible entender qué es esto y en qué afecta a nuestros programas. Desde el principio, la carrera por crear lenguajes de programación ha sido una carrera para intentar realizar abstracciones sobre la máquina. Al principio no eran grandes abstracciones y el concepto de lenguajes imperativos es prueba de ello. Exigen pensar en términos del ordenador y no en términos del problema a solucionar. Esto provoca que los programas sean difíciles de crear y mantener, al no tener una relación obvia con el problema que representan: No abstraen lo suficiente.

Muchos paradigmas de programación intentaron resolver este problema alterando la visión del mundo y adaptándola al lenguaje. Estas aproximaciones modelaban el mundo como un conjunto de objetos o de listas. Funcionaban bien para algunos problemas pero no para otros.
Los lenguajes orientados a objetos, más generales, permiten realizar soluciones que, leídas, describen el problema. Permiten escribir soluciones pensando en el problema y no en el ordenador que debe solucionarlo en último extremo. Se basan en cinco características:

- **Todo es un objeto**. Cada elemento del problema debe ser modelizado como un objeto.

- **Un programa es un conjunto de objetos diciéndose entre sí que deben hacer por medio de mensajes**. Cuando necesitas que un objeto haga algo le mandas un mensaje. Más concretamente, ejecutas un método de dicho objeto.

- **Cada objeto tiene su propia memoria, que llena con otros objetos**. Cada objeto puede contener otros objetos. De este modo se puede incrementar la complejidad del programa, pero detrás de dicha complejidad sigue habiendo simples objetos.

- **Todo objeto tiene un tipo**. En jerga *POO*, cada objeto es una instancia (un caso particular) de una clase (el tipo general). Lo que distingue a una clase de otra es la respuesta a la pregunta, ¿qué mensajes puedes recibir?

- **Todos los objetos de un determinado tipo pueden recibir los mismos mensajes**. Por ejemplo, dado que un objeto de tipo Gato es también un objeto de tipo Animal, se puede hacer código pensando en los mensajes que se mandan a un animal y aplicarlo a todos los objetos de ese tipo, sin pensar si son también gatos o no.

Eso de que la programación orientada a objetos intente establecer una relación biunívoca entre el espacio del problema y el de la solución está muy bien pero... ¿cómo hacemos que un objeto realice cosas útiles y necesarias? Por medio de su interfaz, interfaz que está definida en la clase del objeto. Por ejemplo, supongamos que tenemos la clase *Interruptor* cuyo interfaz define los métodos *apagar()* y *encender()*. Entonces hacemos lo siguiente:

```
Interfaz de un objeto
Interruptor i = new Interruptor();
i.encender();
```

Primero creamos una referencia llamada *i* de tipo *Interruptor*, y luego creamos un objeto, una instancia de dicho tipo, por medio de la palabra reservada *new*. Asignamos ese objeto a la referencia por medio del signo =. Por último, enviamos un mensaje al objeto recién creado y asignado poniendo el nombre de la referencia, un punto, y el nombre del mensaje (método).

No es objetivo de esta memoria el describir exhaustivamente el lenguaje de programación Java, ya que de lo contrario se convertiría en un manual del programador.

## 7.2 Las comunicaciones y la seguridad, todo en uno. La arquitectura JCA de Java.

### 7.2.1 La arquitectura criptográfica de Java.

La capacidad de la plataforma de seguridad y encriptación de Java ha crecido tremendamente durante los últimos años. El JDK (Java Development Kit) versión 1.4 (también conocido como Merlín), viene ahora con muchos paquetes (*packages*) relativos a la seguridad, incluyendo el Java Cryptography Extension (JCE), el Java Secure Socket Layer Extension (JSSE), y el Java Authentication and Authorization Service (JAAS). Cada uno de estos componentes son piezas de la arquitectura criptográfica de Java (JCA) como se ilustra en la figura de abajo:

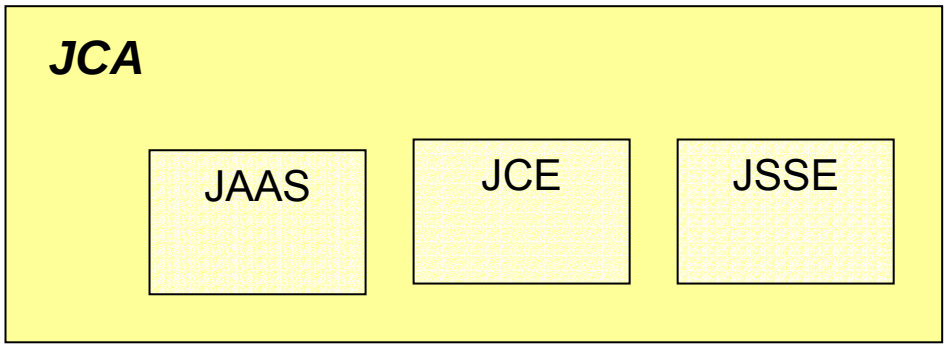


*Figura 5. Esquema de la arquitectura criptográfica de Java (JCA)*

#### *7.2.1.1 JSSE*

Una de las propiedades más importantes de JCA es que no alberga ningún algoritmo particular de encriptación. Cada uno de los conocidos algoritmos de encriptación tiene sus ventajas y desventajas y nuevos algoritmos se están implementando cada día. El JCA permite la incorporación de nuevos algoritmos una vez desarrollados. Utiliza el concepto de *cryptographic service provider* (CSP) que es algo parecido a un *plug-in* de seguridad. Un CSP incorpora la implementación de un algoritmo particular. JDK 1.4 viene con algún CSP, incluyendo el SunJSSE, que provee muchos algoritmos estándar los cuales son suficientes para la mayoría de aplicaciones.

JSSE provee comunicación segura con sockets para la plataforma de Java 2. Implementa el Secure Socket Layer (SSL) y el Transport Layer Security (TLS), dos estandarizados protocolos para implementar comunicaciones seguras a través de Internet. Ambos SSL y TLS recaen en la criptografía de clave publica que se describe a continuación.

#### *7.2.1.2 criptografía de clave pública*

Un problema con muchos algoritmos criptográficos es que requieren la distribución de claves compartidas. Un password es un buen ejemplo de clave compartida. El problema es que se han de compartir entre entidades antes de que la comunicación segura pueda comenzar. El proceso de compartición de claves puede ser vulnerable a ataques externos que nos lleva a un problema del tipo ¿quién fue primero, la gallina o el huevo?: ¡antes de que podamos intercambiar información de forma segura, tenemos que intercambiar las claves de forma segura!.

Este problema se solucionó en 1976 gracias a Whitfield Diffie y Martin Hellman, con la creación de la criptografía de clave publica. En el sistema de Diffie-Hellman cada uno de los comunicantes tiene un par de claves, una privada y otra publica. La clave privada solamente la conoce el comunicante, mientras que la publica se puede dar a cualquiera. La información cifrada con una de las claves solamente puede ser descifrada con la otra.

Así, si queremos crear un mensaje para ser leído solo por una persona en particular, se utiliza su clave publica y de esta forma solamente esa persona con su clave privada puede descifrarlo. Por otra parte, si queremos encriptar utilizando la propia clave privada, todo el mundo que conozca nuestra clave publica podrá descifrarlo. De esta forma se garantiza que hemos sido nosotros quien ha enviado el mensaje ya que solamente nosotros conocemos nuestra clave privada. Este tipo de mensajes son conocidos como *firmas*.

#### *7.2.1.3 Certificados y autoridad de certificación*

Un certificado es una clave publica que ha sido digitalmente firmada por una parte de confianza para probar que esa es una clave válida. Esta parte de confianza se llama autoridad de certificación, *certification authority* (CA). En un sentido, la CA provee testimonio de que la clave publica pertenece realmente a la persona que la posee. Dependiendo del grado de autoridad que deseemos, podemos utilizar desde autoridades comerciales a crear una nosotros. Si una entidad firma su propia clave pública se conoce como un certificado auto-firmado (*self-signed certificate*) que es lo que utilizamos en este proyecto.

#### *7.2.1.4 SSL y TLS*

Como hemos mencionado antes, el marco de trabajo de JSSE junto con el proveedor SunJSSE, implementan los protocolos SSL y TLS.

SSL utiliza criptografía de clave publica para intercambiar un conjunto de claves compartidas y utilizar una encriptación basada en claves compartidas estándar para intercambiar la información. Dichas claves intercambiadas se utilizan tanto para encriptar (haciendo indescifrable la información) como para autentificar la información (asegurando que no ha venido de ningún impostor).

#### *7.2.1.5 El protocolo de negociación de SSL*

Antes de que la información pueda ser enviada a través de una conexión SSL, las dos partes deben negociar e intercambiar la información de la clave. Este proceso es conocido como *protocolo de negociación*. No entraremos en detalle sobre este protocolo ya que esta explicado en el apartado "El protocolo SSL" , pero si veremos como se utiliza orientado a JSSE. Para nuestros propósitos necesitamos saber que el protocolo realiza los siguientes pasos:

- El servidor envía su certificado al cliente y el cliente lo verifica.
- El cliente envía su certificado al servidor y el servidor lo verifica.
- El cliente cifra la información del password con la clave publica del servidor y se la envía. El password es utilizado para crear por ambas partes de la comunicación claves secretas idénticas, las cuales utilizarán para transmitir los datos.

La autentificación del cliente es opcional: el servidor puede requerir el certificado del cliente, pero no es obligatorio. En el proyecto **no se ha utilizado la autentificación del cliente** para hacer la conexión mas rápida, ya que la seguridad esta garantizada.

#### *7.2.1.6 Utilizando los sockets JSSE*

La configuración de la conexión debe seguir unos pasos:

**Para conectarse a una conexión:**

- Crear un *secureRandom*, una fuente de números seguros aleatorios. Dichos números son suficientemente aleatorios para hacerlos invulnerables a ataques externos.

- Crear un objeto *Keystore* que contenga la clave publica de servidor remoto junto con el certificado de clave publica firmado por la CA (en este caso auto-firmado), que se lee del fichero almacenpublico.jks

- Crear un objeto *keystore* que contenga el par de claves publica/privada del cliente. Se lee desde el fichero almacenprivado.jks
- Crear un objeto *TrustManagerFactory* desde el *keystore* del servidor remoto. Se utiliza para autentificar al servidor remoto.
- Crear un objeto *KeyManagerFactory* desde el *keystore* del cliente, que se utiliza para cifrar/descifrar.
- Crear un objeto *SSLContext*, usando el objeto *KeyManagerFactory*, el objeto *TrustManagerFactory* y el objeto *SecureRandom*.
- Utilizar el objeto *SSLContext* para crear un objeto *SSLSocketFactory.*
- Utilizar el objeto *SSLSocketFactory* para crear un *SSLSocket*, el cual actúa como un socket normal salvo por la seguridad que aporta.

**Para crear una conexión capaz de escuchar conexiones entrantes:**

- Crear un *secureRandom*, una fuente de números seguros aleatorios.
- Crear un objeto *Keystore* que contenga la clave publica de cliente y certificado, que se lee del fichero almacenpublico.jks
- Crear un objeto *keystore* que contenga el par de claves publica/privada del servidor. Se lee desde el fichero almacenprivado.jks
- Crear un objeto *TrustManagerFactory* desde el *keystore* del cliente. Se utiliza para autentificar al cliente (Si hace falta).
- Crear un objeto *KeyManagerFactory* desde el *keystore* del cliente, que se utiliza para cifrar/descifrar.
- Crear un objeto *SSLContext*, usando el objeto *KeyManagerFactory*, el objeto *TrustManagerFactory* y el objeto *SecureRandom*.
- Utilizar el objeto *SSLContext* para crear un objeto *SSLServerSocketFactory.*
- Utilizar el objeto *SSLServerSocketFactory* para crear un *SSLServerSocket*, el cual actúa como un *serverSocket* normal salvo por la seguridad que aporta.
- Llamar al método *accept()* del objeto *SSLServerSocket* con el fin de esperar conexiones entrantes seguras.

Es un poco complicado, pero el proceso siempre es el mismo y se utiliza una vez se inicializa la partida o se conecta a una creada. A partir de aquí las conexiones son seguras!.

Se puede ver su aplicación en el código fuente del programa, más concretamente en la fase de inicialización de una partida y de conexión a una existente.

## 7.3 La interfaz para JAVA. AWT.

Las librerías del lenguaje de programación Java proporcionan un paquete de ayuda para la creación de interfaces de usuario llamado kit abstracto de ventana, Abstract Windowing Toolkit (AWT).

### 7.3.1 ¿Que es una interfaz de usuario?

La interfaz de usuario es la parte de un programa que interactúa con el usuario del programa. Pueden ser de muchas formas y varían en complejidad desde simples interfaces de línea de comandos, hasta las mas sofisticadas interfaces basadas en iconos y la utilización del ratón.

Debido a que Java es un lenguaje de programación independiente de plataformas especificas, el AWT debe también serlo. El AWT fue diseñado para proveer una serie de herramientas para el diseño de interfaces de usuario graficas que funcionara en muchas plataformas. Los elementos de esa interfaz proveídos por el AWT están implementados utilizando las herramientas GUI que cada plataforma en particular dispone, preservando de esa forma el aspecto y el sentido de dichas plataformas. Este es uno de los puntos fuertes del AWT. La desventaja es que una interfaz diseñada bajo una plataforma en particular, puede variar si el programa se ejecuta en otra distinta.

### 7.3.2 Componentes y contenedores

Un interfaz grafico de usuario esta construido de elementos gráficos llamados componentes. Componentes típicos incluyen botones, barras de desplazamiento y áreas de texto. Los componentes permiten al usuario interactuar con la aplicación y proveen reacciones visuales que permiten saber al usuario el estado del programa. En el AWT, todos los componentes de la interfaz de usuario son instancias de la clase *Component* o una de sus *subclases*.

Los componentes no funcionan solos si no que se encuentran dentro de contenedores. Los contenedores contienen y controlan el formato o composición de los componentes. Los contenedores son en si mismos componentes, y pueden así estar dentro de otros contenedores. En el AWT, todos los contenedores son instancias de la clase *Container* o una de sus subclases.

Los componentes deben cuadrar completamente dentro del contenedor que lo contiene. Así se forma un árbol de elementos, comenzando con el contenedor en la raíz del árbol y expandiéndose por las ramas, que contienen componentes como por ejemplo, botones.

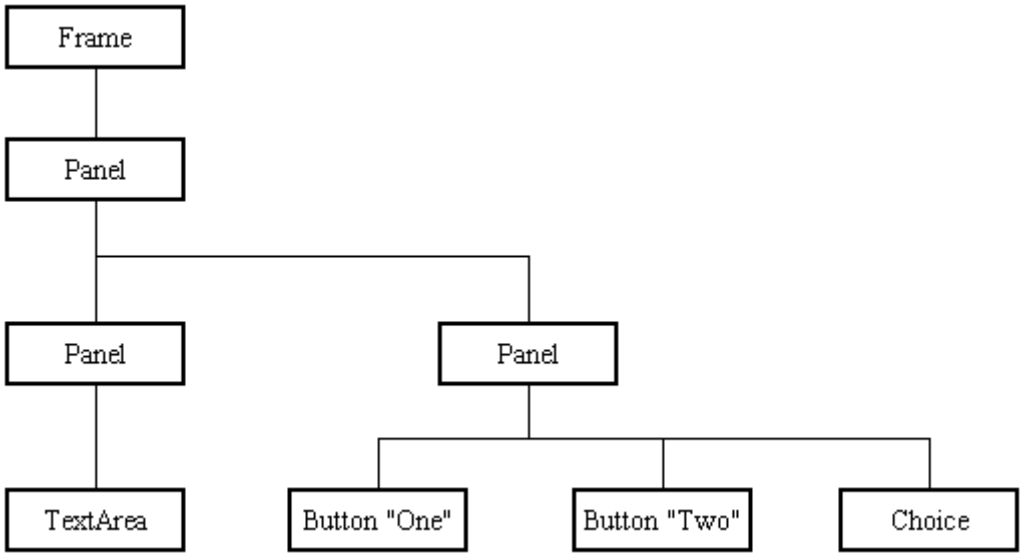


*Figura 6. árbol de componentes de la interfaz de usuario*

### 7.3.3 Tipos de componentes

La siguiente figura muestra la relación entre las clases de componentes del interfaz de usuario que ofrece el AWT. La clase *Component* define la interfaz a la que todos los componentes deben adherirse.

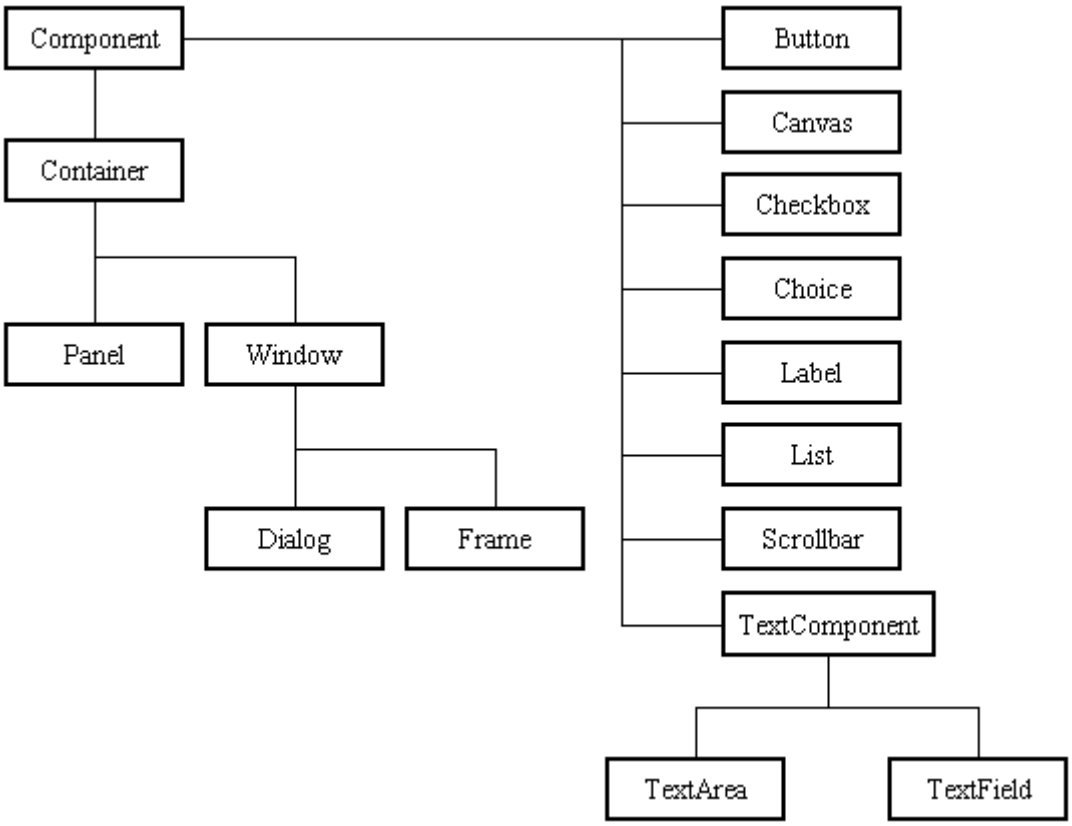


*Figura 7. Las relaciones de herencia de AWT*

### 7.3.4 Tipos de contenedores

El AWT proporciona cuatro clases de contenedores. Existe la clase *Window* y sus dos subclases (clase *Frame* y clase *Dialog*) así como la clase *Panel*.

### 7.3.5 Creando un contenedor

Antes de añadir los componentes que definen la interfaz, el programador debe crear un contenedor. Cuando se construye una aplicación, el programador debe crear primero una instancia de la clase *Window* o de la clase *Frame*. Cuando se construye un *Applet*, se define automáticamente un *frame*. Debido a que el *Applet* es una subclase de la clase *Panel*, el programador puede añadir componentes a la instancia de la clase *Applet*.

### 7.3.6 Las decisiones de comportamiento de los componentes

Para decidir como se comportan los componentes en términos de posiciones en pantalla, se utilizan los *layouts*. De esta forma se pueden alinear dentro del contenedor, y proporcionan, dependiendo del tipo de layout, una forma u otra de alineación. Para nuestro juego se ha utilizado mayoritariamente, el *BorderLayout*.

La clase *BorderLayout* tiene definidas cinco zonas, *North*, *East*, *West*, *South* y *Center*. Un solo componente se puede poner en una de estas cinco zonas, ocupando todo el espacio del contenedor.

# 8 Análisis de la aplicación.

## 8.1 Idea inicial. La figura del iniciador de partida.

Existen varias ideas que han de ser estudiadas antes de pasar al diseño propiamente dicho de la aplicación y que son importantes para el diseño de la aplicación en cuanto a la utilización de la arquitectura punto a punto. La idea inicial del diseño del juego se basa en la figura del iniciador de la partida. El iniciador de la partida es el encargado de, como su nombre indica, iniciar una nueva partida. Es el quien podrá decidir cuando se comienza a jugar una partida y al que se conectarán los demás jugadores no iniciadores de la partida. La necesidad de dicha figura esta justificada si pensamos en establecer un orden de juego en el que el iniciador tiene el primer turno de juego y los demás jugadores que vayan entrando en la partida obtienen su turno sucesivamente y por orden de llegada.

Pero también hay que tener en cuenta otros aspectos importantes a la hora de implementar el proyecto. Estamos diseñando una aplicación punto a punto, lo cual significa que cada una de las aplicaciones que manejan los diferentes jugadores tienen que poder comunicarse con todos los demás jugadores, con lo cual, cada proceso por separado se convierte en servidor (posibles conexiones de jugadores nuevos) y cliente (queremos conectarnos a una partida ya iniciada). ¿Cómo podemos hacer que n jugadores estén interconectados entre sí?. La figura del iniciador y del no iniciador de partida toman aquí un papel principal. Ser iniciador de partida representa controlar la partida nueva, esto es, esperar a que otros jugadores se conecten, pero ser no iniciador ¿qué representa?.

En un principio se no iniciador de partida representa que queremos conectarnos a una partida ya iniciada (o al menos en fase de espera de conexiones), y que una vez conectados tendremos que esperar también conexiones entrantes de jugadores que posteriormente se han enrolado a la partida, ya que si no, no sabríamos de su existencia y por lo tanto estaríamos ante la típica aplicación que utiliza la arquitectura cliente-servidor, un servidor que controla a todos los jugadores por separado y todos los jugadores “creyendo” que el servidor solamente les atiende a ellos.

Por lo tanto, la conexión de jugadores se tendrá que hacer mediante algún método o protocolo que permita la conexión de jugadores entre sí, como la arquitectura punto a punto especifica.

## 8.2 Arquitectura punto a punto. Paso de mensajes de conexión.

El protocolo de conexión de jugadores punto a punto se implementará a través de mensajes de conexión. Los mensajes que se pasarán los jugadores al conectarse a una partida iniciada son:

***Jugador nuevo***: un jugador quiere enrolarse a una partida y para ello se conecta con el iniciador de la partida. Éste le proporciona una lista de jugadores ya conectados para que el nuevo jugador se pueda conectar a su vez con los demás jugadores de la partida. Cuando el nuevo jugador tiene en su poder esta lista, se conecta con los demás jugadores mediante el siguiente mensaje.

***Jugador invitado***: un jugador ya enrolado a una partida se conecta con un jugador de la lista de jugadores que tiene en su poder. El jugador al cual se conecta le pasa sus datos pero no le proporciona ninguna lista de jugadores, ya que representa que esa lista se la ha proporcionado el iniciador de partida.

Como se observa, si un jugador es no iniciador de partida, después de conectarse con todos los jugadores enrolados en la partida, tiene que esperar también conexiones entrantes, para recibir a jugadores invitados por el iniciador de la partida.

De esta forma, se establece el protocolo necesario para conectar a n jugadores entre sí, para que todos dispongan de información de los demás jugadores de la partida y se puedan enviar información sin necesidad de la figura del servidor utilizada en arquitecturas cliente-servidor.

## 8.3 Fases de juego

Así pues, como primera idea, podríamos descomponer el diseño de la aplicación en diferentes fases.

### 8.3.1 Fase 1. Inicialización.

En una primera fase se creará la interfaz de juego, añadiendo los botones y las áreas de texto que servirán para poder seguir el estado del juego y se añadirán los gráficos de tablero de juego. Es en esta fase donde se crea la partida, que dependiendo de si se quiere iniciar una nueva o se quiere enrolar a una existente, la inicialización variará. Si se es el iniciador, se crea el jugador local, se le asigna el turno de juego 0. Si por el contrario queremos enrolarnos a una partida existente, en esta fase se inicializará el jugador local y se conectará a la partida existente. El turno de juego asignado será el numero de jugadores que hayan en la partida más uno.

### 8.3.2 Fase 2. Conexión.

En una segunda fase, y dependiendo de si se es iniciador o si se quiere conectar con una partida ya creada (en un principio lo sabemos si alguien nos dice a través de algún medio de comunicación su dirección IP y el puerto de conexión que ha asignado), se tendrán que hacer distintas operaciones. Es en esta fase donde se tendrá que inicializar el protocolo SSL, verificando claves privadas y certificados de claves publicas.

***Iniciador***. Inicializa un socket servidor que utilizará para recibir conexiones entrantes, y mientras no se pulse el botón de inicio de partida, tendrá que atender a posibles jugadores que quieran enrolarse a su partida iniciada.

***No iniciador***. Se conectará con el jugador indicado y una vez conectado con todos los jugadores de la partida iniciada, de la misma forma que en el caso anterior, esperará posibles conexiones entrantes de jugadores invitados a la partida.

La visión de los jugadores con respecto a los demás se puede esquematizar en el siguiente gráfico:

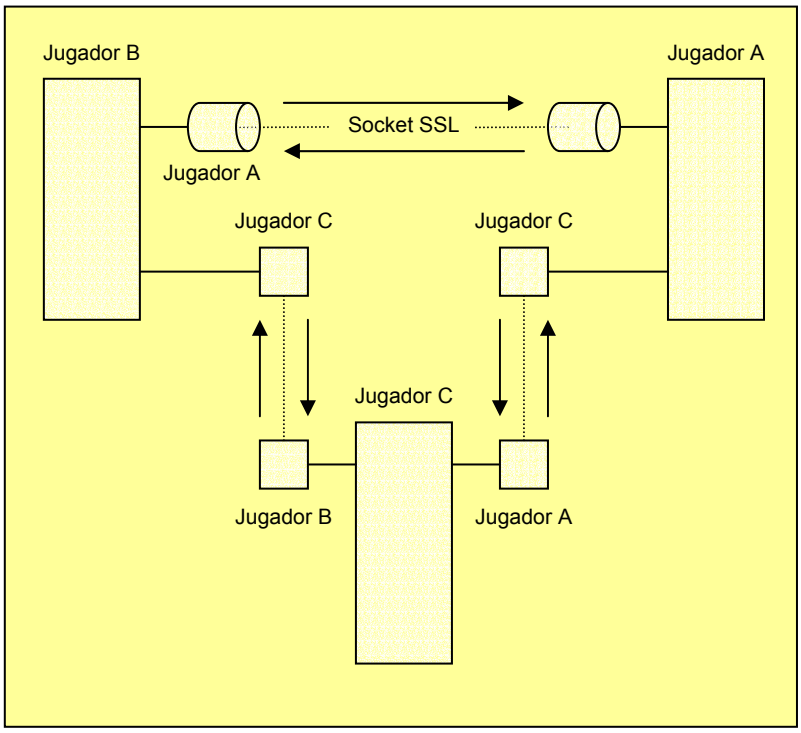


*Figura 9. Visión de los jugadores activos desde los diferentes procesos*

### 8.3.3 Fase 3. Juego

Mientras ningún jugador alcance la casilla 63 (casilla final del tablero), la aplicación escogerá al jugador que tenga el turno, comprobará si tiene que estar algún turno sin jugar con lo cual cambiará de jugador activo y comenzará la fase de juego. Se lanzarán los dados conjuntamente y se actualizará al jugador que posea el turno de juego y se moverán las fichas por el tablero. Dependiendo de la casilla donde este, se bonificará o penalizará al jugador y se cambiará el turno de juego, o no, dependiendo de la posibilidad de volver a "tirar".

#### *8.3.3.1 El lanzamiento de los dados*

En esta fase se define el protocolo del lanzamiento de los dados de forma conjunta. Su funcionamiento descriptivo será el siguiente:

Si se es el jugador que tiene el turno de juego en ese momento, una vez el usuario haya dado indicaciones a la aplicación de que se ha de lanzar el dado (mediante la pulsación de un botón de la interfaz), se calculará el resultado del lanzamiento de varias formas (en claro y cifrado):

**En claro**: Se calcula, mediante el uso de mecanismos de calculo aleatorio, un numero. Dicho numero se ha de cifrar con una clave que se utilizará para cada lanzamiento (así se evita que la clave utilizada para un lanzamiento no sirva para ningún otro lanzamiento).

**Cifrado**: El resultado de cifrar con la clave el resultado en claro.

Una vez calculado estos valores, se envía a todos los jugadores de la partida, el valor del dado de forma cifrada y se esperará a que todos ellos le envíen los suyos (valores cifrados de sus lanzamientos individuales).

Cuando esto se haga efectivo, enviará el valor de la clave utilizada para cifrar el dado y esperará recibir del resto las claves utilizadas por los demás.

Finalmente, calcula el valor del dado colectivo, descifrando los dados cifrados con las claves recibidas y sumando los resultados, haciendo una operación de modulo 6, que dejará un resultado entre 0 y 5. El cero se tomará como un resultado de valor 6.

Si por el contrario, el usuario no tiene el turno, tendrá que esperar a que el que tiene el turno le envíe el valor cifrado de su dado y entonces se seguirá el mismo ciclo. En la imagen podemos ver el esquema seguido.

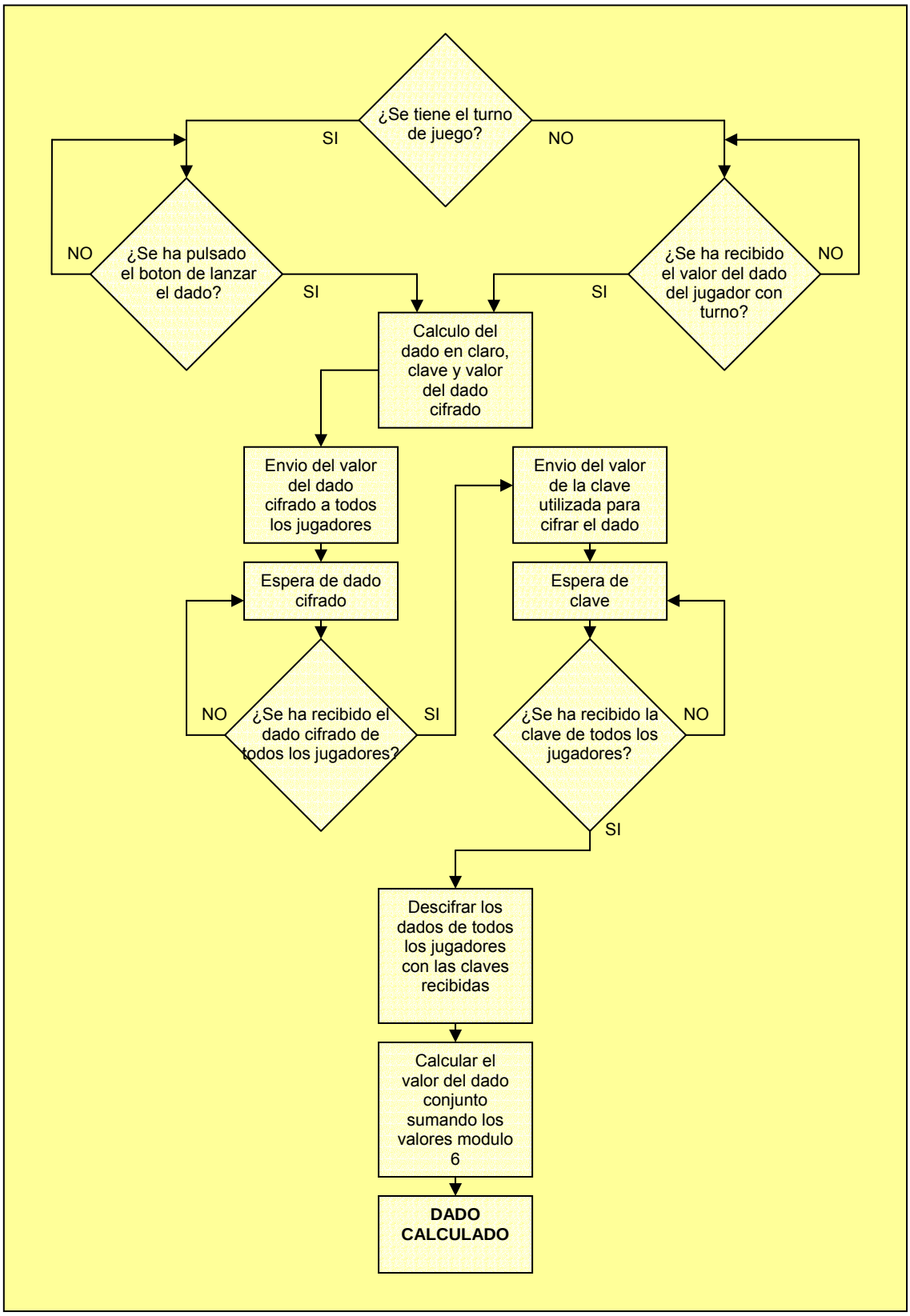


*Figura 10. Calculo del dado de forma segura. utilización de la técnica de compromiso de BIT*

## 8.4 Clases del dominio.

Inicialmente, las clase que podrían ser utilizadas en la aplicación, se han podido intuir del dibujo explicativo del funcionamiento del juego visto en la introducción. Estas son:

- **Juego**: La clase que representa el juego de la oca. En esta clase se tendrá que llamar a los métodos encargados de hacer todas las operaciones de inicialización, conexión y control del juego en sí.

- **Partida**: Esta clase representa una partida en juego. Se podrán manejar todos los jugadores enrolados en la partida o enviar/recibir información a/de todos ellos. Desde aquí se podrá calcular el lanzamiento del dado conjunto y verificar el estado de la partida (finalizada o no). Desde aquí se inicializará una partida o se conectará a una existente.

- **Jugador**: Esta clase contendrá información del estado del jugador, como por ejemplo el turno de juego asignado, el nombre del jugador, la dirección IP y puerto de conexión, la casilla en la que esta en ese momento, la información correspondiente a las bonificaciones y/o penalizaciones del juego (turnos sin tirar, si vuelve a tirar, etc.), la ficha que utilizará para jugar y demás.

- **Dado**: Esta clase proporcionará los métodos necesarios para lanzar el dado, obtener el lanzamiento cifrado, obtener la clave utilizada, etc. Es como un dado normal pero con un mecanismo capaz de cifrar el resultado.

- **Interfaz**: Esta clase representará la interfaz utilizada para el juego de la oca. Es aquí donde se definirá la interfaz necesaria para jugar y se crearán los botones necesarios para interactuar durante la partida, así como las imágenes representativas del tablero de juego, de una ficha o de unos dados, áreas de texto, listas, etc.

- **Tablero**: Desde esta clase se podrán obtener las bonificaciones o penalizaciones de una casilla en concreto.

- **Ficha**: Clase representativa de una ficha de juego. Tendrá un identificador que se corresponderá con el turno del jugador que la controle. Se podrá obtener su casilla actual y su posición física dentro del tablero dibujado en la interfaz.

En una primera instancia, el diagrama de clases del dominio, podría quedar de la siguiente forma:

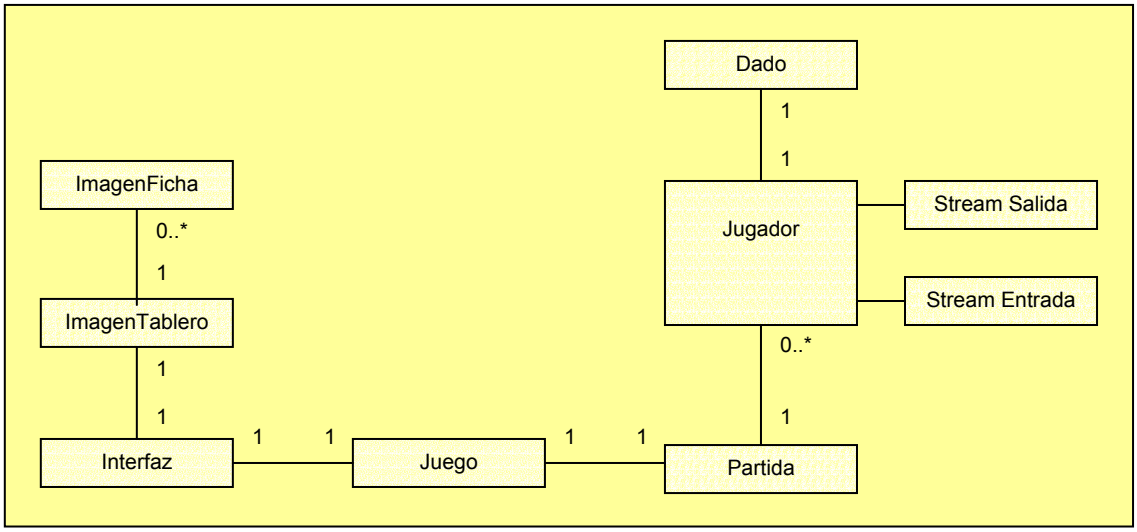


*Figura 12. Diagrama de clases del dominio*

## 8.5 Diagrama de estado.

Podemos confeccionar el diagrama de estado de la partida, que describe como funcionará una partida. El diagrama podría quedar de la forma siguiente:

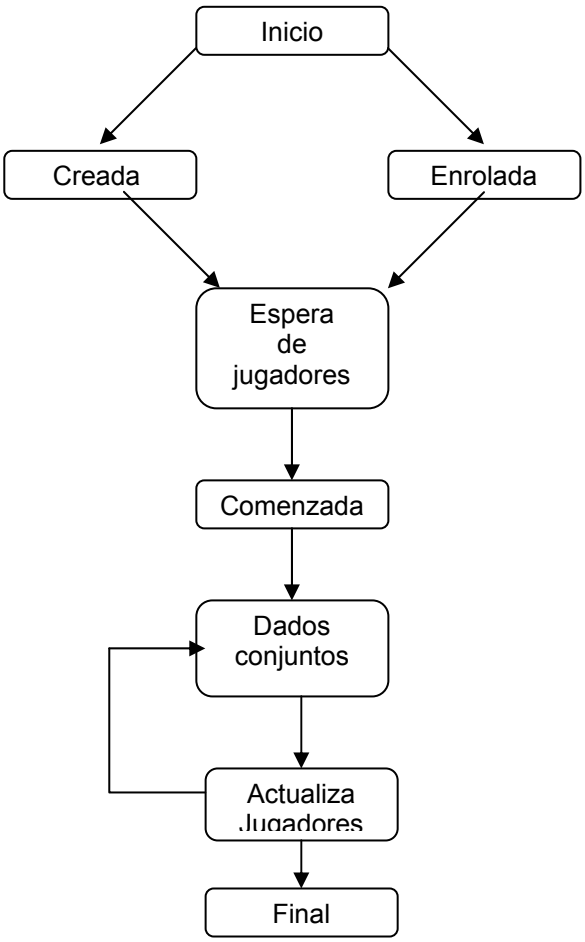


*Figura 13. Diagrama de estados de la aplicación*

## 8.6 Casos de uso.

Básicamente se pueden crear los casos de uso correspondientes a la figura.

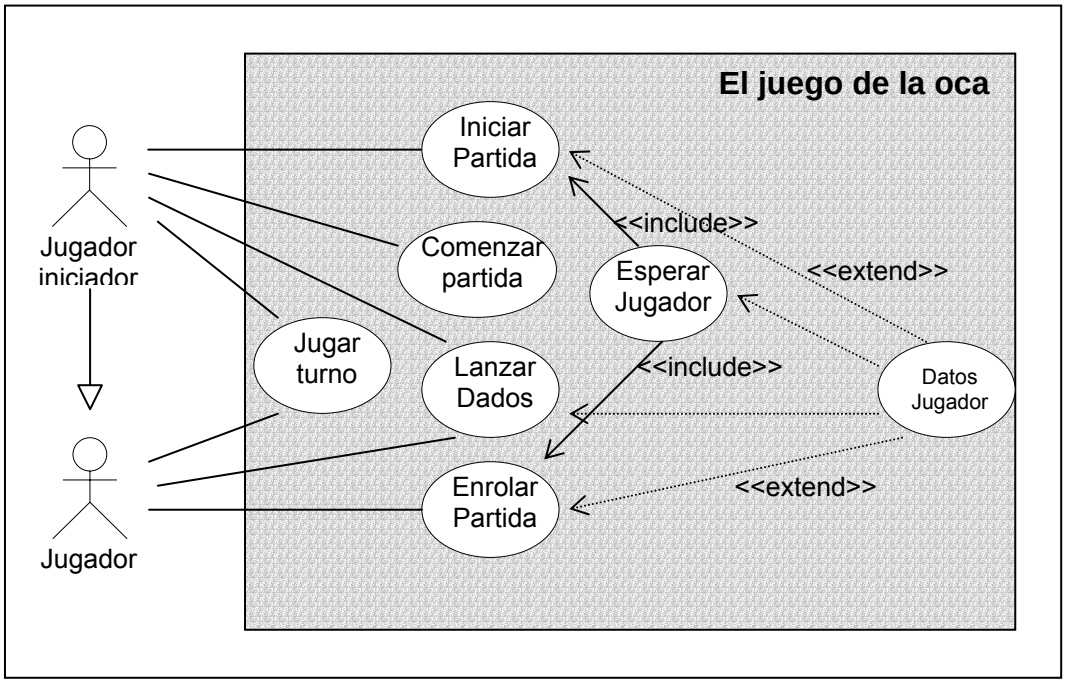


*Figura 14. Diagrama de casos de uso*

De donde se puede extraer la siguiente información:

Existen dos actores principales, el jugador normal y el jugador iniciador, que es una extensión del primero, ya que puede realizar las mismas acciones que él pero añadiendo el iniciar la partida y la decisión de comenzar a jugar.

Existen también varios casos de uso, iniciar partida se refiere a la posibilidad de iniciar una partida desde cero, una vez iniciada incluye el caso de uso *esperar jugador* que espera a posibles jugadores que se quieren enrolar en la partida. El caso de uso enrolar a partida lo realiza el actor jugador, este se conecta a una partida existente y también espera a posibles jugadores que quieran enrolarse a su vez a la misma partida.

El caso de uso datos jugador incluye el dar de alta a los jugadores e inicializarlos. El caso de uso lanzar dados, que puede realizarlo los dos actores definidos, lanza un dado de forma conjunta con el resto y actualiza los datos del jugador que tiene el turno asignado. Por último el caso de uso comenzar partida lo puede hacer el actor iniciador y explica como una vez dada de alta una partida nueva (no enrolada, ya que el actor jugador normal no puede decidir el comenzar una partida) se comienza a jugar.

## 8.7 Documentación de los casos de uso.

### 8.7.1 Caso de uso. Inicialización de una partida

Este caso de uso explica como se inicializa una partida en fase de análisis. No se especifica como se diseñará mas tarde cada una de las tareas pendientes, como por ejemplo, la inicialización del contexto de seguridad o comunicaciones.

***Caso de uso:*** *Iniciar partida*
***Resumen de la funcionalidad:*** *Inicia una nueva partida*
***Actores:*** ***Iniciador***
***Casos de uso relacionados:*** ***CDU. Espera Jugador***
***Precondición:*** *Se quiere iniciar una nueva partida*
***Poscondición:*** *Se ha iniciado una nueva partida*
***Proceso normal principal:***

1. *La aplicación inicializa la interfaz*
2. *La aplicación inicializa el contexto de seguridad*
3. *La aplicación inicializa al jugador local*
4. *La aplicación inicializa el contexto de comunicaciones entrantes*
5. *Se llama al* ***CDU. Espera Jugador.***

### 8.7.2 Caso de uso. Enrolarse a una partida ya iniciada

Este caso de uso explica como se conecta a una partida previamente inicializada (no comenzada).

***Caso de uso:*** *Enrolar partida*
***Resumen de la funcionalidad:*** *Se conecta a una partida ya inicializada*
***Actores:*** ***Jugador***
***Casos de uso relacionados:*** ***CDU. Espera Jugador***
***Precondición:*** *Se quiere conectar a una partida inicializada. La partida tiene que estar inicializada.*
***Poscondición:*** *Se ha conectado con la partida.*
***Proceso normal principal:***

1. *La aplicación inicializa la interfaz*
2. *La aplicación inicializa el contexto de seguridad*
3. *La aplicación inicializa el contexto de comunicaciones*
4. *La aplicación inicializa al jugador local (sin turno asignado).*
5. *La aplicación se conecta con un proceso remoto.*
6. *La aplicación envía información de nuestro jugador (nombre, dirección, etc.)*
7. *La aplicación recibe información del jugador que se ha conectado.*
8. *La aplicación recibe información de los jugadores enrolados a la partida.*
9. ***Por cada uno de los jugadores enrolados en la partida***
   9.1. *La aplicación se conecta con el jugador de la lista*
   9.2. *La aplicación envía información de nuestro jugador*
   9.3. *La aplicación añade un jugador a su lista de jugadores activos.*
10. *La aplicación añade al jugador que se ha conectado inicialmente*
11. *La aplicación establece el turno de juego al jugador local, que será el numero de jugadores de la partida conectada, mas uno.*
12. *La aplicación establece el contexto de comunicaciones entrantes.*
13. *Se llama al* ***CDU. Espera Jugador.***

***Alternativas de proceso y excepciones:***

*4a. La aplicación no se puede conectar con la partida*
*4a1. La aplicación sale del programa.*

### 8.7.3 Caso de uso. Esperar jugadores entrantes.

En este caso de uso, se explica como la aplicación espera conexiones de jugadores entrantes. Este CDU lo pueden inicializar tanto el iniciador como un jugador normal, y también puede ser llamado automáticamente por la aplicación.

| | |
|---|---|
| ***Caso de uso:*** | *Espera Jugador* |
| ***Resumen de la funcionalidad:*** | *Espera conexiones de jugadores nuevos o invitados.* |
| ***Actores:*** | ***Iniciador, Jugador*** |
| ***Casos de uso relacionados:*** | ***Ninguno*** |
| ***Precondición:*** | *Se esta en una partida (iniciada o enrolada) y se esperan posibles conexiones de jugadores.* |
| ***Poscondición:*** | *Se ha conectado un jugador o no.* |

***Proceso normal principal:***

1. *La aplicación espera durante un tiempo la conexión de jugadores.*
2. *La aplicación lee un comando de entrada (un mensaje del jugador conectado)*
3. *La aplicación determina un mensaje de* ***nuevo jugador***
   - *3.1. La aplicación lee información del jugador nuevo (nombre, dirección, etc.)*
   - *3.2. La aplicación envía información del jugador local.*
   - *3.3. La aplicación envía información de todos los jugadores enrolados en la partida.*
   - *3.4. La aplicación añade a la lista de jugadores al nuevo jugador.*
4. *La aplicación determina un mensaje de* ***jugador invitado****.*
   - *4.1. La aplicación lee información del jugador invitado (nombre, dirección, etc.)*
   - *4.2. La aplicación añade al jugador invitado en la lista de jugadores enrolados en la partida.*

***Alternativas de proceso y excepciones:***

*2a. No se ha conectado ningún jugador durante el tiempo de espera.*
*2a1. La aplicación sale del CDU.*
*3a La aplicación no determina el mensaje recibido*
*3a1. La aplicación sale del CDU.*

### 8.7.4 Caso de uso. Comenzar partida.

En este caso de uso, se explica la orden de comienzo de partida que solamente el iniciador puede activar.

| | |
|---|---|
| ***Caso de uso:*** | *Comenzar partida* |
| ***Resumen de la funcionalidad:*** | *Comienza la partida.* |
| ***Actores:*** | ***Iniciador*** |
| ***Casos de uso relacionados:*** | ***Ninguno*** |
| ***Precondición:*** | *Existe una partida inicializada y no hay jugadores pendientes de dar de alta.* |
| ***Poscondición:*** | *Se ha comenzado una partida y se espera el lanzamiento de los dados.* |

***Proceso normal principal:***

1. *La aplicación envía a todos los jugadores el mensaje de comienzo de partida*

### 8.7.5 Caso de uso. Jugar turno.

En este caso de uso, se explica el caso de uso principal que es el de jugar un turno.

| | |
|---|---|
| ***Caso de uso:*** | *Jugar turno* |
| ***Resumen de la funcionalidad:*** | *Hace la funcionalidad de jugar un turno de juego* |
| ***Actores:*** | ***Iniciador, Jugador*** |
| ***Casos de uso relacionados:*** | ***CDU lanzar dados*** |
| ***Precondición:*** | *Existe una partida inicializada y comenzada.* |
| ***Poscondición:*** | *Se ha jugado un turno del juego de la Oca.* |
| ***Proceso normal principal:*** | |

2. *La aplicación obtiene al jugador que tiene el turno actual.*
3. *El jugador que tiene el turno es el local*
   3.1. *La aplicación espera hasta la pulsación del botón de lanzar dados.*
4. *El jugador que tiene el turno no es el local*
   4.1. *La aplicación espera hasta recibir el valor de los dados cifrados de dicho jugador*
5. *La aplicación ejecuta el* ***CDU lanzar dados***
6. *La aplicación envía el mensaje DADO CIFRADO a todos los jugadores*
7. *La aplicación envía el valor de los dados cifrados*
8. *La aplicación espera hasta recibir el valor de los dados cifrados de todos los jugadores*
9. *La aplicación envía el mensaje DADO CLAVE a todos los jugadores*
10. *La aplicación envía el valor de la clave utilizada para cifrar el valor de los dados*
11. *La aplicación espera hasta recibir la clave de todos los jugadores*
12. *La aplicación calcula el dado conjunto utilizando las claves recibidas para descifrar los valores cifrados recibidos de todos los jugadores.*
13. *La aplicación actualiza el valor de la casilla del jugador, y del turno actual al siguiente.*
14. *La aplicación calcula la bonificación o penalización de la casilla en la que se encuentra el jugador.*

### 8.7.6 Caso de uso. Lanzar dados.

En este caso de uso, se explica como se realiza el lanzamiento de los dados. Se puede consultar la explicación del protocolo del lanzamiento de los dados desde esta referencia.

| | |
|---|---|
| ***Caso de uso:*** | *Lanzar dados* |
| ***Resumen de la funcionalidad:*** | *Hace la funcionalidad de lanzar unos dados* |
| ***Actores:*** | ***Iniciador, Jugador*** |
| ***Casos de uso relacionados:*** | ***Ninguno*** |
| ***Precondición:*** | *Existe una partida inicializada y comenzada y se quiere lanzar los dados.* |
| ***Poscondición:*** | *Se han lanzado los dados* |
| ***Proceso normal principal:*** | |

15. *La aplicación calcula un valor aleatorio que será el valor del lanzamiento de los dados*
16. *La aplicación calcula una clave de cifrado*
17. *La aplicación cifra el resultado de los dados*
18. *La aplicación devuelve el valor de los dados cifrados y en claro, así como de la clave utilizada para cifrar dichos dados.*

# 9 Diseño de la aplicación.

En este apartado veremos como se ha diseñado lo analizado en el apartado anterior, es decir, como se implementa en la aplicación utilizando las librerías que proporciona Java para seguridad y comunicaciones, así como los diferentes protocolos de lanzamiento de dados y de alta de nuevos usuarios, es decir, todo lo referente al análisis de la aplicación.

Para ello, se citaran las clases utilizadas y su funcionalidad a nivel de diseño, es decir, especificando claramente todos y cada uno de los métodos utilizados a la hora de programar la aplicación.

El diseño se ha llevado a cabo, como ya se ha explicado, utilizando el lenguaje de programación Java. Debido a su naturaleza orientada a objetos, se utilizará el lenguaje UML para especificar las clases del dominio (clases de diseño en esta parte del desarrollo de la aplicación) vistas en el apartado de análisis de la aplicación.

Primero, analizaremos los casos de uso para diseñar los diagramas de secuencias que servirán para ver los mensajes que se necesitan entre clases. Después veremos como se han diseñado cada una de las clases necesarias para la implementación de la aplicación.

## 9.1 Diagramas de secuencias

A continuación estudiaremos los diferentes casos de uso para realizar el diseño, es decir, lo que se denomina casos de uso reales a través de los diagramas de secuencias. Mediante estos diagramas podemos saber que mensajes hacen falta pasar entre clases y así poder implementar dichas clases.

Veremos todos los casos de uso analizados en el capítulo anterior.

### 9.1.1 Diagrama del caso de uso Inicialización de partida

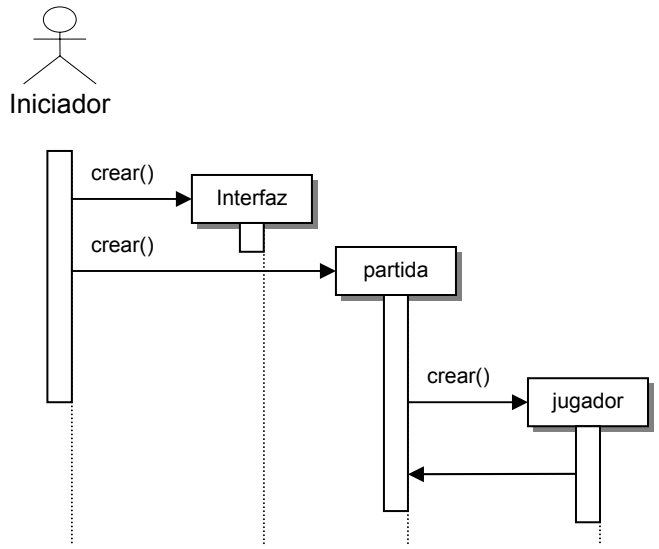


*Figura 15. Diagrama de secuencias. Inicialización de partida*

La inicialización de la partida la realiza la figura del iniciador. Como se observa, se inicializa la interfaz y acto seguido la clase partida (que se encarga de inicializar las comunicaciones y la seguridad). Después se crea al jugador local.

### 9.1.2 Diagrama del caso de uso enrolarse a una partida.

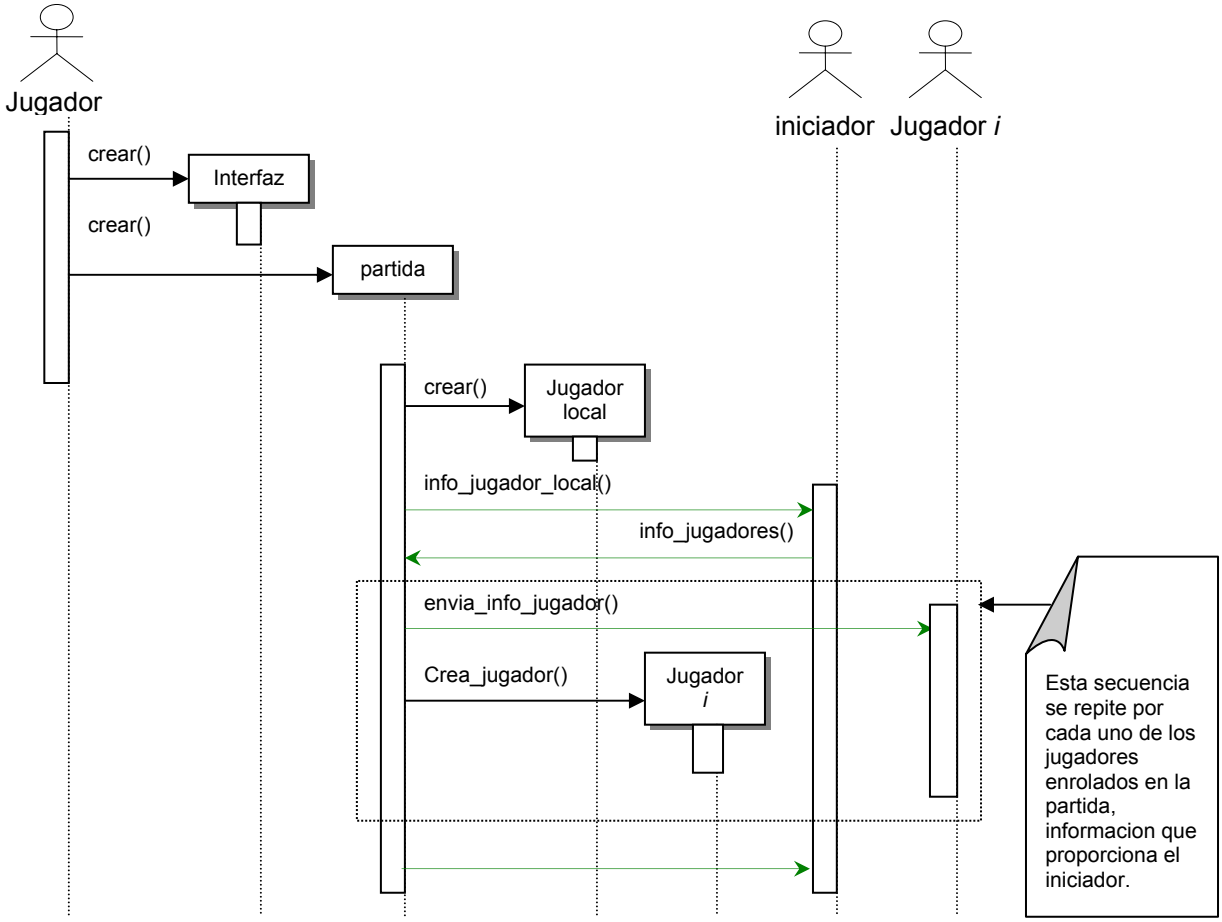


*Figura 16. Diagrama de secuencias. Enrolarse en una partida*

Como se observa, tenemos varios actores que entran en juego. El actor Jugador que inicia la secuencia, es el jugador local. Los otros dos son jugadores remotos, es decir, el iniciador esta en otro proceso. Se han indicado con una flecha de color verde y con diferente punta los accesos a actores remotos a la aplicación.

Al principio se crea la interfaz y la partida, que se encarga de inicializar las comunicaciones SSL. Después la clase partida crea al jugador local (nombre, dirección IP y puerto). A continuación se conecta con el iniciador de la partida y se le envía la información de nuestro jugador. Él responde devolviendo la información del numero de jugadores enrolados en la partida, así como sus nombres, direcciones IP y puertos de comunicaciones.

A continuación, tendremos que comunicarnos con cada uno de los jugadores de la partida enviándole nuestro nombre, dirección IP y puerto de comunicaciones y crear un jugador por cada uno con su información.

Finalmente, se avisa al jugador iniciador que se ha finalizado con el contacto con todos los jugadores de la partida. Este punto es importante para que el iniciador no comience la partida antes de que el nuevo jugador se ponga en contacto con los demás, ya que si así fuera, existirían incongruencias en la comunicación.

### 9.1.3 Diagrama de secuencia esperar jugadores entrantes

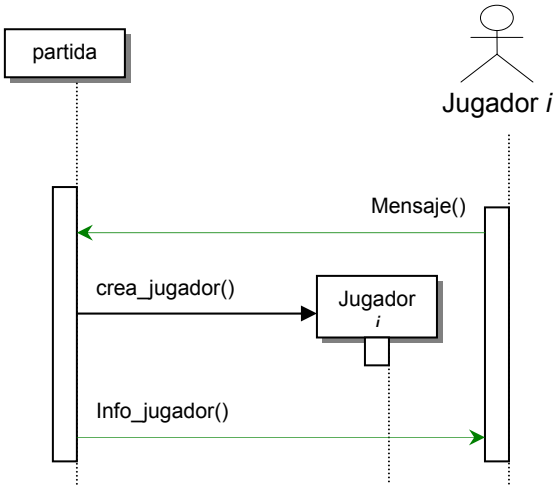


*Figura 17. Diagrama de secuencias. Esperar jugadores.*

Como se puede ver en el diagrama, existe un actor que es remoto. Cuando este jugador se quiere conectar con nosotros, nos envía un mensaje (NUEVO o INVITADO). Dependiendo de si es nuevo jugador o invitado, se hará un proceso u otro. En cualquier caso, se ha de dar de alta como un jugador nuevo. Lo único que difiere es si le tenemos que enviar información (Si somos iniciador) o no (no somos iniciador).

### 9.1.4 Diagrama de secuencia comenzar partida

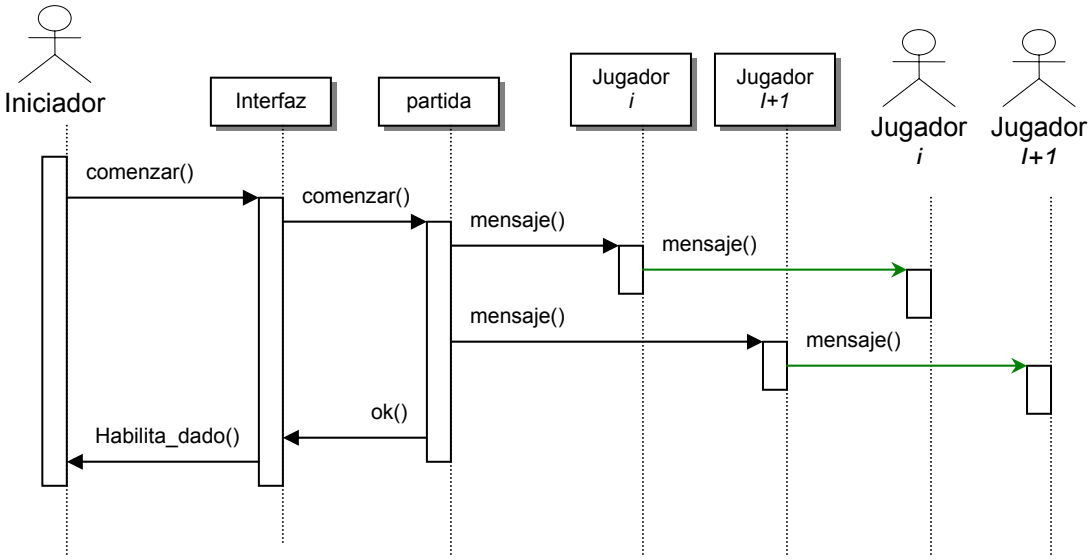


*Figura 18. Diagrama de secuencias. Comenzar partida.*

En este diagrama de secuencias, el iniciador de la partida decide cuando comenzar. Avisa a la aplicación pulsando un botón de la interfaz e inmediatamente la partida avisa a todos los jugadores enrolados mediante un mensaje.

### 9.1.5 Diagrama de secuencia jugar turno

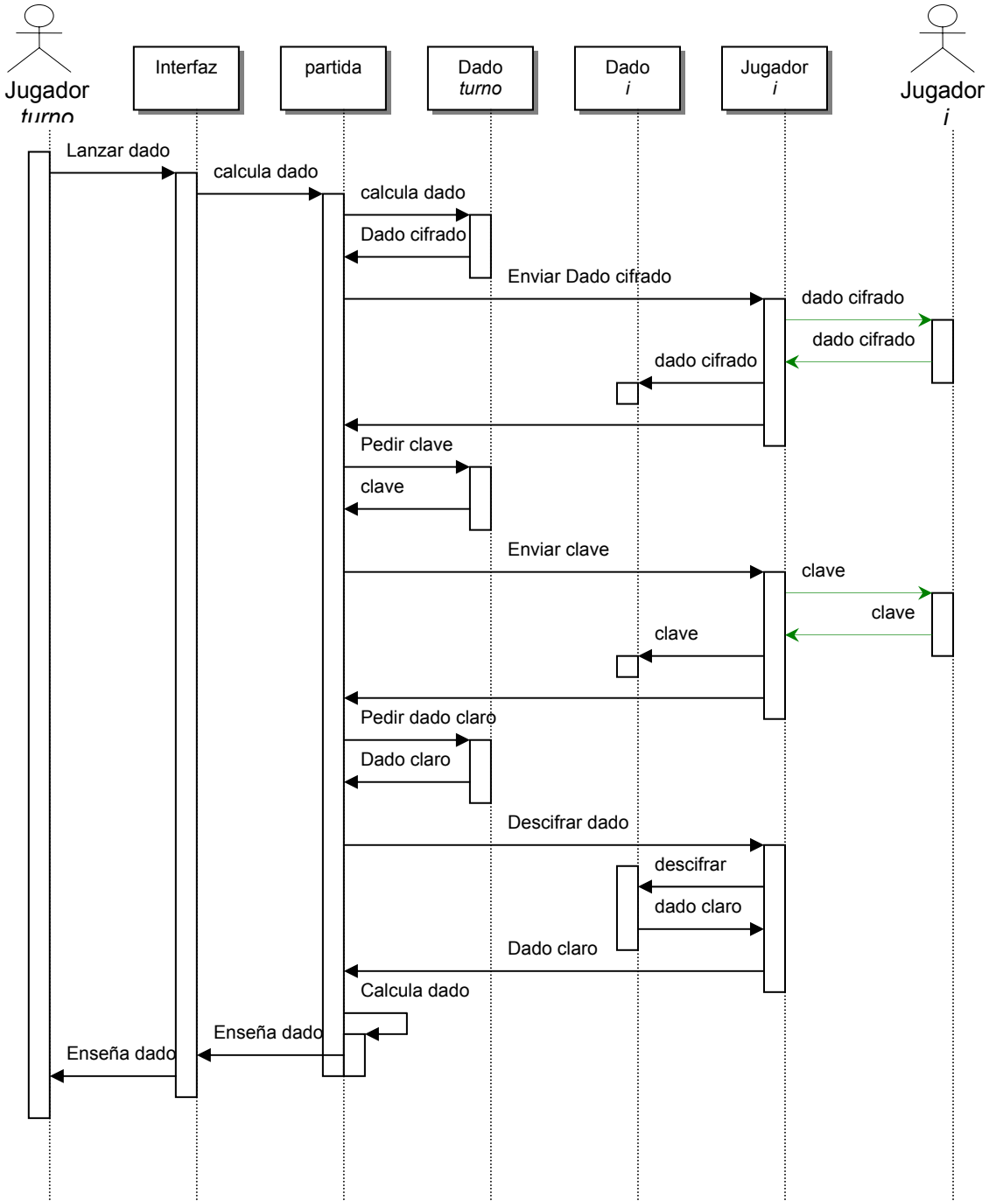


*Figura 19. Diagrama de secuencias. Jugar turno.*

En este diagrama de secuencias se puede observar el juego de un turno, por parte del jugador que tiene dicho turno. Se explica el calculo del dado conjunto y el resultado de los dados. Por un lado tenemos la figura del jugador que tiene el turno, que a su vez, tiene asignado un dado, desde el que puede calcular el lanzamiento en claro, cifrado y la clave utilizada para cifrarlo. Se puede observar también el jugador *i*, que representa cada uno de los jugadores enrolados en la partida. Este jugador (*i*) tiene asignado también un dado, y se actualizan los valores del dado a través de mensajes que se reciben de los jugadores remotos. Finalmente es la partida quien calcula el dado conjunto con los datos recibidos de todos los jugadores.

## 9.2 Diseño de las clases del dominio

En este apartado veremos como se han diseñado las diferentes clases que componen el diagrama de clases del dominio. Recordemos dicho diagrama en el siguiente gráfico, al que se le añade una nueva clase, la imagen del dado (*ImagenDado*). Se utilizará para visualizar en pantalla la imagen del dado y crear efectos, como por ejemplo, el del lanzamiento del dado al aire.

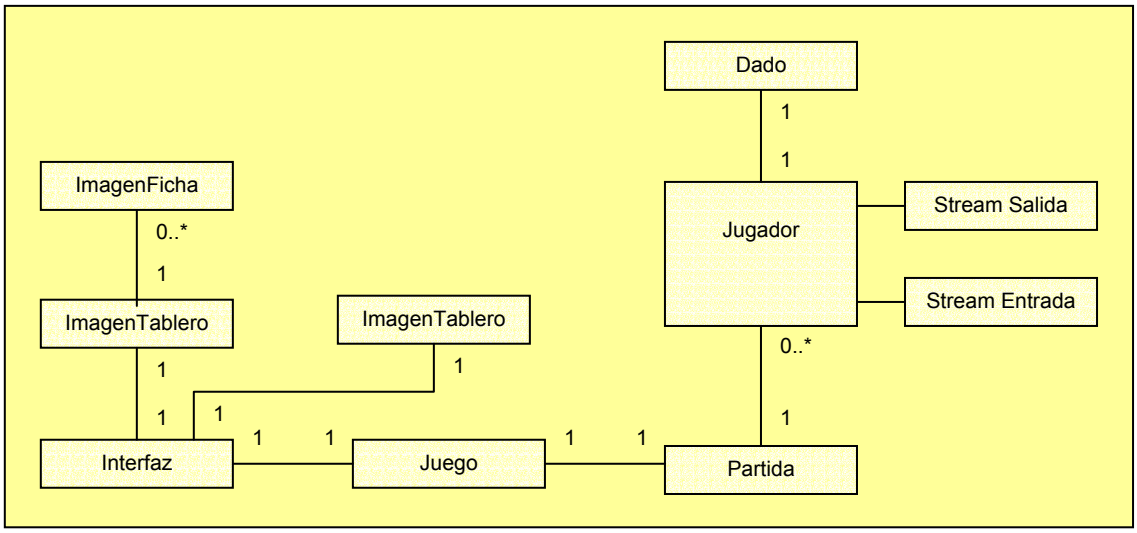


*Figura 20. Diagrama de clases del dominio definitiva.*

### 9.2.1 Clase Interfaz

La clase que se ocupa de inicializar y definir la interfaz del juego es la clase *Interfaz.*

### 9.2.2 Clase Partida

La clase partida es la más importante de la aplicación. Dentro de esta clase tenemos métodos para inicializar la partida de las dos formas posibles (inicializarla o enrolarse), métodos de espera de jugadores entrantes y métodos de calculo de lanzamiento de los dados.

### 9.2.3 Clase Jugador

La clase jugador es la imagen de un jugador de la partida. Tenemos propiedades como por ejemplo el nombre, la dirección y puerto de comunicaciones. Esta clase se tendrá que implementar como un hilo de ejecución paralelo al normal, de esta forma cuando un jugador quiera enviar un mensaje, no tendremos que preguntarlo directamente al jugador remoto, si no que el jugador remoto enviará un mensaje y actualizará valores de la clase, como por ejemplo, el valor del dado cifrado, o la clave utilizada para cifrarlo. También proporciona métodos para actualizar la casilla del tablero del juego y bonificaciones o penalizaciones del juego.

### 9.2.4 Clase Dado

Esta clase representa un dado que al lanzarlo obtiene el valor en forma cifrada o normal. Así, contendrá propiedades como la clave utilizada y el resultado de esos valores. Cada jugador de la partida tiene asociado un dado particular, ya que para el computo del dado cada uno de ellos lanza uno de forma individual, para luego calcularlo en conjunto.

### 9.2.5 Clase ImagenDado

Esta clase representa la imagen de un dado para insertar en la interfaz. Contiene un método de simulación de lanzamiento de dado, así como reimplementa métodos de pintado en pantalla.

### 9.2.6 Clase ImagenTablero

Esta clase representa la imagen de un tablero de juego. Además proporciona métodos para añadir fichas al juego, calcular casillas dependiendo del valor de un dado, o movimientos de fichas dentro del tablero. Desde esta clase podremos obtener las bonificaciones de una casilla y sus penalizaciones.

### 9.2.7 Clase ImagenFicha

Esta clase representa la imagen de una ficha en el tablero. Contiene propiedades como la posición física en pantalla de la ficha y métodos para obtener dichas coordenadas. También se puede obtener la casilla en la cual se encuentra en un momento dado. Cada ficha tiene un identificador que se corresponde con el turno de juego del jugador que la controla.

### 9.2.8 Clase Mensajes

Esta clase proporciona los mensajes necesarios para implementar el protocolo de conexión de jugadores y de calculo del dado de forma conjunta.

### 9.2.9 Clase StreamEntrada

Representa un flujo de entrada desde un jugador remoto. Cada jugador mantiene un socket abierto hacia su imagen, y también un flujo de datos desde donde leer información.

### 9.2.10 Clase StreamSalida

Representa un flujo de salida hacia un jugador remoto. Cada jugador mantiene un socket abierto hacia su imagen, y también un flujo de datos hacia donde escribir información.

# 10 Implementación

En este capitulo veremos las particularidades de utilizar el lenguaje de programación Java para implementar la aplicación. La estructura de esta sección será siguiendo el programa resultante de la implementación, explicando cada detalle que sea importante en la implementación de los diferentes protocolos analizados y posteriormente diseñados, nos referimos a la comunicación y el protocolo de alta de nuevos jugadores, la seguridad (utilizando la librería SSL para java), el cifrado y descifrado utilizando DES y el lanzamiento del dado conjunto.

## 10.1 Las comunicaciones seguras y el protocolo de alta de jugadores

A la hora de implementar la aplicación en este aspecto, es decir, la comunicación y la seguridad, se han de tener en cuenta algunos aspectos:

1. Necesitamos almacenes de claves privadas/publicas para la autenticación de la parte servidor de la aplicación (conexiones entrantes). En este caso no se ha habilitado la autenticación de la parte cliente, ya que la seguridad que se proporciona ya es suficiente para el proyecto.
2. Necesitamos poder generar números aleatorios seguros.
3. Finalmente, necesitamos crear sockets seguros.

En un principio, la aplicación permite especificar los almacenes de claves que se quieren utilizar, pero si no se especifican, se utilizan dos almacenes previamente creados y que se distribuyen con el programa llamados *almacenprivado.jks* y *almacenpublico.jks*.

La clase que utiliza dichos almacenes es la clase *Partida*. Desde esta clase se crean las partidas y por lo tanto se crean los sockets servidores para permitir que nuevos usuarios puedan enrolarse a la partida. En el siguiente extracto de código se puede observar como se utilizan los almacenes para crear un contexto SSL de comunicación segura, como se genera un numero aleatorio seguro y finalmente como se crea el socket seguro:

```
//argumentos que estan definidos como parámetros de entrada:
//  ptPrivado o ptPublico = tipo de almacen privado o publico
//  pfPrivado o pfPublico = nombre del fichero de almacén privado o publico
//  ppPrivado o ppPublico = password del almacén privado/publico

//Crea un numero aleatorio seguro para contexto SSL
numeroAleatorioSeguro = new SecureRandom();
numeroAleatorioSeguro.nextInt();

//Setup los almacenes con claves privadas y publicas
//Tipo de los almacenes
almacenPrivado = KeyStore.getInstance( ((ptPrivado == null) ? "JKS" :
ptPrivado) );
almacenPublico = KeyStore.getInstance( ((ptPublico == null) ? "JKS" :
ptPublico) );

//ficheros y password de los almacenes
almacenPrivado.load( new FileInputStream( ((pfPrivado == null) ?
"almacenprivado.jks"  : pfPrivado) ), ((pfPrivado == null) ? "password" :
ppPrivado).toCharArray() );
almacenPublico.load( new FileInputStream( ((pfPublico == null) ?
"almacenpublico.jks" : pfPublico) ), ((pfPublico == null) ? "password" :
ppPublico).toCharArray() );

//Trust manager para los certificados firmados por una CA de la clave publica
TrustManagerFactory tmf = TrustManagerFactory.getInstance( "SunX509" );
tmf.init( almacenPublico );

//Key Manager para las claves privada/publica del jugador
KeyManagerFactory kmf = KeyManagerFactory.getInstance( "SunX509" );
kmf.init( almacenPrivado , ((pfPrivado == null) ? "password" :
ppPrivado).toCharArray() );
```

```
//Inicializa contexto SSL
contextoSSL = SSLContext.getInstance( "SSL" );

contextoSSL.init( kmf.getKeyManagers(), tmf.getTrustManagers(),
numeroAleatorioSeguro );

//Setup del server socket
SSLServerSocketFactory sf = contextoSSL.getServerSocketFactory();

//Crea el socket servidor
socketServidor = (SSLServerSocket)sf.createServerSocket( puertoLocal );
…
```

Una vez creado el socket servidor, se pueden aceptar conexiones entrantes. Esta operación se realiza desde el método de espera de conexiones que contiene la misma clase: la clase *Partida*.

```
//Acepta conexiones entrantes
SSLSocket socketNuevoJugador = (SSLSocket)socketServidor.accept();
```

Cuando se ha recibido la conexión de un nuevo jugador, la comunicación se hace utilizando flujos de comunicación (de entrada y de salida):

```
//Stream de entrada
StreamEntrada i = new StreamEntrada(socketNuevoJugador.getInputStream());
StreamSalida o  = new StreamSalida(socketNuevoJugador.getOutputStream());
```

A partir de aquí, se pueden enviar y recibir mensajes desde o hacia nuevos jugadores, utilizando estos *streams* creados:

```
…
…
//Lee el mensaje que ha recibido
String comando = i.leeLinea();

//Se atiende la peticion de nuevo jugador
if( comando.equals( Mensajes.NUEVO_JUGADOR ) ) {

    //Si el jugador local es el iniciador de la partida
    if( iniciada )
    {
        //Recibe la información del jugador remoto
        nom = i.leeLinea();
        dir = i.leeLinea();
        prt = i.leeLinea();
        …
        …
```

### 10.1.1 Iniciar una partida

Para iniciar una partida, simplemente se inicializa el contexto SSL y se crea el socket servidor, como ya hemos visto en los cuadros de código anteriores.

### 10.1.2 Enrolarse a una partida

El procedimiento de enrolarse a una partida, básicamente es la misma que iniciar una partida pero se añade el protocolo de nuevos jugadores.

En este caso, a parte de inicializar un socket servidor (para posibles jugadores invitados, que se conecten mas tarde), se ha de inicializar un socket cliente, que servirá para conectarse al iniciador de la partida:

```
//Inicializa las factorías para el socket servidor y cliente
```

```
SSLServerSocketFactory sf = contextoSSL.getServerSocketFactory();
SSLSocketFactory sfc = contextoSSL.getSocketFactory();

//Se crea un socket seguro a través del protocolo SSL hacia la dirección
indicada
SSLSocket nuevoSocket = (SSLSocket)sfc.createSocket( pDireccionIP, pPuerto );

//Se crean los streams de comunicación
StreamEntrada i = new StreamEntrada( nuevoSocket.getInputStream() );
StreamSalida  o = new StreamSalida( nuevoSocket.getOutputStream() );

//Protocolo de comunicación de nuevo usuario, se envía mensaje de petición de
nuevo jugador
o.escribeLinea( Mensajes.NUEVO_JUGADOR );

//Envía información de mi jugador
o.escribeLinea( miJugador.getNombre() );
o.escribeLinea( miJugador.getDireccionIP() );
o.escribeLinea( String.valueOf( miJugador.getPuerto() ) );

//Recibe información del jugador remoto
nom = i.leeLinea();
dir = i.leeLinea();
prt = i.leeLinea();
tur = i.leeLinea();

//Leemos la información de los jugadores conectados al remoto
int enrolados = Integer.parseInt( i.leeLinea() );
for( int j = 0 ; j < enrolados; j++ )
{
   //Lee los datos del jugador de la lista del remoto
   nom2 = i.leeLinea();
   dir2 = i.leeLinea();
   prt2 = i.leeLinea();
   tur2 = i.leeLinea();

   //Establece un socket hacia el jugador
   SSLSocket invitadoSocket = (SSLSocket)sfc.createSocket( dir2,
Integer.parseInt( prt2 ) );

   //Se crean los streams de comunicación
   StreamEntrada i2 = new StreamEntrada( invitadoSocket.getInputStream() );
   StreamSalida  o2 = new StreamSalida( invitadoSocket.getOutputStream() );

   //Escribe el mensaje de jugador invitado
   o2.escribeLinea( Mensajes.INVITADO_JUGADOR );

   //Envía información de mi jugador
   o2.escribeLinea( miJugador.getNombre() );
   o2.escribeLinea( miJugador.getDireccionIP() );
   o2.escribeLinea( String.valueOf( miJugador.getPuerto() ) );
}

//Avisa al jugador remoto que hemos acabado de conectarnos con todos los
jugadores de la partida
nuevoJugador.escribe( Mensajes.OK );
```

Como se puede observar el protocolo sigue el diseño especificado.

1. se crea el socket cliente hacia la dirección IP y puerto del iniciador de la partida
2. se crean los flujos de comunicación
3. se envía el mensaje de NUEVO JUGADOR
4. se envía información de nuestro jugador: nombre, dirección y puerto
5. se recibe la información del iniciador de la partida
6. por cada jugador que ya este en la partida, se lee su nombre, dirección y puerto, y se crea un socket hacia él, para enviarle el mensaje de INVITADO y enviar a su vez la información de nuestro nombre, dirección y puerto

7. Finalmente, se envía el mensaje de OK al iniciador, para indicarle que ya se ha contactado con todos los jugadores de la partida y que el protocolo a finalizado.

La clase que define los mensajes necesarios para implementar todos los protocolos de comunicación y de lanzamiento del dado es la clase *Mensajes.*

### 10.1.3 Esperar conexiones de jugadores

Para esperar jugadores, se llama al método *espera_jugadores()* de la clase *Partida*. Aquí se definen los protocolos de comunicaciones de NUEVOS JUGADORES e INVITADOS.

El código explicativo es el siguiente:

```
//Acepta conexiones entrantes
SSLSocket socketNuevoJugador = (SSLSocket)socketServidor.accept();

//Stream de entrada
StreamEntrada i = new StreamEntrada(socketNuevoJugador.getInputStream());
StreamSalida o  = new StreamSalida(socketNuevoJugador.getOutputStream());

//Lee el mensaje que ha recibido
String comando = i.leeLinea();

//Se atiende la peticion de nuevo jugador
if( comando.equals( Mensajes.NUEVO_JUGADOR ) ) {

    //Si el jugador local es el iniciador de la partida
    if( iniciada )
    {

        //Recibe la información del jugador remoto
        nom = i.leeLinea();
        dir = i.leeLinea();
        prt = i.leeLinea();

        //Envía información de mi jugador
        o.escribeLinea( miJugador.getNombre() );
        o.escribeLinea( miJugador.getDireccionIP() );
        o.escribeLinea( String.valueOf( miJugador.getPuerto() ) );
        o.escribeLinea( String.valueOf( miJugador.getTurno() ) );

        //Envía información de mi jugadores activos
        o.escribeLinea( String.valueOf( numeroDeJugadores ) );

        //Enviamos información de todos los jugadores conectados
        for( int iterador = 0 ; iterador < numeroDeJugadores ; iterador++ )
        {
            //Enviamos información del jugador Conectado
            o.escribeLinea( ((Jugador)jugadores.get(iterador)).getNombre() );
            o.escribeLinea( ((Jugador)jugadores.get(iterador)).getDireccionIP()
);
            o.escribeLinea( String.valueOf(
((Jugador)jugadores.get(iterador)).getPuerto()) );
            o.escribeLinea( String.valueOf(
((Jugador)jugadores.get(iterador)).getTurno()) );
        }

        //Hasta que el jugador al que se le ha pasado la lista de jugadores de
la partida, no se ponga en contacto con todos los demás, no se puede seguir,
ya que si no existe la posibilidad de que el jugador local quiera comenzar la
partida y se bloqueen las comunicaciones
        while( !nuevoJugador.lee().equals( Mensajes.OK ) ) {}

        } else {
```

```
        //Envía no conforme, si se lee un mensaje de nuevo jugador y el local no
es iniciador de partida
        o.escribeLinea( Mensajes.NOK );
    }
}

//Se atiende la petición de jugador invitado por alguien
if( comando.equals( Mensajes.INVITADO_JUGADOR ) ) {

    //Recibe información del jugador invitado
    nom = i.leeLinea();
    dir = i.leeLinea();
    prt = i.leeLinea();

}
```

En este trozo de código se observa que dependiendo de si se es iniciador o no, se lee el mensaje de NUEVO_JUGADOR, para permitir que nuevos jugadores se enrolen a la partida. Si no se es iniciador de partida y se recibe el mensaje de NUEVO JUGADOR, no se ha de dar de alta, sin embargo si no se es iniciador, se ha de recibir mensajes de INVITADO.

## 10.2 EL protocolo del lanzamiento del dado

En el siguiente cuadro de código se puede observar el protocolo de lanzamiento del dado:

```
//Se lanza el dado de forma conjunta...
partida.getJugadorLocal().getDado().lanzar();

//lanza el dado y se lo envia a todos cifrado
partida.enviaATodos( Mensajes.DADOCIFRADO );
partida.enviaATodos( new Base64().encode(
partida.getJugadorLocal().getDado().getValorCifrado() ) );

//Espera recibir de todos el dado cifrado
partida.recibeDeTodos( Mensajes.DADOCIFRADO );

//envia a todos la clave utilizada para encriptar el dado
partida.enviaATodos( Mensajes.DADOCLAVE );
partida.enviaATodos( new Base64().encode(
partida.getJugadorLocal().getDado().getValorClave() ) );

//Espera recibir de todos las claves utilizadas para cifrar los dados
partida.recibeDeTodos( Mensajes.DADOCLAVE );

//Calcula el valor del lanzamiento de dados compartido
int valorDados = partida.calculaDadoConjunto();
```

En primera instancia se lanza el dado. La clase encargada de lanzar el dado es la clase *Dado*. En el método constructor del dado se inicializa el contexto para cifrar y descifrar mediante DES:

```
//inicializacion del generador de claves (tipo DES)
generadorClave = KeyGenerator.getInstance("DES");
cipher = Cipher.getInstance( "DES" );
```

Después el método de lanzamiento del dado, se encarga de cifrar el resultado de los dados.

```
//genera una clave DES
clave = generadorClave.generateKey();

//Obtiene el array de bytes representativo de la clave generada
valorClave = clave.getEncoded();

//lanza el dado generando un valor aleatorio seguro entre 0 y valorMayor
valorEnClaro = new SecureRandom().nextInt( valorMayor );
```

```
//Genera un secretkeyspec que utiliza un array de bytes donde esta la
//clave a utilizar para codificar y/o descodificar (tipo DES)
SecretKeySpec cSpec = new SecretKeySpec( valorClave, "DES" );

//inicialización para la encriptación (utilizando la clave anterior)
cipher.init(Cipher.ENCRYPT_MODE, (SecretKey) cSpec);

//encripta el resultado del valor del dado en claro
valorCifrado = cipher.doFinal( String.valueOf(valorEnClaro).getBytes() );
```

A partir de aquí, se implementa el protocolo del lanzamiento del dado:

1. Se envía el mensaje de envío del dado cifrado
2. Se envía el valor del dado cifrado, codificado en base64 (para evitar problemas de caracteres).
3. Se espera a recibir de todos los jugadores el valor del dado cifrado
4. Se envía el mensaje de envío de clave utilizada
5. Se envía el valor de la clave utilizada, también en codificación base64.
6. Se espera recibir de todos los jugadores el valor de la clave utilizada para cifrar los dados individuales.
7. Se calcula el dado conjunto. Lo calcula la clase *Partida*.

La clase encargada de realizar todas las llamadas a métodos y que contiene la lógica de la aplicación es la clase *Juego*.

## 10.3 La interfaz del juego

La clase Interfaz es una subclase de *Frame*. Cuando se crea una instancia de esta clase, se definen unos marcos (paneles) dentro de la ventana que son:

- **LayoutBotones**: Es la zona de la ventana dedicada a visualizar los diferentes botones que componen la Interfaz. Éstos son tres: botón de iniciar partida, botón de lanzar el dado y botón de salir del programa. En esta zona también se inicializa un gráfico que hace la función de dado (es solamente un gráfico y no se comporta como un botón). Este gráfico simula el comportamiento de un dado al lanzarlo al aire.

- **LayoutConexion**: Es la zona que comprende los mensajes de sistema, de error y la lista de jugadores conectados. Se compone de tres *layouts* diferentes:

    - **LayoutMensajesError**: Es el panel donde se enviarán los mensajes de error

    - **LayoutMensajesSistema**: Es el panel donde se visualizarán los mensajes que vaya generando la aplicación.

    - **LayoutQuienHayConectado**: Es la zona donde se visualizan los jugadores conectados.

- **LayoutCopyright**: Es la zona de pantalla que enseña el copyright del proyecto

- **LayoutTablero**: Es la zona donde se ve el gráfico del tablero del juego

De los métodos que define esta clase, podemos destacar como mas importantes:

- **EscribeMensajeError**: Escribe un mensaje de error en el área de texto definida para tal efecto.

- **EscribeMensajeSistema**: Escribe un mensaje de sistema en la zona de mensajes de sistema

Por lo demás, podemos encontrar métodos para deshabilitar o habilitar los diferentes botones que componen el área de botones de la interfaz.

En el siguiente gráfico podemos observar como se define la interfaz y como se visualiza en la pantalla de nuestro ordenador:

*Figura 31. Interfaz del juego de la Oca*

Se puede ver el código entero de la aplicación resultante en el anexo Código fuente de la aplicación.

# 11 Pruebas

Las pruebas realizadas sobre la aplicación resultante, aseguran el correcto funcionamiento de hasta cuatro jugadores distribuidos a través de una red corporativa. Es lógico pensar que el funcionamiento para más jugadores será también correcto, aunque cada vez que se conecta un nuevo jugador y debido al protocolo de negociación de SSL y al cifrado/descifrado de los dados, la aplicación funciona mas lentamente.

La aplicación se ha probado en un entorno de red corporativa utilizando direcciones internas. No se ha probado a través de Internet, pero puede ser que, debido a que las redes internas distribuidas utilizando servidores internos proporcionan direcciones relativas y no absolutas, la aplicación no funcione si las comunicaciones pasan a través de los citados servidores, es decir, desde una red interna hacia una dirección absoluta, debido a la naturaleza de las librerías de comunicaciones seguras SSL.

# 12 Conclusiones

El trabajo que aquí se presenta, junto con este documento, han sido elaborados como trabajo final de carrera en la Universitat Oberta de Catalunya, dentro del área de seguridad informática.

La experiencia de su realización ha sido muy enriquecedora, proporcionando una base práctica sobre métodos de seguridad (protocolo SSL, DES, compromiso de BIT), de comunicación entre procesos (sockets, protocolo de conexión y comunicación mediante mensajes) y de interfaces, así como experiencia en el campo de los juegos, enfocada hacia los juegos de azar que representa una parte muy importante dentro de dicho campo. Como he explicado en un capítulo anterior, la motivación para realizar y estudiar todos estos campos, se centra en dos apartados, la seguridad y los videojuegos.

La aplicación proporciona finalmente lo que en un principio los objetivos marcaban: seguridad en la comunicación, seguridad en el lanzamiento del dado, la sincronización en el estado de la partida por parte de todos los procesos y una interfaz usable y adecuada.

Las herramientas utilizadas se han adecuado también a los objetivos y solamente se ha utilizado una parte importante de las mismas. La parte más difícil (o que ha dado más trabajo) ha sido la inicialización del protocolo SSL, y la parte más delicada, los ficheros de certificados y claves, ya que si no se logra entender su funcionamiento, puede ser complicado utilizar otros certificados y claves diferentes a los que incluye la aplicación. Por otro lado, la parte más creativa es la de la creación de la interfaz y de la simulación del lanzamiento del dado.

En definitiva, aunque existen muchas posibles amplicaciones y mejoras (la interfaz, la comunicación de los errores o la mejora en cuanto a la velocidad de proceso debido al protocolo SSL), la aplicación proporciona las características expuestas en este documento, y cumple así con los objetivos propuestos al iniciar el estudio y analizar la aplicación, en la fase inicial de este proyecto.

# 13 Glosario

***SSL. Secure Socket Layer.*** Protocolo de seguridad que utiliza claves publicas y privadas para proporcionar codificación de datos, autenticación e integridad de mensajes para conexiones TCP/IP.

***DES. Data Encryption Standard.*** Esquema de cifrado/descifrado que proporciona un algoritmo de cifrado normalizado para redes de ordenadores consistente en la aplicación de sucesivas permutaciones y sustituciones del texto a cifrar/descifrar.

***Compromiso de BIT.*** Esquema que permite comprometerse a un valor frente otro participante a través de una comunicación no presencial, como por ejemplo, en el lanzamiento de una moneda telefónicamente

***Aplicación punto-a-punto.*** Una aplicación distribuida que no necesita de un servidor para controlar la posición global de la aplicación. Cada punto (proceso) mantiene dicha información.

***Clase.*** Conjunto de objetos que tienen los mismos atributos y operaciones

***Dominio.*** Porción del mundo real considerada por una aplicación.

***Dirección IP.*** Código numérico que identifica a un PC en Internet.

***Puerto.*** Dirección de memoria establecida para comunicaciones.

***Intranet.*** Red local de uso interno de las organizaciones que actúa de manera similar a Internet, ya que utiliza también los mismos protocolos de comunicaciones.

***Socket.*** Término inglés que puede referirse a la conexión en sí o al lugar físico que permite la conexión de un usuario, como por ejemplo un enchufe.

***Clave de sesión.*** Clave utilizada para cifrar o descifrar datos válida solamente en un periodo de tiempo determinado. También puede ser válida en un determinado número de comunicaciones.

***Clave simétrica.*** Clave utilizada para una comunicación simétrica, es decir, comunicación cifrada mediante dicha clave.

***Clave asimétrica.*** Claves publicas y/o privadas utilizadas en la comunicación asimétrica, es decir, la clave publica se utiliza para cifrar y la clave privada para descifrar.

***Interfaz.*** Diseño de pantalla que permite la interacción entre el usuario y el ordenador.

***UDP. User Datagram Protocol.*** Protocolo de comunicaciones utilizando datagramas.

***TCP. Transfer Control Protocol***. Protocolo orientado a conexión.

## 14 Bibliografía

**Java World.** *Sockets programming in Java: A tutorial.* En línea. Internet. Accesible en www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html. Consultado en 25/05/04.

**Java World.** *Introduction to AWT. A description of Java's user interface toolkit.* En línea. Internet. Accesible en www.javaworld.com/javaworld/jw-07-1996/jw-07-awt.html. Consultado en 25/05/04.

**Sun Microsystems** (1995-2003). *The Java Tutorial. Custom Networking. All about Sockets* . En línea. Internet. Accesible en http://java.sun.com/docs/books/tutorial/networking/sockets///index.html. Consultado en 25/05/04.

**Rodriguez Herrera, Daniel.** *Java en castellano. Introducción a Java.* En linea. Internet. Accesible en www.programacion.com/java/tutorial/intjava/. Consultado en 25/05/04.

**Travis, G**. *Using JSSE for secure socket communication.* En línea. Internet. Accesible en www.ibm.com/developerWorks. Consultado en 25/05/04.

**Miller R. Williams A**. *Java Sockets 101.* En línea. Internet. Accesible en www.ibm.com/developerWorks. Consultado en 25/05/04.

**Srinivas, R**. *Article: Using AES with Java Technology.* En línea. Internet. Accesible en http://java.sun.com/developer/technicalArticles/Security/AES/AES_v1.html. Consultado en 25/05/04.

**Todd E. Sundsted** (2001). *The practice of peer-to-peer computing: P2P meets SSL.* En línea. Internet. Accesible en http://www-106.ibm.com/developerworks/java/library/j-p2pssl/. Consultado en 25/05/04.

**Dillinger R**. (2001-2003) *Cyclopedia Cryptologia. Fair Coin Tosses and Die Rolls.* En línea. Internet. Accesible en http://www.disappearing-inc.com. Consultado en 25/05/04.

**Jonathan Knudsen**. *Java Cryptography. O'Reilly.*

**Ministerio de educación, cultura y deporte**. *Origen del juego de la OCA.*

**Ministerio de educación, cultura y deporte**. *Reglas del juego de la OCA.*

**Campderrich Falgueras, Benet**. *Enginyeria del programari I.* UOC

**Domingo Ferrer, Josep. Herrera Joancomartí, Jordi. X. Sánchez del Castillo, Ricardo**. *Criptografía.* UOC

# 15 Anexos

## 15.1 Paquete de distribución de la aplicación

La distribución de la aplicación se realiza utilizando un fichero comprimido en formato *zip* (programa compresor/descompresor **WinZip**). Una vez descomprimido el archivo, se creará una estructura de directorios como la que aparece a continuación:

- Directorio ***bin***: Dentro del directorio *bin* se encuentran los ficheros *.class* que son los ejecutables (bytecodes para la máquina virtual de java). También los ficheros de gráficos y los ficheros de almacenes de claves privadas/publicas así como el fichero generarKeystores.bat.

- Directorio ***doc***: Dentro del directorio *doc* esta la documentación de la utilización de las clases útil para el programador. El fichero *index.html* es el índice de contenidos.

- Directorio ***src***: Dentro del directorio *src* esta el código fuente de las clases, estructurado en diferentes ficheros *.java*, cada uno de los cuales es el código fuente de una clase en particular, que viene dada por el nombre del fichero.

## 15.2 Manual del usuario

En este apartado se explica como comenzar a utilizar la aplicación *El juego de la Oca*, explicando cuales son los argumentos de entrada de la aplicación, y como interactuar durante el desarrollo de la partida con la interfaz de usuario de la aplicación.

### 15.2.1 Requerimientos del sistema.

***NOTA: para la correcta ejecución de este proyecto, se recomienda el uso de la versión de Java 1.4 o superior, también llamada Java2, debido al uso de librerías criptográficas que esta distribución contiene (JSSE). El uso de otras versiones anteriores no garantiza que esta aplicación funcione correctamente.***

### 15.2.2 Instalación.

Para instalar la aplicación simplemente se ha de descomprimir el fichero *zip*, que crea una estructura de directorios, dentro de la cual está el programa principal de la aplicación.

### 15.2.3 Comenzar la aplicación: argumentos de entrada.

Para arrancar la aplicación necesitamos situarnos en la consola del sistema, para ello desde la consola de comandos tecleareremos la siguiente orden:

***: java Juego nombre_del_jugador***

Esta forma es la más básica de funcionamiento y se corresponde con la inicialización de una nueva partida como iniciador. Para ver todos los parámetros de entrada disponibles en el juego podemos teclear la siguiente orden:

***: java Juego***

De esta forma nos aparecerá una ayuda con todos los parámetros posibles de la aplicación. Estos son:

- *Nombre de jugador*: Nombre que se utilizará durante la ejecución de la partida. Ej: ***java Juego Daniel***

- *Dirección IP*: La dirección IP donde está el proceso iniciador de la partida. Se utiliza para conectarse con una partida ya iniciada. Se especifica entre paréntesis. Ej: ***java Juego Daniel (127.0.0.1)***

- *Puerto de conexión*: Puerto desde donde la partida a la que se quiere conectar escucha posibles jugadores entrantes. Se utiliza junto con la dirección IP para conectarse a una partida ya iniciada. Si no se especifica se utiliza el numero de puerto 9999. Se especifica entre corchetes. Ej: ***java Juego Daniel (127.0.0.1)* [*9998*]**

- *Puerto de conexión propio*: Es el puerto desde donde nosotros escucharemos jugadores entrantes. Si no se especifica, se utiliza el puerto por defecto que es el 9999. Ej: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000-**

- *Fichero de almacén de claves privadas*: Fichero que contiene nuestra clave privada. Se utiliza para el protocolo de negociación SSL. Si se especifica este parámetro se han de especificar también todos los siguientes que son tipo de almacén privado y password, y lo mismo para el almacén de claves publicas. Si no se especifica ninguno se utiliza por defecto el fichero *almacenprivado.jks*. Ej.: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000- *almacenprivado.jks jks password***

- *Tipo del fichero de almacén de claves privadas*: Tipo del almacén de claves especificado. Si no se especifica el tipo del fichero *almacenprivado.jks* es *jks*. Puede ser *p12*, *jks*, etc. Ej.: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000- *almacenprivado.jks jks password***

- *Password del fichero de almacén de claves privadas*: Password utilizado para proteger el almacén de claves privado. Ej: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000- *almacenprivado.jks jks password***

- *Fichero de almacén de claves publicas*: Fichero que contiene las claves publicas de todos los jugadores, ya que cada uno de los procesos que arranca cada jugador para jugar toman el papel de servidor. Se utiliza para el protocolo de negociación SSL. Si no se especifica ninguno se utiliza por defecto el fichero *almacenpublico.jks*. Ej.: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000- *almacenprivado.jks jks password almacenpublico.jks jks password***

- *Tipo del fichero de almacén de claves publicas*: Tipo del almacén de claves especificado. Si no se especifica el tipo del fichero *almacenpublico.jks* es *jks*. Puede ser *p12*, *jks*, etc. Ej.: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000- *almacenprivado.jks jks password almacenpublico.jks jks password***

- *Password del fichero de almacén de claves publicas*: Password utilizado para proteger el almacén de claves publicas. Ej.: ***java Juego Daniel (127.0.0.1)* [*9998*] –9000- *almacenprivado.jks jks password almacenpublico.jks jks password***

Si por el contrario, queremos conectarnos a una partida ya iniciada, tendremos que teclear la siguiente orden:

***: java Juego nombre_del_jugador (direccion_IP)* [puerto_remoto]**

La dirección IP y el puerto nos la ha de facilitar el iniciador de la partida. Si él no ha introducido ninguna y ha utilizado la forma básica de funcionamiento, solamente tendremos que teclear la orden siguiente:

***: java Juego nombre_del_jugador (direccion_IP)***

Ya que significará que el iniciador, al iniciar la partida, no ha especificado ningún puerto de conexión, si no que la aplicación utiliza el puerto por defecto.

Cuando necesitemos cambiar nuestro puerto de conexiones debido a que el de defecto (9999) lo tenemos ocupado con alguna aplicación de nuestro sistema operativo, tendremos que especificar uno diferente que no esté ocupado y anunciarlo a jugadores que quieran conectarse con nosotros, para ello teclearemos la orden:

***: java Juego nombre_del_jugador (direccion_IP)*** **[puerto_remoto] –puerto_propio-**

### 15.2.4 Jugando al juego de la Oca

Una vez dentro del programa, podemos ver cuatro zonas de pantalla. Una zona de mensajes, otra zona donde esta el tablero del juego y una tercera donde tenemos los botones que utilizaremos a lo largo del transcurso de la partida.

*Figura 32. Interfaz del juego de la Oca (2)*

- En el área de mensajes tenemos los mensajes del sistema que nos informan del estado de la aplicación y del estado de la partida. Desde aquí veremos si alguien se esta conectando y el estado de la conexión. Cuando comienza la partida, podemos ver el resultado de los dados, la casilla en la que estamos y además las bonificaciones o penalizaciones de la casilla en la que hemos caído. En el área de mensajes de error, veremos si alguien se ha desconectado o si la aplicación ha dado algún error, en cuyo caso es mejor reiniciar la partida para evitar posibles fallos de la aplicación.

- Por otro lado tenemos la lista de jugadores de la partida. Desde aquí veremos el numero de jugadores activos en la partida, así como su dirección IP y puerto utilizados. Además veremos el turno asignado a cada uno de ellos y el numero de casilla en el que estamos. El primer nombre de la lista siempre corresponde con nuestro jugador, sea iniciador de partida o no.

- En una tercera área de pantalla, podemos ver el tablero y la posición de la ficha que se nos ha sido asignada. Desde aquí controlamos en movimiento de todos los jugadores de una forma visual.

- Por último tenemos el área de botones con los cuales interactuaremos a lo largo de la partida. El botón de "comenzar partida" solamente lo tendremos activo si somos los iniciadores de la partida, ya que tenemos la decisión de comenzar la partida cuando queramos. El botón de "Lanzar dado" estará activo solamente cuando tengamos el turno de juego, para poder lanzar el dado. El botón de "Salir del juego" lo podemos pulsar siempre que queramos salir del juego, antes de finalizarlo. Por último y en esta misma zona de pantalla, podemos ver el gráfico del dado. Éste grafico se animará al lanzar el dado, simulando el comportamiento de un dado real al lanzarse.

Cuando se haya comenzado la partida y siempre que tengamos el turno de juego, podremos lanzar el dado. Si no tenemos el turno de juego, tendremos que limitarnos a ver como se mueven las fichas de los demás...

Al caer en una casilla especial, podremos obtener bonificaciones (oca, puente y dados), que resultarán en una avance de nuestras posiciones y en la posibilidad de volver a lanzar el dado. Si por el contrario, caemos en una casilla con penalización ( posada, pozo, laberinto, cárcel o muerte) perderemos algún turno en el mejor de los casos. ¡Si caemos en la muerte, tendremos que volver a la casilla numero 1!.

Cuando alguien llegue a la casilla 63 el juego se terminará y el jugador habrá ganado el juego. Entonces se tendrá que reiniciar la partida para volver a jugar. *¡Valor y a la Oca!...*

### 15.2.5 Errores más frecuentes

Quizás el error más frecuente sea al utilizar la aplicación en un mismo ordenador, con varios procesos comunicándose en local (esto se consigue, si el proceso que se quiere conectar al iniciador especifica como dirección IP de conexión, la dirección 127.0.0.1, que es la dirección local). En este caso el proceso que se quiere conectar con el iniciador, ha de especificar como parámetro de entrada, un puerto diferente que el asignado por defecto, ya que si ambos procesos utilizan el mismo puerto, se genera un error. Cuando esto ocurra, se ha de reiniciar tanto el proceso iniciador como los demás, ya que sino, en la negociación de conexión, la lista de jugadores activos que se le pasaría a un nuevo proceso sería errónea.

Otro error puede ser debido a la caducidad de los certificados que bienen con la aplicación. En ese caso se han de borrar los ficheros que se encuentran dentro del paquete de distribución de la aplicación, directorio bin, llamados *almacenprivado.jks* y *almacenpublico.jks* (ver explicación en el siguiente capitulo). A continuación se ha de ejecutar el fichero por lotes *generarKeystores.bat* que generará de nuevo los almacenes con certificados actualizados. Este proceso se ha de llevar a cabo en todas las distribuciones que participen en el juego.

## 15.3 El fichero generarKeystores.bat

Para la generación de claves y su manipulación se ha utilizado el programa estándar *keytool* que se incluye con el paquete JSSE en el JDK 1.4. Keytool puede ser utilizado para multitud de objetivos. En nuestro proyecto, se ha utilizado para crear parejas de claves privadas/publicas, y para extraer los certificados de la clave publica desde las parejas de claves creadas y ponerlas en sus propios ficheros: *almacenprivado.jks* y *almacenpublico.jks*.

El almacén almacenprivado.jks aloja las claves privadas/publicas y certificados de la parte cliente y de la parte servidor.

Se alojan en este mismo fichero debido a que la aplicación hace las funciones tanto de cliente como de servidor.

Así pues cuando se utiliza la parte cliente se utilizará la clave publica del servidor para comprobar su certificado y su propia clave privada que se encuentra en el fichero almacenprivado.jks. Cuando se utilice la parte servidor, en teoría y siempre y cuando se activa la opción de autenticación de cliente, en el fichero *almacenpublico.jks* estará su certificado y en el fichero *almacenprivado.jks* su clave privada.

Así pues, nos quedarán dos ficheros:

- *almacenprivado.jks* : contiene las claves privadas de la parte cliente y servidor. Se utilizan cuando se haga el papel de servidor y de cliente ya que necesitan para negociar sus claves privadas.

- *almacenpublico.jks* : contiene las claves publicas del servidor y del cliente. Se utilizan para verificar los certificados cuando se requiera la autentificación por parte del servidor y si se utilizara, la autentificación por parte del servidor del cliente.

### 15.3.1 Generando una pareja de claves

El siguiente comando de **keytool** se utiliza para generar un nuevo par de claves publica/privada:

*Keytool –genkey –keystore [nombre_de_fichero*

Cuando se ejecuta este comando, se hacen una serie de preguntas, las cuales preguntan información de la autoridad de certificación (nombre, organización y otras). La información proveída se utiliza para crear un certificado auto-firmado que se asocia con la clave publica y testifica la autenticidad de la asociación. Después se pregunta por un password que sirve para asegurar el fichero y opcionalmente claves para proteger las claves que se están generando.

El comando completo que crea el par de claves y provee toda la información en un único comando tiene el aspecto siguiente:

*keytool -genkey -alias jugadorcliente -keystore almacenprivado.jks -storetype JKS -keyalg rsa -dname "CN=Jugador Cliente, OU=TFC, O=TFC, L=Barcelona, C=SP" -storepass password -keypass password*

| Opción | ¿qué significa? |
|---|---|
| -genkey | Le dice a **keytool** que genere una pareja de claves |
| -alias jugadorcliente | Identifica el par de claves dentro del almacén |
| -keystore almacenprovado.jks | Utiliza el fichero almacenprivado.jks como el almacén |
| -storetype JKS | Declara el tipo de almacén. JKS viene por defecto |
| -keyalg rsa | Declara el algoritmo a utilizar. Utilizaremos encriptación RSA |
| -dname “CN=Jugador... | Información sobre la entidad poseedora de las claves |
| -storepass password | Establece el password del almacén |
| -keypass password | Establece el password de las claves |

*Tabla 4. opciones del software criptográfico keytool*

### 15.3.2 Par de claves cliente/servidor

El primer paso en la preparación de nuestro algoritmo es generar un par de claves publica/privada para el cliente y otra para el servidor que utilizará la aplicación.

Parte cliente:

*keytool -genkey -alias jugadorcliente -keystore almacenprivado.jks -storetype JKS -keyalg rsa -dname "CN=Jugador Cliente, OU=TFC, O=TFC, L=Barcelona, C=SP" -storepass password -keypass password*

Parte servidor:

*keytool -genkey -alias jugadorservidor -keystore almacenprivado.jks -storetype JKS -keyalg rsa -dname "CN=Jugador Servidor, OU=TFC, O=TFC, L=Barcelona, C=SP" -storepass password -keypass password*

### 15.3.3 Extraer las claves publicas

El próximo paso es extraer las claves publicas para insertarlas en un mismo almacén que se instalarán junto con la aplicación. La parte cliente de la aplicación utilizará la clave publica del servidor y la parte servidor la del cliente.

Pare ello extraemos las claves publicas del almacén creado *almacenprivado.jks* y los insertamos en ficheros temporales. A continuación insertamos en el almacén publico las claves publicas extraídas:

Exportación de claves publicas:

*keytool -export -alias jugadorcliente -keystore almacenprivado.jks -file temp1.key -storepass password*
*keytool -export -alias jugadorservidor -keystore almacenprivado.jks -file temp2.key -storepass password*

Importación de las claves extraídas:

*keytool -import -noprompt -alias trustcliente -keystore almacenpublico.jks -file temp1.key -storepass password*
*keytool -import -noprompt -alias trustservidor -keystore almacenpublico.jks -file temp2.key -storepass password*

## 15.4 Los ficheros almacenes de claves: *almacenprivado.jks* y *almacenpublico.jks*

Estos ficheros se utilizan para autentificar a las partes y para poder realizar la negociación en la conexión SSL.

El almacén almacenprivado.jks aloja las claves privadas/publicas y certificados de la parte cliente y de la parte servidor. Se alojan en este mismo fichero debido a que la aplicación hace las funciones tanto de cliente como de servidor.

Así pues cuando se utiliza la parte cliente se utilizará la clave publica del servidor para comprobar su certificado y su propia clave privada que se encuentra en el fichero almacenprivado.jks. Cuando se utilice la parte servidor, en teoría y siempre y cuando se activa la opción de autenticación de cliente, en el fichero *almacenpublico.jks* estará su certificado y en el fichero *almacenprivado.jks* su clave privada.

Así pues, nos quedarán dos ficheros:

- *almacenprivado.jks* : contiene las claves privadas de la parte cliente y de la parte servidor. Se utilizan cuando se toma el papel de servidor y de cliente ya que se necesitan para la negociación previa a la conexión.

- *almacenpublico.jks* : contiene las claves publicas del servidor y del cliente. Se utilizan para verificar los certificados cuando se requiera la autentificación por parte del servidor y si se utilizara, la autentificación por parte del servidor del cliente.

## 15.5 Los ficheros gráficos. El grafico del tablero y de las fichas.

Para la representación del tablero del juego y de las fichas se han utilizado varios ficheros gráficos de formato PNG.

El primero representa el tablero del juego de la oca, y se llama Tablero.png.

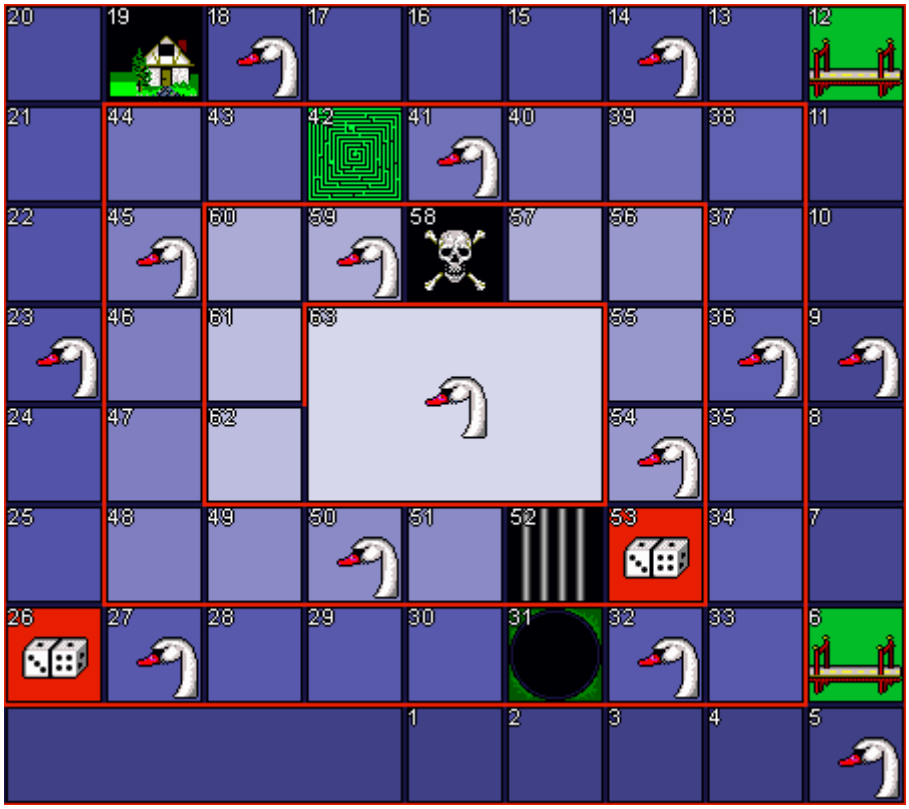


*Figura 33.Grafico Tablero.png*

Los ficheros de gráficos que representan las fichas del juego son cuatro. Cada uno es de un color diferente y se asigna rotativamente a los jugadores enrolados en la partida.

*Figura 34. Gráficos de las fichas. Ficha1.png, ficha2.png, ficha3.png y ficha4.png*

Para representar el dado y realizar la simulación del lanzamiento se han utilizado también unos gráficos en formato PNG, estos son:

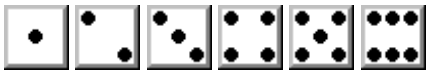

*Figura 35. Gráficos del dado. Dado1.png, dado2.png, dado3.png y dado4.png*

## 15.6 código fuente comentado

### 15.6.1 Clase Base64.

```
/**
 * Freeware from:
 * Roedy Green
 * Canadian Mind Products
 * #327 - 964 Heywood Avenue
 * Victoria, BC Canada V8V 2Y5
 * tel:(250) 361-9093
 * mailto:roedyg@mindprod.com
 */

/**
 * Encode arbitrary binary into printable ASCII using BASE64 encoding.
 * very loosely based on the Base64 Reader by
 * Dr. Mark Thornton
 * Optrak Distribution Software Ltd.
 * http://www.optrak.co.uk
 * and Kevin Kelley's  http://www.ruralnet.net/~kelley/java/Base64.java
 *
 * Base64 is a way of encoding 8-bit characters using only ASCII printable
 * characters similar to UUENCODE.  UUENCODE includes a filename where
 * BASE64 does not.
 * The spec is described in RFC 2045.  Base64 is a scheme where
 * 3 bytes are concatenated, then split to form 4 groups of 6-bits each; and
 * each 6-bits gets translated to an encoded printable ASCII character, via a
 * table lookup.  An encoded string is therefore longer than the original by
 * about 1/3.  The "=" character is used to pad the end.  Base64 is used,
 * among other things, to encode the user:password string in an
 * Authorization: header for HTTP.  Don't confuse Base64 with
 * x-www-form-urlencoded which is handled by
 * Java.net.URLEncoder.encode/decode
 * If you don't like this code, there is another implementation
 * at http://www.ruffboy.com/download.htm
 * Sun has an undocumented method called sun.misc.Base64Encoder.encode.
 * You could use hex, simpler to code, but not as compact.
 *
 * If you wanted to encode a giant file, you could do it in large chunks that
 * are even multiples of 3 bytes, except for the last chunk, and append
 * the outputs.
 *
 * To encode a string, rather than binary data java.net.URLEncoder may
 * be better. See
 * printable characters in the Java glossary for a discussion of
 * the differences.
 *
 * version 1.4 2002 February 15  -- correct bugs with uneven line lengths,
 *                                  allow you to configure line separator.
 *                                  now need Base64 object and instance
 * methods.
 *                                  new mailing address.
 * version 1.3 2000 September 12 -- fix problems with estimating output length
 * in encode
 * version 1.2 2000 September 09 -- now handles decode as well.
 * version 1.1 1999 December 04 -- more symmetrical encoding algorithm.
 *                                 more accurate StringBuffer allocation size.
 * version 1.0 1999 December 03 -- posted in comp.lang.java.programmer.
 * Futures Streams or files.
 */

public class Base64
   {

   /**
    * how we separate lines, e.g. \n, \r\n, \r etc.
```

```
 */
private String lineSeparator = System.getProperty( "line.separator" );

/**
 * max chars per line, excluding lineSeparator.  A multiple of 4.
 */
private  int lineLength = 72;

/* constructor */
public Base64()
   {
   }

/**
 * Encode an arbitrary array of bytes as Base64 printable ASCII.
 * It will be broken into lines of 72 chars each.  The last line is not
 * terminated with a line separator.
 * The output will always have an even multiple of data characters,
 * exclusive of \n.  It is padded out with =.
 */
public  String encode(byte[] b)
   {
   // Each group or partial group of 3 bytes becomes four chars
   // covered quotient
   int outputLength = ((b.length + 2) / 3) * 4;

   // account for trailing newlines, on all but the very last line
   if ( lineLength != 0 )
      {
      int lines =  ( outputLength + lineLength -1 ) / lineLength - 1;
      if ( lines > 0 )
         {

         outputLength += lines  * lineSeparator.length();
         }
      }

   // must be local for recursion to work.
   StringBuffer sb = new StringBuffer( outputLength );

   // must be local for recursion to work.
   int linePos = 0;

   // first deal with even multiples of 3 bytes.
   int len = (b.length / 3) * 3;
   int leftover = b.length - len;
   for ( int i=0; i<len; i+=3 )
      {
      // Start a new line if next 4 chars won't fit on the current line
      // We can't encapsulete the following code since the variable need to
      // be local to this incarnation of encode.
      linePos += 4;
      if ( linePos > lineLength )
         {
         if ( lineLength != 0 )
            {
            sb.append(lineSeparator);
            }
         linePos = 4;
         }

      // get next three bytes in unsigned form lined up,
      // in big-endian order
      int combined = b[i+0] & 0xff;
      combined <<= 8;
      combined |= b[i+1] & 0xff;
      combined <<= 8;
      combined |= b[i+2] & 0xff;
```

```
         // break those 24 bits into a 4 groups of 6 bits,
         // working LSB to MSB.
         int c3 = combined & 0x3f;
         combined >>>= 6;
         int c2 = combined & 0x3f;
         combined >>>= 6;
         int c1 = combined & 0x3f;
         combined >>>= 6;
         int c0 = combined & 0x3f;

         // Translate into the equivalent alpha character
         // emitting them in big-endian order.
         sb.append( valueToChar[c0]);
         sb.append( valueToChar[c1]);
         sb.append( valueToChar[c2]);
         sb.append( valueToChar[c3]);
         }

      // deal with leftover bytes
      switch ( leftover )
         {
         case 0:
         default:
            // nothing to do
            break;

         case 1:
            // One leftover byte generates xx==
            // Start a new line if next 4 chars won't fit on the current line
            linePos += 4;
            if ( linePos > lineLength )
               {

               if ( lineLength != 0 )
                  {
                  sb.append(lineSeparator);
                  }
               linePos = 4;
               }

            // Handle this recursively with a faked complete triple.
            // Throw away last two chars and replace with ==
            sb.append(encode(new byte[] {b[len], 0, 0}
                             ).substring(0,2));
            sb.append("==");
            break;

         case 2:
            // Two leftover bytes generates xxx=
            // Start a new line if next 4 chars won't fit on the current line
            linePos += 4;
            if ( linePos > lineLength )
               {
               if ( lineLength != 0 )
                  {
                  sb.append(lineSeparator);
                  }
               linePos = 4;
               }
            // Handle this recursively with a faked complete triple.
            // Throw away last char and replace with =
            sb.append(encode(new byte[] {b[len], b[len+1], 0}
                             ).substring(0,3));
            sb.append("=");
            break;

         } // end switch;
```

```
      if ( outputLength != sb.length() )
         {
         System.out.println("oops: minor program flaw: output length mis-
estimated");
         System.out.println("estimate:" + outputLength);
         System.out.println("actual:" + sb.length());
         }
      return sb.toString();
      }// end encode

   /**
    * decode a well-formed complete Base64 string back into an array of bytes.
    * It must have an even multiple of 4 data characters (not counting \n),
    * padded out with = as needed.
    */
   public  byte[] decode( String s)
      {

      // estimate worst case size of output array, no embedded newlines.
      byte[] b = new byte[(s.length() / 4) * 3];

      // tracks where we are in a cycle of 4 input chars.
      int cycle = 0;

      // where we combine 4 groups of 6 bits and take apart as 3 groups of 8.
      int combined = 0;

      // how many bytes we have prepared.
      int j = 0;
      // will be an even multiple of 4 chars, plus some embedded \n
      int len = s.length();
      int dummies = 0;
      for ( int i=0; i<len; i++ )
         {

         int c = s.charAt(i);
         int value  = (c <= 255) ? charToValue[c] : IGNORE;
         // there are two magic values PAD (=) and IGNORE.
         switch ( value )
            {
            case IGNORE:
               // e.g. \n, just ignore it.
               break;

            case PAD:
               value = 0;
               dummies++;
               // fallthrough
            default:
               /* regular value character */
               switch ( cycle )
                  {
                  case 0:
                     combined = value;
                     cycle = 1;
                     break;

                  case 1:
                     combined <<= 6;
                     combined |= value;
                     cycle = 2;
                     break;

                  case 2:
                     combined <<= 6;
                     combined |= value;
                     cycle = 3;
```

```
                        break;

                    case 3:
                        combined <<= 6;
                        combined |= value;
                        // we have just completed a cycle of 4 chars.
                        // the four 6-bit values are in combined in big-endian
order
                        // peel them off 8 bits at a time working lsb to msb
                        // to get our original 3 8-bit bytes back

                        b[j+2] = (byte)combined;
                        combined >>>= 8;
                        b[j+1] = (byte)combined;
                        combined >>>= 8;
                        b[j] = (byte)combined;
                        j += 3;
                        cycle = 0;
                        break;
                    }
                break;
            }
        } // end for
      if ( cycle != 0 )
        {
        throw new ArrayIndexOutOfBoundsException ("Input to decode not an
even multiple of 4 characters; pad with =.");
        }
      j -= dummies;
      if ( b.length != j )
        {
        byte[] b2 = new byte[j];
        System.arraycopy(b, 0, b2, 0, j);
        b = b2;
        }
      return b;

      }// end decode

   /**
    * determines how long the lines are that are generated by encode.
    * Ignored by decode.
    * @param length 0 means no newlines inserted. Must be a multiple of 4.
    */
   public  void setLineLength(int length)
      {
      this.lineLength = (length/4) * 4;
      }

   /**
    * How lines are separated.
    * Ignored by decode.
    * @param lineSeparator may be "" but not null.
    * Usually contains only a combination of chars \n and \r.
    * Could be any chars not in set A-Z a-z 0-9 + /.
    */
   public  void setLineSeparator(String lineSeparator)
      {
      this.lineSeparator = lineSeparator;
      }

   /**
    * letter of the alphabet used to encode binary values 0..63
    */
   static final char[] valueToChar = new char[64];

   /**
    * binary value encoded by a given letter of the alphabet 0..63
```

```
 */
static final int[] charToValue = new int[256];

/**
 * Marker value for chars we just ignore, e.g. \n \r high ascii
 */
static final int IGNORE = -1;

/**
 * Marker for = trailing pad
 */
static final int PAD = -2;

static  /* initialise valueToChar and charToValue tables */
{
   // build translate valueToChar table only once.
   // 0..25 -> 'A'..'Z'
   for ( int i=0; i<=25; i++ )
      valueToChar[i] = (char)('A'+i);
   // 26..51 -> 'a'..'z'
   for ( int i=0; i<=25; i++ )
      valueToChar[i+26] = (char)('a'+i);
   // 52..61 -> '0'..'9'
   for ( int i=0; i<=9; i++ )
      valueToChar[i+52] = (char)('0'+i);
   valueToChar[62] = '+';
   valueToChar[63] = '/';

   // build translate charToValue table only once.
   for ( int i=0; i<256; i++ )
      {
      charToValue[i] = IGNORE;  // default is to ignore
      }

   for ( int i=0; i<64; i++ )
      {
      charToValue[valueToChar[i]] = i;
      }

   charToValue['='] = PAD;
}

/**
 * used to disable test driver
 */
private static final boolean debug = false;

/**
 * debug display array
 */
public static void show (byte[] b)
   {
   for ( int i=0; i<b.length; i++ )
      {
      System.out.print( Integer.toHexString(b[i] & 0xff) + " ");
      }
   System.out.println();
   }

/**
 * debug display array
 */
public static void display (byte[] b)
   {
   for ( int i=0; i<b.length; i++ )
      {
      System.out.print( (char)b[i]);
      }
```

```
      System.out.println();
      }

   /**
     * test driver
    */
   public static void main(String[] args)
      {
      if ( debug )
         {
         byte[] a = { (byte)0xfc, (byte)0x0f, (byte)0xc0};
         byte[] b = { (byte)0x03, (byte)0xf0, (byte)0x3f};
         byte[] c = { (byte)0x00, (byte)0x00, (byte)0x00};
         byte[] d = { (byte)0xff, (byte)0xff, (byte)0xff};
         byte[] e = { (byte)0xfc, (byte)0x0f, (byte)0xc0, (byte)1};
         byte[] f = { (byte)0xfc, (byte)0x0f, (byte)0xc0, (byte)1, (byte)2};
         byte[] g = { (byte)0xfc, (byte)0x0f, (byte)0xc0, (byte)1, (byte)2,
(byte)3};
         byte[] h = "AAAAAAAAAAB".getBytes();

         show(a);
         show(b);
         show(c);
         show(d);
         show(e);
         show(f);
         show(g);
         show(h);
         Base64 b64 = new Base64();
         show(b64.decode(b64.encode(a)));
         show(b64.decode(b64.encode(b)));
         show(b64.decode(b64.encode(c)));
         show(b64.decode(b64.encode(d)));
         show(b64.decode(b64.encode(e)));
         show(b64.decode(b64.encode(f)));
         show(b64.decode(b64.encode(g)));
         show(b64.decode(b64.encode(h)));
         b64.setLineLength(8);
         show((b64.encode(h)).getBytes());
         }
      }// end main

   } // end Base64
```

### 15.6.2 Clase Dado.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

/**
 * Clase que representa un dado, que al lanzarse puede devolver el resultado
de dos
 * formas diferentes, el valor en claro del lanzamiento y el valor cifrado.
 * la encriptación la resuelve aplicando metodos de clave simetrica DES.
 * La clave utilizada también se puede pedir mediante metodos de lectura.
 */
public class Dado
{
   //Propiedad que representa el valor mayor que puede obtenerse al tirar el
dado
   private int    valorMayor;
   //Representa el valor del dado encriptado
   private byte[] valorCifrado;
   //Representa el valor de la clave utilizada para encriptar (clave DES)
   private byte[] valorClave;
   //Representa el valor en claro del lanzamiento
   private int    valorEnClaro;
   //clave generada mediante el metodo DES.
   private SecretKey clave;
   //Generador de claves, se utiliza el protocolo DES
   private KeyGenerator generadorClave;
   //Variable de entorno para la generacion de la clave de sesion
   private Cipher cipher;

   /**
    * Metodo constructor.
    * construye un dado, inicializando la generación de claves para posterior
    * utilización.
    * Se le pasa un parametro que represenra el mayor numero posible a
alcanzar
    * al lanzar el dado, es decir, genera un numero aleatorio entre 0 y ese
numero
    * @param int numero mayor que puede obtener el lanzamiento del dado
    */
   public Dado( int pValorMayor ) throws Exception
   {
      //Asignacion de variables de inicialización
      valorMayor   = pValorMayor;
      //Valores de clave y valor cifrado a null
      valorCifrado = null;
      valorClave   = null;
      //inicializacion del generador de claves (tipo DES)
      generadorClave = KeyGenerator.getInstance("DES");
      cipher = Cipher.getInstance( "DES" );
   }

   /**
    * Metodo que lanza un dado y devuelve el resultado en claro.
    * obtiene los valores de la clave utilizada para cifrar, y el
    * resultado de cifrar el valor en clave utilizando dicha clave
    * @return int valor del lanzamiento del dado
    */
   public int lanzar() throws Exception
   {
```

```
        //genera una clave DES
        clave = generadorClave.generateKey();
        //Obtiene el array de bytes representativo de la clave generada
        valorClave = clave.getEncoded();
        //lanza el dado generando un valor aleatorio seguro entre 0 y valorMayor
        valorEnClaro = new SecureRandom().nextInt( valorMayor );
        //Genera un secretkeyspec que utiliza un array de bytes donde esta la
        //clave a utilizar para codificar y/o descodificar (tipo DES)
        SecretKeySpec cSpec = new SecretKeySpec( valorClave, "DES" );
        //inicializacion para la encriptacion (utilizando la clave anterior)
        cipher.init(Cipher.ENCRYPT_MODE, (SecretKey) cSpec);
        //encripta el resultado del valor del dado en claro
        valorCifrado = cipher.doFinal( String.valueOf(valorEnClaro).getBytes()
);
        //devuelve el valor del dado en claro
        return valorEnClaro;
    }

    /**
     * Metodo inicializador.
     * Pone a null la variable que contendra el valor del dado cifrado.
     */
    public void resetValorCifrado()
    {
        valorCifrado = null;
    }

    /**
     * Metodo escritura.
     * Establece el valor de la variable del valor cifrado del lanzamiento del
dado
     * @param byte[] valor del dado encriptado
     */
    public void setValorCifrado( byte[] pValor )
    {
        valorCifrado = pValor;
    }

    /**
     * Metodo lectura.
     * Devuelve si la variable que ha de contener el valor del lanzamiento del
dado
     * encriptado se ha establecido o no (es decir, si se ha lanzado el dado).
     * @return boolean cierto si el valor cifrado del dado se puede obtener.
     */
    public boolean isValorCifrado()
    {
        return ((valorCifrado == null) ? false : true);
    }

    /**
     * Metodo lectura.
     * Obtiene el valor del resultado del dado cifrado.
     * @return byte[] el valor del dado cifrado.
     */
    public byte[] getValorCifrado()
    {
        return valorCifrado;
    }

    /**
     * Metodo inicializador.
     * pone a null el valor de la clave utilizada para cifrar/descifrar el
valor
     * del dado.
     */
    public void resetValorClave()
    {
```

```
        valorClave = null;
    }

    /**
     * Metodo escritura.
     * Establece el valor de la clave a utilizar para cifrar/descifrar el dado.
     * @param byte[] valor de la clave (tiene que ser tipo DES)
     */
    public void setValorClave( byte[] pValor )
    {
        valorClave = pValor;
    }

    /**
     * Metodo lectura.
     * Obtiene si el valor de la clave esta presente.
     * @return  boolean cierto si la clave se ha generado o establecido por
otra via
     */
    public boolean isValorClave()
    {
        return ((valorClave == null) ? false : true);
    }

    /**
     * Metodo lectura.
     * Obtiene el valor de la clave utilizada para cifrar/descifrar el dado
     * @return byte[] el valor de la clave DES
     */
    public byte[] getValorClave()
    {
        return valorClave;
    }

    /**
     * Metodo lectura.
     * Obtiene el valor del dado en claro.
     * @return int valor del lanzamiento del dado.
     */
    public int getValorEnClaro()
    {
        return valorEnClaro;
    }

    /**
     * Metodo escritura.
     * Descifra el valor del dado con la clave DES establecida.
     * @return int valor en claro de descifrar el valor cifrado con la clave
DES
     */
    public int descifraDado() throws Exception
    {
        //Genera un secretkeyspec que utiliza un array de bytes donde esta la
        //clave a utilizar para codificar y/o descodificar (tipo DES)
        SecretKeySpec cSpec = new SecretKeySpec( valorClave, "DES" );
        //inicializacion para la desencriptacion (utilizando la clave anterior)
        cipher.init(Cipher.DECRYPT_MODE, (SecretKey) cSpec);
        //descifra el valor cifrado con la clave
        valorEnClaro = Integer.parseInt( new String( cipher.doFinal(
valorCifrado ) ) );
        //Devuelve el valor descifrado.
        return valorEnClaro;
    }
}
```

### 15.6.3 Clase ImagenDado.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.awt.*;

 /**
  * Clase que representa el grafico de un dado compuesto por seis imagenes
(dado a uno, dado a dos, etc)
  * es una extension de la clase Canvas que reimplementa algunos de sus
metodos para dibujar en pantalla
  * el dado.
  * Un objeto de esta clase se puede añadir a un Layout de un panel, ya que es
una subclase de la clase
  * Canvas. Mirar la clase Interfaz.
  */
public class ImagenDado extends Canvas
{
   //Array de imagenes que representan los seis graficos del dado
   private Image frame[] = new Image[6];
   //Mediatracker utilizado para añadir graficos al Canvas
   private MediaTracker m;
   //Propiedad que representa el "frame" del grafico del dado actual, es
decir, es un indice
   //indicativo del grafico que se ha de dibujar en cada llamada al método
paint().
   private int frameActual;

   /**
    * Metodo Constructor.
    * Inicializa el MediaTracker para añadir imagenes al Canvas. Carga las
imagenes desde disco
    * en formato PNG, llamadas "dado1.png", "dado2.png" .. "dado6.png".
    * Inicializa el indice de frames a 1, que representa el primer grafico del
dado (el uno)
    * Establece el tamaño del grafico, que es igual al tamaño del primer
grafico (del primero y de
    * cualquiera de ellos, ya que todos tienen el mismo tamaño).
    */
   public ImagenDado()
   {
      //Inicializa un MediaTracker y el ToolKit utilizado para cargar imagenes
desde disco
      m = new MediaTracker( this );
      java.awt.Toolkit tk = java.awt.Toolkit.getDefaultToolkit();

      //Añadimos al mediatracker las seis imagenes del dado
      for( int i = 1; i <= 6; i++ )
      {
         //Se utiliza el metodo getImage de la clase Toolkit
         frame[i-1] = tk.getImage( "dado"+i+".png" );
         //Añade mediante el metodo addImage de MediaTracker el grafico al
Canvas, les asigna
         //el identificador 0 a todos, en este caso nos es indiferente este
dato.
         m.addImage( frame[i-1], 0 );
      }

      //Espera a que se carguen todas las imagenes, identificadas todas con 0.
      try {
         m.waitForID( 0 );
      } catch( Exception e ){}

      //Establece el frame actual a cero (a la primera imagen)
```

```
        frameActual = 0;

        //Establece el tamaño del Canvas.
        setSize( frame[frameActual].getWidth(null),
frame[frameActual].getHeight(null) );
    }

    /**
     * Metodo de escritura.
     * Establece el frame actual, es decir, el índice representante de la
imagen a dibujar.
     * @param int numero de 1 a 6 indicador de la imagen actual
     */
    public void setFrame( int pFrame )
    {
        //Si el frame que se quiere establecer esta entre uno y 6
        //en caso contrario, dibujar el primer grafico
        if( (pFrame > 0) && (pFrame <= 6) ) frameActual = pFrame-1;
        else frameActual = 0;
    }

    /**
     * Metodo de escritura.
     * Simula el lanzamiento del dado graficamente, disminuyendo la velocidad
del dado
     * a medida que avanza hacia el valor final indicado por parametro.
     * @param int valor final que tiene el dado (1 a 6)
     */
    public void simulaLanzamiento( int valorFinal ) throws Exception
    {
        //Contador del bucle de simulacion
        int contador = 1;
        //para el valor final valorFinal, con 20 iteraciones se ha de comenzar
        //con el valorFinal-2.
        int dado = valorFinal-2;
        //Hace 20 repeticiones, cambiando la imagen del dado haciendo la
simulacion
        //de movimiento del dado
        while( contador <= 20 )
        {
            //Establece el "frame" del dado, calculado a partir de la variable
dado.
            //se hace una operacion modulo 6, para que el valor siempre este
entre 1 y 6.
            setFrame( (dado%6)+1 );
            //Redibuja el dado
            repaint();
            //Pausa este thread un tiempo dependiendo de la iteracion en la que
este
            //el bucle
            Thread.sleep( 10*contador );
            //suma 1 al contador y al dado.
            contador++; dado++;
        }
    }

    /**
     * Metodo de actualizacion.
     * Este metodo se llama cada vez que se quiere redibujar la pantalla (una
llamada al metodo
     * repaint()).
     * El metodo original del Canvas, primero borra la pantalla, pero para
evitar el "flickering", o
     * el parpadeo de imagen, reimplementamos este metodo para evitarlo,
simplemente llamando al metodo
     * de dibujado, sin borrar antes la pantalla.
     * @param Graphics la "pantalla" donde se quiere actualizar.
     */
```

```
    public void update( Graphics g )
    {
        paint(g);
    }

    /**
     * Metodo de dibujado.
     * Este metodo dibuja el grafico, indicado por la propiedad de frame
actual, del dado.
     * es una reimplementacion del mismo método de la clase Canvas. Solamente
nos interesa dibujar
     * el dado.
     * @param Graphics pantalla donde se quiere dibujar
     */
    public void paint( Graphics g )
    {
        //Dibuja la imagen del dado indicada por la propiedad frameActual, en la
posicion x=0, y=0.
        g.drawImage( frame[frameActual], 0, 0, this );
    }
}
```

### 15.6.4 Clase ImagenFicha.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.awt.*;

 /**
  * Esta clase representa la posicion en el tablero y la imagen de una ficha
utilizada
  * por un jugador.
  */
public class ImagenFicha
{
   //Imagen de la ficha
   private Image i;
   //identificacion de ficha, asocia una ficha a un jugador
   private int identificador;
   //Posicion x e y de la ficha dentro del panel de imagen (Clase ImagePanel)
   private int x, y;
   //casilla donde esta la ficha
   private int casilla;

   /**
    * Constructor.
    * Carga el grafico indicado por parametro y establece las coordenadas de
la ficha.
    * @param String nombre del fichero que contiene el grafico representativo
de la ficha, en formato PNG, JPEG o GIF
    * @param int posicion X de la ficha
    * @param int posicion Y de la ficha
    */
   public ImagenFicha( String fichero, int posicionX, int posicionY, int
pIdentificador )
   {
      //Inicializa un objeto Toolkit, para cargar imagenes de disco
      java.awt.Toolkit tk = java.awt.Toolkit.getDefaultToolkit();
      //Carga el grafico indicado por parametro
      i = tk.getImage(fichero);
      //Establece las coordenadas de la ficha
      x = posicionX;
      y = posicionY;
      //Establece el identificador de la ficha
      //que es igual al turno del jugador que la mueve por el tablero.
      identificador = pIdentificador;
      //La ficha esta en la primera casilla del tablero
      casilla = 0;
   }

   /**
    * Metodo de lectura.
    * Obtiene el grafico de la ficha como un Objeto Image
    * @return Image la imagen asignada al gráfico
    */
   public Image getImagen()
   {
      return i;
   }

   /**
    * Metodo de lectura.
    * Obtiene la coordenada X de la ficha.
    * @return int posicion X de la ficha
    */
   public int getX()
```

```
    {
        return x;
    }

    /**
     * Metodo de lectura.
     * Obtiene la coordenada Y de la ficha.
     * @return int posicion Y de la ficha
     */
    public int getY()
    {
        return y;
    }

    /**
     * Metodo de escritura.
     * Establece las coordenadas X e Y de la ficha.
     * @param int posicion X de la ficha a establecer
     * @param int posicion Y de la ficha a establecer
     */
    private void setXY( int pX, int pY )
    {
        x = pX;
        y = pY;
    }

    /**
     * Metodo de lectura.
     * Obtiene el identificador de esta ficha.
     * @return int identificador de la ficha.
     */
    public int getIdentificador()
    {
        return identificador;
    }

    /**
     * Metodo de lectura.
     * Obtiene el numero de casilla donde esta la ficha.
     * @return int numero de casilla donde esta la ficha.
     */
    public int getCasilla()
    {
        return casilla;
    }

    /**
     * Metodo de escritura.
     * mueve la ficha graficamente, es decir, cambia sus coordenadas de
pantalla.
     * estas coordenadas, estan en relación al tablero.
     * @param int casilla a la que se quiere mover.
     */
    public void setCasilla( int pCasilla )
    {
        casilla = pCasilla;
        switch( casilla )
        {
            case  1: setXY(217, 367); break;
            case  2: setXY(267, 367); break;
            case  3: setXY(317, 367); break;
            case  4: setXY(367, 367); break;
            case  5: setXY(417, 367); break;
            case  6: setXY(417, 317); break;
            case  7: setXY(417, 267); break;
            case  8: setXY(417, 217); break;
            case  9: setXY(417, 167); break;
            case 10: setXY(417, 117); break;
```

```
            case 11: setXY(417,  67); break;
            case 12: setXY(417,  17); break;
            case 13: setXY(367,  17); break;
            case 14: setXY(317,  17); break;
            case 15: setXY(267,  17); break;
            case 16: setXY(217,  17); break;
            case 17: setXY(167,  17); break;
            case 18: setXY(117,  17); break;
            case 19: setXY( 67,  17); break;
            case 20: setXY( 17,  17); break;
            case 21: setXY( 17,  67); break;
            case 22: setXY( 17, 117); break;
            case 23: setXY( 17, 167); break;
            case 24: setXY( 17, 217); break;
            case 25: setXY( 17, 267); break;
            case 26: setXY( 17, 317); break;
            case 27: setXY( 67, 317); break;
            case 28: setXY(117, 317); break;
            case 29: setXY(167, 317); break;
            case 30: setXY(217, 317); break;
            case 31: setXY(267, 317); break;
            case 32: setXY(317, 317); break;
            case 33: setXY(367, 317); break;
            case 34: setXY(367, 267); break;
            case 35: setXY(367, 217); break;
            case 36: setXY(367, 167); break;
            case 37: setXY(367, 117); break;
            case 38: setXY(367,  67); break;
            case 39: setXY(317,  67); break;
            case 40: setXY(267,  67); break;
            case 41: setXY(217,  67); break;
            case 42: setXY(167,  67); break;
            case 43: setXY(117,  67); break;
            case 44: setXY( 67,  67); break;
            case 45: setXY( 67, 117); break;
            case 46: setXY( 67, 167); break;
            case 47: setXY( 67, 217); break;
            case 48: setXY( 67, 267); break;
            case 49: setXY(117, 267); break;
            case 50: setXY(167, 267); break;
            case 51: setXY(217, 267); break;
            case 52: setXY(267, 267); break;
            case 53: setXY(317, 267); break;
            case 54: setXY(317, 217); break;
            case 55: setXY(317, 167); break;
            case 56: setXY(317, 117); break;
            case 57: setXY(267, 117); break;
            case 58: setXY(217, 117); break;
            case 59: setXY(167, 117); break;
            case 60: setXY(117, 117); break;
            case 61: setXY(117, 167); break;
            case 62: setXY(117, 217); break;
            case 63: setXY(167, 217); break;
        }
    }
}
```

### 15.6.5 Clase ImagenTablero.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.awt.*;
import java.util.*;

/**
 * Esta clase es subclase de Canvas y contiene metodos para dibujarse en
pantalla.
 * Por lo demás, define las casillas del tablero (OCA, DADOS, etc.) y
proporciona
 * metodos para añadir fichas, moverlas, calcular nuevas casillas dependiendo
de
 * unos dados y obtener bonificaciones o penalizaciones de una casilla en
particular.
 */
public class ImagenTablero extends Canvas
{
   //Constantes que definen las casillas
   private final int NORMAL    = 0;     //Casillas normales
   private final int OCA       = 1;     //Casilla de OCAS
   private final int PUENTE    = 2;     //Casilla de PUENTE
   private final int POSADA    = 3;     //Casilla de POSADA
   private final int DADOS     = 4;     //Casilla de DADOS
   private final int POZO      = 5;     //Casilla de POZO
   private final int LABERINTO = 6;     //Casilla de LABERINTO
   private final int CARCEL    = 7;     //Casilaa de CARCEL
   private final int MUERTE    = 8;     //Casilla de MUERTE
   private final int FINAL     = 9;     //Casilla de FINAL

   //Array de casillas
   int[] casilla = new int[63];
   //Indicador de si ha pasado por la casilla final al mover la ficha
   boolean pasaPorElFinal;
   //Array de imagenes que representan los seis graficos del dado
   private Image i;
   //Lista de fichas del tablero
   private java.util.List ficha;
   //Mediatracker utilizado para añadir graficos al Canvas
   private MediaTracker m;

   /**
    * Metodo constructor.
    * Inicializa un tablero con el array de casillas y la imagen
representativa
    * del tablero de juego.
    */

   public ImagenTablero( String fichero )
   {
      //Asigna Casillas normales
      for( int i=0; i < 63; i++ ) casilla[i] = NORMAL;

      //Asigna casillas especiales
      casilla[ 4] = OCA;
      casilla[ 8] = OCA;
      casilla[13] = OCA;
      casilla[17] = OCA;
      casilla[22] = OCA;
      casilla[26] = OCA;
      casilla[31] = OCA;
      casilla[35] = OCA;
      casilla[40] = OCA;
```

```
        casilla[44] = OCA;
        casilla[49] = OCA;
        casilla[53] = OCA;
        casilla[58] = OCA;
        casilla[ 5] = PUENTE;
        casilla[11] = PUENTE;
        casilla[18] = POSADA;
        casilla[25] = DADOS;
        casilla[52] = DADOS;
        casilla[30] = POZO;
        casilla[41] = LABERINTO;
        casilla[51] = CARCEL;
        casilla[57] = MUERTE;
        casilla[62] = FINAL;

        //Inicializa el Toolkit para cargar imagenes
        java.awt.Toolkit tk = java.awt.Toolkit.getDefaultToolkit();

        //Carga la imagen a la propiedad i (Image)
        i = tk.getImage(fichero);

        //Inicializa unMediaTracker y añade la imagen a este Canvas
        m = new MediaTracker( this );
        m.addImage( i, 0 );

        //Espera a que se cargue la imagen
        try {
            m.waitForID( 0 );
        } catch( Exception e ){}

        //Establece el tamaño del Canvas, que se corresponde al tamaño de la
Imagen cargada
        setSize( i.getWidth(null), i.getHeight(null) );

        //Inicializa el array de fichas
        ficha = new ArrayList();
    }




    /**
     * Metodo de escritura.
     * Añade unas fichas (imagenes) definidas por unos jugadores al tablero.
     * @param Jugador[] array de jugadores que quieren añadir sus fichas al
tablero
     */
    public void addFichas( Jugador[] jugador )
    {
        //Recorre el array de jugadores
        for( int iterador = 0; iterador < jugador.length; iterador++ )
        {
            //Obtiene el numero que servirá para el grafico a cargar
            int numeroDeGrafico = ((jugador[iterador].getTurno()%5)+1);
            //Identificador que será de la ficha (dependiendo del turno del
jugador)
            int identificador = jugador[iterador].getTurno();
            //Inicializa una nueva ficha
            ImagenFicha fichaNueva = new ImagenFicha(
"ficha"+numeroDeGrafico+".png", (16*(numeroDeGrafico+1)), 368, identificador
);
            //Añade la ficha al array de fichas
            ficha.add( fichaNueva );
            //Añade la imagen al mediaTracker para poder visualizarla en
pantalla en este Canvas
            m.addImage( fichaNueva.getImagen(), identificador );
            //Espera a que se cargue correctamente
```

```
            try {
                m.waitForID( identificador );
            } catch( Exception e ){}
        }
        //Redibuja el Canvas para visualizar las fichas nuevas
        repaint();
    }


    /**
     * Metodo de escritura.
     * Añade una ficha (imagen) definida por el jugador.
     * @param Jugador jugador que tiene que insertar su ficha en el tablero
     */
    public void addFicha( Jugador jugador )
    {
        //Obtiene el numero que servirá para el grafico a cargar
        int numeroDeGrafico = ((jugador.getTurno()%5)+1);
        //Identificador que será de la ficha (dependiendo del turno del jugador)
        int identificador = jugador.getTurno();
        //Inicializa una nueva ficha
        ImagenFicha fichaNueva = new ImagenFicha(
"ficha"+numeroDeGrafico+".png", (16*(numeroDeGrafico+1)), 368, identificador
);
        //Añade la ficha al array de fichas
        ficha.add( fichaNueva );
        //Añade la imagen al mediaTracker para poder visualizarla en pantalla en
este Canvas
        m.addImage( fichaNueva.getImagen(), identificador );
        //Espera a que se cargue correctamente
        try {
            m.waitForID( identificador );
        } catch( Exception e ){}
        //Redibuja el Canvas para visualizar las fichas nuevas
        repaint();
    }

    /**
     * Metodo de actualizacion
     * Mueve una ficha identificada mediante el primer parametro, desde la
casilla inicial, hasta la casilla final,
     * esperando en cada movimiento de casilla "tiempoEspera" milisegundos.
     * @param int identificador de la ficha qe queremos mover
     * @param int casilla hacia donde se ha de mover
     * @param int tiempo de espera entre movimientos de casilla, en
milisegundos
     */
    public void mueveFicha( int identificador, int casillaFinal, int
tiempoEspera ) throws Exception
    {
        //Obtiene la ficha con el identificador
        for( Iterator it = ficha.iterator() ; it.hasNext() ;  )
        {
            ImagenFicha ficha = (ImagenFicha)it.next();
            //Si la ficha que obtenemos es la que nos interesa
            if( ficha.getIdentificador() == identificador )
            {
                //Si ha pasado por la casilla final, hay que hacer un movimiento
hacia mayores y despues
                //a menores. Hay que avanzar hasta la casilla 63 y despues
retroceder.
                if( pasaPorElFinal )
                {
                    //inicializa el flag de pasar por la ultima casilla
                    pasaPorElFinal = false;
                    //define variables de movimiento y otro flag que indica cuando
se posiciona en la casilla
                    //final
```

```
                int sentido = +1;   boolean haPasadoPorLaCasillaFinal = false;
                //Recorre todas las casillas por las que se movera la ficha
                for( int casillaActual = ficha.getCasilla() ; (casillaActual
>= casillaFinal) || !haPasadoPorLaCasillaFinal ; casillaActual += sentido )
                {
                    //Establece la nueva posicion de la ficha
                    ficha.setCasilla( casillaActual );
                    //Redibuja la imagen del tablero
                    repaint();
                    //Espera un tiempo
                    Thread.sleep( tiempoEspera );
                    //cambia de sentido y pone el flag de pasa por el final a
verdadero
                    if( casillaActual == 63 )
                    {
                        sentido = -1;
                        haPasadoPorLaCasillaFinal = true;
                    }
                }

            } else {
                //No ha pasado por la casilla final, movimiento normal
                //Si el movimiento es hacia casillas mayores
                if( ficha.getCasilla() < casillaFinal )
                {
                    //Recorre todas las casillas por las que se movera la
ficha
                    for( int casillaActual = ficha.getCasilla() ;
casillaActual <= casillaFinal ; casillaActual++ )
                    {
                        //Establece la nueva posicion de la ficha
                        ficha.setCasilla( casillaActual );
                        //Redibuja la imagen del tablero
                        repaint();
                        //Espera un tiempo
                        Thread.sleep( tiempoEspera );
                    }
                } else {
                    //Si el movimiento es hacia casillas menores
                    //Recorre todas las casillas por las que se movera la
ficha
                    for( int casillaActual = ficha.getCasilla() ;
casillaActual >= casillaFinal ; casillaActual-- )
                    {
                        //Establece la nueva posicion de la ficha
                        ficha.setCasilla( casillaActual );
                        //Redibuja la imagen del tablero
                        repaint();
                        //Espera un tiempo
                        Thread.sleep( tiempoEspera );
                    }
                }
            }
        }
    }
  }

  /**
   * Metodo de calculo
   * Calcula la nueva casilla dada una casilla actual y un resultado de un
dado lanzado
   * @param int casilla actual
   * @param int resultado de los dados
   */
  public int calculaNuevaCasilla( int casillaActual, int valorDados )
  {
      //calcula la nueva posicion
      int nuevaCasilla = casillaActual+valorDados;
```

```
        //Si es mayor a la casilla 63, calcula la nueva casilla que será menor a
la ultima
        if( nuevaCasilla > 63 ) {
            nuevaCasilla = 63 + (63-nuevaCasilla);
            //Pone el flag de pasar por el final a verdadero
            pasaPorElFinal = true;
        }
        //devuelve la casilla calculada
        return nuevaCasilla;
    }

    /**
     * Metodo de lectura.
     * Segun una casilla, calcula si se ha de volver a lanzar el dado.
     * @param int casilla del tablero
     * @return boolean si se ha de volver a lanzar, devuelve cierto. Falso en
otro caso
     */
    public boolean isMovimientoAutomatico( int casillaActual )
    {
        //Flag de volver a lanzar, en principio a falso
        boolean vuelveAMover = false;
        //Si esta en alguna de las casillas de OCA, etc. cambia el flag a
verdadero
        if( casilla[casillaActual-1] == OCA    ) vuelveAMover = true;
        if( casilla[casillaActual-1] == PUENTE ) vuelveAMover = true;
        if( casilla[casillaActual-1] == DADOS  ) vuelveAMover = true;
        if( casilla[casillaActual-1] == MUERTE ) vuelveAMover = true;
        //Devuelve el valor del flag
        return vuelveAMover;
    }

    /**
     * Metodo de lectura.
     * Obtiene la nueva casilla donde mover, si ha caido en una OCA, DADO, etc.
     * @param int casilla actual
     * @return int nueva casilla donde moverse
     */
    public int getNuevaCasilla( int casillaActual )
    {
        //nuevaCasilla será la nueva casilla donde moverse
        int nuevaCasilla = casillaActual;
        //Dependiendo de la casilla, escribe un mensaje de bonificacion y
obtiene la nueva casilla
        switch( casilla[casillaActual-1] )
        {
            //Si la casilla es una OCA
            case OCA:
                //Obtiene la nueva OCA en el tablero, hacia atrás o hacia delante
                nuevaCasilla = getSiguiente( casillaActual-1, OCA );
                //Escribe mensaje de bonificacion
                Juego.interfaz.escribeMensajeSistema( "¡De oca a oca y tiro
porque me toca!\navanzas y tiras otra vez" );
                break;
            case PUENTE:
                nuevaCasilla = getSiguiente( casillaActual-1, PUENTE );
                Juego.interfaz.escribeMensajeSistema( "¡De puente a puente y tiro
porque me lleva la corriente!\navanzas y tiras otra vez" );
                break;
            case DADOS:
                nuevaCasilla = getSiguiente( casillaActual-1, DADOS );
                Juego.interfaz.escribeMensajeSistema( "¡De dado a dado y tiro
porque me ha tocado!\navanzas y tiras otra vez" );
                break;
            case MUERTE:
                nuevaCasilla = 1;
                Juego.interfaz.escribeMensajeSistema( "¡Has caido en la
muerte!\nvuelve a la casilla numero 1!!!" );
```

```
                break;
        }
        //Devuelve la nueva casilla donde moverse
        return nuevaCasilla;
    }

    /**
     * Metodo de lectura.
     * Obtiene, dependiendo de un valor del casilla (OCA, DADO, etc.), la
siguiente
     * casilla con el mismo "palo" o valor.
     * @param int casilla actual
     * @param int figura o "palo" a buscar
     * @return int nueva casilla con la misma figura
     */
    private int getSiguiente( int casillaActual, int pFigura )
    {
        //Avanza hasta la siguiente casilla con la misma figura
        for( int i = casillaActual+1 ; i < 63 ; i++ )
            if( casilla[i] == pFigura ) return i+1;
        //Retrocede hasta la siguiente casilla con la misma figura
        for( int i = casillaActual-1 ; i > 0 ; i-- )
            if( casilla[i] == pFigura ) return i+1;
        //No se ha encontrado!!!
        return -1;
    }

    /**
     * Metodo de lectura.
     * Obtiene el numero de turnos sin tirar asociados a una casilla.
     * @param int casilla a valorar
     * @return int numero de turnos sin tirar
     */
    public int getTurnosSinTirar( int casillaActual )
    {
        //En un principio, son 0 turnos sin tirar (casilla normal)
        int turnosSinTirar = 0;
        //Dependiendo de la figura de la casilla valorar el numero de turnos sin
tirar
        //y escribir en pantalla el texto de penalizacion
        switch( casilla[casillaActual-1] )
        {
            //Si la casilla es una POSADA
            case POSADA:
                //Establece el numero de turnos sin tirar
                turnosSinTirar = 2;
                //Escribe en pantalla el mensaje de penalización
                Juego.interfaz.escribeMensajeSistema( "¡Has caido en la
posada!\npierdes dos turnos");
                break;
            case POZO:
                turnosSinTirar = 2;
                Juego.interfaz.escribeMensajeSistema( "¡Has caido en el
pozo!\npierdes dos turnos");
                break;
            case LABERINTO:
                turnosSinTirar = 2;
                Juego.interfaz.escribeMensajeSistema( "¡Has caido en el
laberinto!\npierdes dos turnos");
                break;
            case CARCEL:
                turnosSinTirar = 3;
                Juego.interfaz.escribeMensajeSistema( "¡Has caido en la
carcel!\npierdes tres turnos");
                break;
        }
        //Devuelve el numero de turnos sin tirar
        return turnosSinTirar;
```

```
    }

    /**
     * Metodo de lectura.
     * Obtiene el indicador de si se ha de volver a tirar, segun la figura de
una casilla determinada
     * @param int numero de casilla a valorar
     * @return boolean flag de si se ha de volver a lanzar el dado
     */
    public boolean getVuelveATirar( int casillaActual )
    {
        //Flag de volver a tirar a falso
        boolean vuelveATirar = false;
        //Valora las figuras de la casilla y pone el flag a verdadero
        if( casilla[casillaActual-1] == OCA    ) vuelveATirar = true;
        if( casilla[casillaActual-1] == PUENTE ) vuelveATirar = true;
        if( casilla[casillaActual-1] == DADOS  ) vuelveATirar = true;
        //Devuelve el flag de volver a lanzar el dado
        return vuelveATirar;
    }

    /**
     * Metodo de lectura.
     * Devuelve si la casilla es la última del tablero o no
     * @param int casilla actual
     * @return boolean si se ha alcanzado la última casilla devuelve cierto
     */
    public boolean isCasillaFinal( int casilla )
    {
        //Flag de casilla final a falso
        boolean casillaFinal = false;
        //Comprueba si es la ultima casilla
        if( casilla == 63 )
        {
            //Pone el flag a verdadero
            casillaFinal = true;
            //Escribe un mensaje en pantalla
            Juego.interfaz.escribeMensajeSistema( "¡HAS GANADO!");
        }
        //Devuelve el flag de casilla final
        return casillaFinal;
    }

    /**
     * Metodo de actualizacion.
     * Este metodo se llama cada vez que se quiere redibujar la pantalla (una
llamada al metodo
     * repaint()).
     * El metodo original del Canvas, primero borra la pantalla, pero para
evitar el "flickering", o
     * el parpadeo de imagen, reimplementamos este metodo para evitarlo,
simplemente llamando al metodo
     * de dibujado, sin borrar antes la pantalla.
     * @param Graphics la "pantalla" donde se quiere actualizar.
     */
    public void update( Graphics g )
    {
        paint(g);
    }

    /**
     * Metodo de dibujado.
     * Este metodo dibuja el grafico, indicado por la propiedad de frame
actual, del dado.
     * es una reimplementacion del mismo método de la clase Canvas. Solamente
nos interesa dibujar
     * el dado.
     * @param Graphics pantalla donde se quiere dibujar
```

```
    */
    public void paint( Graphics g )
    {
        //Dibuja la imagen del dado indicada por la propiedad frameActual, en la
posicion x=0, y=0.
        g.drawImage( i, 0, 0, this );
        //Dibuja las fichas de los jugadores que hay en la partida
        for( Iterator it = ficha.iterator() ; it.hasNext() ;  )
        {
            //Obtiene la siguiente ficha
            ImagenFicha imF = (ImagenFicha)it.next();
            //Dibuja la ficha en la posicion en la que esta actualmente
            g.drawImage( imF.getImagen(), imF.getX(), imF.getY(), this );
        }
    }
}
```

### 15.6.6 Clase Interfaz.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.awt.*;
import java.awt.List;
import java.awt.event.*;
import java.awt.image.*;
import java.util.*;


/**
 * Clase que representa la interfaz del juego.
 * Esta clase es subclase de Frame y añade una serie de paneles, cada una con
diferentes configuraciones
 * de botones, areas de texto, listas y paneles de imagenes, adecuados al
juego de la Oca
 * Esta configurado por 5 paneles principales:
 *
 * 1.- El area de textos de sistema. Aquí se escribirán la mayoria de los
mensajes indicadores del juego,
 *     conexiones de jugadores, lanzamiento de dados, establecimiento de
turnos, etc.
 *
 * 2.- El area de lista de jugadores activos. Aquí se verán todos los
jugadores que estan jugando, incluido
 *     el jugador actual, que se refleja en primera posicion. Se escribe el
nombre del jugador, su direccion
 *     IP y puerto, así como el turno que tiene asignado.
 *
 * 3.- El area de mensajes de error. Aquí se escriben los mensajes de error
que genera la aplicación.
 *     Indicarán las clases y métodos que generan los errores.
 *
 * 4.- El area de botones. Aquí estarán los botones de salir del programa,
iniciar el juego, o tirar los dados
 *
 * 5.- El area de dibujo. Estará el tablero de juego (con las fichas que
contiene).
 *
 */
public class Interfaz extends Frame
{
   //TextArea que representa el area de mensajes de sistema
   private TextArea areaTextoMensajesSistema;
   //TextArea que representa el area de mensajes de error
   private TextArea areaTextoMensajesError;
   //Boton de inicio de juego
   private Button botonInicio;
   //Indicador de boton pulsado
   private boolean botonInicioPulsado;
   //Boton de lanzamiento del dado
   private Button botonDado;
   //Indicador de boton pulsado
   private boolean botonDadoPulsado;
   //Boton de salir del juego
   private Button botonSalir;
   //Lista que representa los jugadores que hay en la partida
   private List listaJugadoresActivos;
   //Imagen del tablero
   private ImagenTablero imagenTablero;
   //Imagen del dado
   private ImagenDado imagenDado;
```

```
    /**
     * Metodo Constructor.
     * Este metodo inicializa la interfaz. En primer lugar llama al mismo
metodo de la
     * superclase (Frame).
     * Seguidamente inicializa las propiedades de la clase, carga la imagen
para el tablero,
     * y construye las diferenres areas de paneles que incluira la interfaz.
Cada uno de esos
     * paneles contienen a su vez diferentes areas.
     * Establece el tamaño de la ventana y la muestra.
     */
    public Interfaz()
    {
        //Llama a la superclase, mismo metodo (constructor)
        super("El juego seguro de la OCA");

        //Inicializa un BorderLayout (tiene areas norte, sur, este y oeste) y
        setLayout( new BorderLayout() );
        add( "North" , layoutCopyright() );
        add( "East"  , layoutConexion() );
        add( "Center", layoutTablero() );
        add( "South" , layoutBotones() );

        //Añade un listener para la accion de cerrar la ventana, en cuyo caso
borra al jugador
        //local y sale de la aplicacion.
        addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing( WindowEvent e )
                {
                    //Llamada de sistema, para salir de la aplicacion
                    System.exit( 0 );
                }
            }
        );

        //Establece el tamaño de la ventana
        setSize(775, 525);

        //Muestra la ventana
        show();
    }

    /**
     * Area de Mensajes y conexiones.
     * Mediante un borderLayout, inicializa un panel donde se incluyen los
mensajes de sistema,
     * la lista de jugadores conectados y los mensajes de error generados.
     * @return Panel panel de mensajes y conexiones.
     */
    public Panel layoutConexion( )
    {
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un borderLayout
        panel.setLayout( new BorderLayout() );
        //Añade el panel mensajes de sistema
        panel.add( "North" , layoutMensajesSistema() );
        //Añade el panel de jugadores conectados
        panel.add( "Center", layoutQuienHayConectado() );
        //Añade el panel de mensajes de error
        panel.add( "South" , layoutMensajesError() );
        //Retorna el panel
        return panel;
    }
```

```
    /**
     * Area de Conexion.
     * Mediante un border layout, muestra la etiqueta "Jugadores conectados" y
añade
     * la lista de jugadores que hay conectados actualmente
     * @return Panel panel con la lista de jugadores activos (a cero).
     */
    public Panel layoutQuienHayConectado( )
    {
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un borderLayout
        panel.setLayout( new BorderLayout() );
        //Añade la etiqueta de nombre del panel
        panel.add( "North" , new Label( "Jugadores conectados" ) );
        //Añade una lista contenedora de los jugadores activos (a cero)
        panel.add( "Center", listaJugadoresActivos = new List( 10, false ) );
        //Retorna el panel
        return panel;
    }

    /**
     * Area de Mensajes de sistema.
     * Mediante un border layout, crea un area de texto donde se pueden
escribir
     * los mensajes de sistema desde diferentes metodos de las clases de la
aplicacion
     * @return Panel panel con el area de texto de mensajes (vacia).
     */
    public Panel layoutMensajesSistema( )
    {
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un borderLayout
        panel.setLayout( new BorderLayout() );
        //Añade una etiqueta con el nombre del panel
        panel.add( "North" , new Label( "Mensajes del sistema" ) );
        //Añade un area de texto donde se escribirán los mensajes de sistema
        panel.add( "Center", areaTextoMensajesSistema = new TextArea(10, 40) );
        //Inicializa el entorno del area de texto, no se puede escribir y esta
        //activo
        areaTextoMensajesSistema.setEnabled( true );
        areaTextoMensajesSistema.setEditable( false );
        //Retorna el panel creado
        return panel;
    }

    /**
     * Area de Copyright.
     * Mediante un border layout, crea una etiqueta mostrando informacion
     * del copyright de la aplicacion.
     * @return Panel panel con informacion del copyright del juego.
     */
    public Panel layoutCopyright( )
    {
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un borderLayout
        panel.setLayout( new BorderLayout() );
        //Añade una etiqueta con informacion del copyright del juego
        panel.add( "Center" , new Label( "Daniel Celemín García - TFC Seguridad
UOC" ) );
        //Retorna el panel creado
        return panel;
    }

    public Panel layoutTablero( )
    {
```

```
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un borderLayout
        panel.setLayout( new BorderLayout() );
        //Añade una etiqueta con informacion del copyright del juego
        panel.add( "North" , new Label( "Tablero de juego" ) );
        panel.add( "Center", imagenTablero = new ImagenTablero( "tablero.png" )
);
        //Retorna el panel creado
        return panel;
    }

    /**
     * Area de Mensajes de error.
     * Mediante un border layout, crea un area de texto donde se pueden
escribir
     * los mensajes de error desde diferentes metodos de las clases de la
aplicacion
     * @return Panel panel con el area de texto de mensajes de error (vacia).
     */
    public Panel layoutMensajesError( )
    {
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un borderLayout
        panel.setLayout( new BorderLayout() );
        //Añade una etiqueta con el nombre del panel
        panel.add( "North" , new Label( "Mensajes de error" ) );
        //Añade un area de texto con los mensajes de error de la aplicacion
(vacia)
        panel.add( "Center", areaTextoMensajesError = new TextArea(3, 40) );
        //Inicializa el estado del area de texto, activada y no se puede editar
        areaTextoMensajesError.setEnabled( true );
        areaTextoMensajesError.setEditable( false );
        //Retorna el panel creado
        return panel;
    }

    /**
     * Area de Botones.
     * Mediante un Flow layout (dispone componentes en columnas o filas), añade
botones
     * y les asigna un "Listener" (escuchador) de acciones (boton pulsado).
     * @return Panel el panel de botones creado
     */
    public Panel layoutBotones( )
    {
        //Inicializa un panel
        Panel panel = new Panel();
        //Inicializa un FlowLayout
        panel.setLayout( new FlowLayout( FlowLayout.LEFT, 25, 5 ) );
        //Añade un boton de comienzo de partida
        panel.add( botonInicio = new Button( "Comenzar partida" ) );
        //Le asigna un "Listener" que actua al pulsar el boton
        botonInicio.addActionListener(
            new ActionListener()
            {
                public void actionPerformed( ActionEvent e )
                {
                    //Pone el flag de boton pulsado a verdadero
                    botonInicioPulsado = true;
                }
            }
        );
        //Añade un boton de lanzamiento de dado
        panel.add( botonDado = new Button( "Lanzar dado" ) );
        //Añade un listener de acciones
        botonDado.addActionListener(
```

```
            new ActionListener()
            {
                public void actionPerformed( ActionEvent e )
                {
                    //Pone a verdadero el flag de boton pulsado
                    botonDadoPulsado = true;
                }
            }
        );
        //Añade una imagen de un dado
        panel.add( imagenDado = new ImagenDado() );
        panel.add( botonSalir = new Button( "Salir del juego" ) );
        //Asigna un listener a las acciones del boton
        botonSalir.addActionListener(
            new ActionListener()
            {
                public void actionPerformed( ActionEvent e )
                {
                    //Sale de la aplicacion
                    System.exit( 0 );
                }
            }
        );
        //Retorna el panel creado
        return panel;
    }

    /**
     * Metodo de escritura.
     * Refresca la lista de jugadores activos. Recorre todos los jugadores
activos y
     * recoge informacion de todos ellos.
     * @param Jugador jugador local
     * @param Jugador[] lista de jugadores en la partida
     */
    public void showQuienHayConectado( Jugador[] jugadoresActivos )
    {
        //Borra la lista de jugadores activos
        listaJugadoresActivos.removeAll();
        //Recorre la lista de jugadores activos y añade al area de lista de
jugadores
        //la informacion obtenida de cada uno de ellos.
        for( int i = 0 ; i < jugadoresActivos.length ; i++ )
            listaJugadoresActivos.add( jugadoresActivos[ i ].toString() , i+1 );
    }

    /**
     * Metodo de escritura.
     * Escribe un mensaje de error en el area de mensajes de error.
     * @param String mensaje que se quiere escribir
     */
    public void escribeMensajeError( String mensaje )
    {
        //Añade el mensaje al area de texto de mensajes de error con un final de
linea.
        areaTextoMensajesError.append( mensaje + "\n" );
    }

    /**
     * Metodo de escritura.
     * Escribe un mensaje de sistema en el area de mensajes de sistema.
     * @param String mensaje que se quiere escribir
     */
    public void escribeMensajeSistema( String mensaje )
    {
        //Añade el mensaje al area de texto de mensajes de sistema con un final
de linea.
        areaTextoMensajesSistema.append( mensaje + "\n" );
```

```
    }

    /**
     * Metodo de lectura.
     * Obtiene el estado del flag del boton de inicio pulsado
     * @return boolean true si el boton se ha pulsado, false en caso contrario
     */
    public boolean isBotonInicioPulsado()
    {
        return botonInicioPulsado;
    }

    /**
     * Metodo de escritura.
     * Establece el flag de boton de inicio pulsado. (para restaurar el valor a
false)
     */
    public void setBotonInicioPulsado( boolean pBotonInicio )
    {
        botonInicioPulsado = pBotonInicio;
    }

    /**
     * Metodo de lectura.
     * Obtiene el estado del flag del boton de dado lanzado
     * @return boolean true si el boton se ha pulsado, false en caso contrario
     */
    public boolean isBotonDadoPulsado()
    {
        return botonDadoPulsado;
    }

    /**
     * Metodo de escritura.
     * Establece el flag de boton de dado pulsado.
     */
    public void setBotonDadoPulsado( boolean pBotonDado )
    {
        botonDadoPulsado = pBotonDado;
    }

    /**
     * Metodo de escritura.
     * Deshabilita la posibilidad de presionar el boton de inicio.
     */
    public void disableBotonInicio()
    {
        botonInicio.setEnabled( false );
    }

    /**
     * Metodo de escritura.
     * Habilita la posibilidad de presionar el boton de inicio.
     */
    public void enableBotonInicio()
    {
        botonInicio.setEnabled( true );
    }

    /**
     * Metodo de escritura.
     * Deshabilita la posibilidad de presionar el boton de lanzamiento de dados
     */
    public void disableBotonDado()
    {
        botonDado.setEnabled( false );
    }
```

```
    /**
     * Metodo de escritura.
     * Habilita la posibilidad de presionar el boton de lanzamiento de dados.
     */
    public void enableBotonDado()
    {
        botonDado.setEnabled( true );
    }

    /**
     * Metodo de lectura
     * obtiene la imagen del tablero del juego
     * @return ImagenTablero el tablero de juego
     */
    public ImagenTablero getTablero()
    {
        return imagenTablero;
    }

    /**
     * Metodo de lectura
     * Obtiene la imagen del dado del juego
     * @return ImagenDado el dado del juego
     */
    public ImagenDado getDado()
    {
        return imagenDado;
    }
}
```

### 15.6.7 Clase Juego.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.io.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;

/**
 * Clase principal de la aplicacion.
 * Esta clase proporciona constantes de puertos por defecto, version de la
aplicacion, etc.
 * Contiene la lógica de la aplicacion:
 *
 * 1.- Parsea los argumentos de entrada, es decir, separa cada uno de los
parametros de entrada y los
 *     guarda en las variables correspondientes, si hay error, enseña un
mensaje de ayuda por consola.
 * 2.- Dependiendo de si se le indica o no la direccion IP a conectarse, crea
una partida nueva o se conecta
 *     a una existente.
 * 3.- Espera coneciones entrantes de jugadores.
 * 4.- Si se es iniciador, cuando se pulsa el boton de iniciar partida, se
envia un mensaje a todos
 *     los jugadores de la partida que comienza la partida, si no se es
iniciador, cuando el que tiene
 *     el turno, nos envía el resultado de los dados cifrado, iniciamos la
partida
 * 5.- Se calculan los dados cifrados y normales, y comienza el protocolo de
lanzamiento del dado utilizando
 *     la teoria del compromiso de BIT.
 * 6.- Finalmente, se calcula el resultado de los dados y se actualiza la
posicion del jugador que
 *     tiene el turno, dibujando las fichas en el tablero, etc.
 */
public class Juego
{
   //Define el puerto que se utiliza por defecto
   public static final int PUERTO_POR_DEFECTO = 9999;
   //Puerto utilizado por defecto

   //Variables de argumentos entrados
   public static String nombre             = null;
   //nombre del jugador local
   public static String direccionIP        = null;
   //direccion IP de conexion
   public static int puerto                = PUERTO_POR_DEFECTO;
   //puerto de conexion remoto
   public static int puertoPropio          = PUERTO_POR_DEFECTO;
   //puerto de conexion propio
   public static String fileNormalKeystore = null;
   //fichero de claves privadas como servidor y como cliente
   public static String typeNormalKeystore = null;
   //Tipo de fichero JKS, P12, etc.
   public static String fileTrustKeystore  = null;
   //fichero de claves publicas certificadas (con certificado)
   public static String typeTrustKeystore  = null;
   //tipo de fichero JKS, P12, etc.
   public static String pswNormalKeystore  = null;
   //password del fichero de claves privada
   public static String pswTrustKeystore   = null;
   //password del fichero certificado de claves publicas
```

```
    //Interfaz del juego
    public static Interfaz interfaz;

    //partida del juego
    public static Partida partida;

    /**
     * Metodo de escritura.
     * Escribe el mensaje de ayuda de parametros de la aplicacion.
     * Devuelve el mensaje de error del parser de los argumentos, si los hay
     * @param String Mensaje de error a escribir
     */
    public static void enseñaAyuda( String pError )
    {
        System.out.println( pError );
        System.out.println( "Parametros de entrada" );
        System.out.println( "         1 : Nombre del jugador" );
        System.out.println( "Opcional 2 : Direccion IP del iniciador de la
partida, entre parentesis. Ej. (127.0.0.1)" );
        System.out.println( "Opcional 3 : Puerto de conexion entre corchetes.
Ej. [9999]" );
        System.out.println( "Opcional 4 : Puerto de conexion de escucha entre
guiones. Ej. -9998-" );
        System.out.println( "Opcional 5 : Almacenes de claves publica/privada y
certificados local y remoto." );
        System.out.println( "Si se especifica 5, 5.1 : Almacen (Keystore) de
claves privada/publica local." );
        System.out.println( "Si se especifica 5, 5.2 : Tipo de almacen. Ej. JKS,
P12, etc." );
        System.out.println( "Si se especifica 5, 5.3 : Password del almacen." );
        System.out.println( "Si se especifica 5, 5.4 : Almacen (Keystore) de
claves publica y certificado remoto." );
        System.out.println( "Si se especifica 5, 5.5 : Tipo de almacen. Ej. JKS,
P12, etc." );
        System.out.println( "Si se especifica 5, 5.6 : Password del almacen." );
    }

    /**
     * Metodo de actualizacion.
     * Actualiza las propiedades de la aplicacion, si no existe error
     * Lee los argumentos de entrada y los clasifica, dependiendo de si existe
     * error.
     * @param String[] array de Strings que contienen los argumentos entrados
por consola por
     * el usuario
     * @return String Mensaje de error, o null si no existe error
     */
    private static String parseArgumentos( String[] args )
    {
        //Inicializa un string que servirá como variable auxiliar
        String s = null;

        //Si no se ha entrado ningun parametro
        if( (args == null) || (args.length == 0) ) return "No hay suficientes
argumentos";

        //Lee el nombre del jugador, quitando espacios en blanco
        nombre = args[0].trim();
        if( nombre.length() == 0 ) return "No ha especificado un nombre de
jugador";

        //lee el siguiente parametro si existe
        int indice = 1; if( args.length > indice ) s = args[indice].trim(); else
return null;

        //Si esta entre parentesis, es la direccion IP a la que nos queremos
conectar
        if( s.startsWith( "(" ) && s.endsWith( ")" ) ) {
```

```
            direccionIP = s.substring( 1, s.length()-1 );
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return null;
        }

        //Si esta entre corchetes, es el puerto que se utiliza para conectarse
al remoto
        if( s.startsWith( "[" ) && s.endsWith( "]" ) ) {
            puerto = Integer.parseInt( s.substring( 1, s.length()-1 ) );
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return null;
        }

        //Si esta entre guiones, es el puerto que se utiliza para escucha de
conexiones o puerto local
        if( s.startsWith( "-" ) && s.endsWith( "-" ) ) {
            puertoPropio = Integer.parseInt( s.substring( 1, s.length()-1 ) );
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return null;
        }

        //Lee el nombre de los keystores Normal y Trust
        if( s.length() != 0 ) {
            fileNormalKeystore = s;
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return "No se ha especificado el tipo de almacen: JKS, P12, etc.";
            typeNormalKeystore = s;
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return "No se ha especificado el password para el almacen de claves";
            pswNormalKeystore = s;
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return "No se ha especificado el Trust Keystore del remoto";
            fileTrustKeystore = s;
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return "No se ha especificado el tipo de almacen: JKS, P12, etc.";
            typeTrustKeystore = s;
            //lee el siguiente parametro si existe
            indice++; if( args.length > indice ) s = args[indice].trim(); else
return "No se ha especificado el password para el almacen de clave publica y
certificado firmado";
            pswTrustKeystore = s;
        }
        return null;
    }

    /**
     * Metodo principal.
     * Este metodo es el principal de la aplicacion. Ver explicacion de clase.
     * @param String argumentos de entrada
     */
    public static void main(String args[]) throws Exception
    {

        //obtiene los argumentos de entrada y retorna error si existe
        String error = parseArgumentos( args );

        //Si hay error en los argumentos
        if( error != null ) {

            //Escribe el mensaje de error por consola y la ayuda de la entrada
de parametros
```

```
            enseñaAyuda( error );
            System.exit(0);

        } else {

            try {

                //Creamos la interfaz del juego
                interfaz = new Interfaz();

                interfaz.disableBotonInicio();
                interfaz.disableBotonDado();

                //----------------------------------
                //FASE DE INICIALIZACION Y CONEXION
                //----------------------------------

                //Si se trata del iniciador de partida (no se ha indicado
direccion de conexion)
                if( direccionIP == null ) {

                    //Crea una nueva partida
                    partida = new Partida( nombre, puertoPropio,
fileNormalKeystore, typeNormalKeystore, pswNormalKeystore, fileTrustKeystore,
typeTrustKeystore, pswTrustKeystore );

                    //Enseña quien hay conectado (solamente estará el iniciador!)
                    interfaz.showQuienHayConectado( partida.getJugadores() );
                    interfaz.getTablero().addFicha( partida.getJugadorLocal() );

                    //Habilita el boton de iniciar partida
                    interfaz.enableBotonInicio();

                    //Muestra un mensaje de inicio de partida
                    interfaz.escribeMensajeSistema( "Se ha iniciado una nueva
partida" );
                    interfaz.escribeMensajeSistema( "esperando jugadores..." );

                    //Mientras no se pulse el boton de iniciar partida
                    while( !interfaz.isBotonInicioPulsado() ) {

                        //Espera la conexion de un jugador durante 100
milisegundos
                        Jugador nuevoJugador = partida.esperaJugador( 100 );

                        if( nuevoJugador != null ) {

                            //Añade la ficha en el tablero del nuevo jugador
                            interfaz.getTablero().addFicha( nuevoJugador );

                            //Enseña quien hay conectado
                            interfaz.showQuienHayConectado( partida.getJugadores()
);
                        }

                    }

                    //Envia a todos los jugadores de la partida el mensaje de
comienzo de partida
                    partida.enviaATodos( Mensajes.COMIENZA_PARTIDA );

                } else {

                    //Crea una partida conectandose a una direccion
                    partida = new Partida( nombre, puertoPropio,
fileNormalKeystore, typeNormalKeystore, pswNormalKeystore, fileTrustKeystore,
typeTrustKeystore, pswTrustKeystore, direccionIP, puerto );
```

```
                    //Enseña quien hay conectado
                    interfaz.showQuienHayConectado( partida.getJugadores() );
                    interfaz.getTablero().addFichas( partida.getJugadores() );

                    //Muestra un mensaje de partida conectada
                    interfaz.escribeMensajeSistema( "Se ha conectado con la
partida" );
                    interfaz.escribeMensajeSistema( "esperando jugadores..." );

                    //Mientras el jugador iniciador (con turno 0) no inicie la
partida
                    while( !partida.getJugadorConTurno().isComenzarPartida() ) {

                        //Esperar conexiones de un jugador invitados (durante 100
milisegundos)
                        Jugador nuevoJugador = partida.esperaJugador( 100 );

                        if( nuevoJugador != null ) {

                            //Añade la ficha en el tablero del nuevo jugador
                            interfaz.getTablero().addFicha( nuevoJugador );

                            //Enseña quien hay conectado
                            interfaz.showQuienHayConectado( partida.getJugadores()
);
                        }

                    }
                }

                //-------------------------
                // FASE DE JUEGO
                //-------------------------

                //Deshabilita el boton de inicio de partida
                interfaz.disableBotonInicio();

                //Enseña un mensaje de comienzo de partidas
                interfaz.escribeMensajeSistema( "Comienza la partida!" );

                //Inicializa un objeto que contendra al jugador con el turno
                Jugador jugadorConTurno;

                //Mientras no este la partida finalizada
                while( !partida.isFinalizada() )
                {
                    //Deshabilita el boton de lanzamiento de dados, solamente el
que tenga el turno
                    //podrá comenzar a lanzar los dados
                    interfaz.disableBotonDado();

                    //Resetea los dados de los jugadores, los pone a null
                    partida.resetDadoDeJugadores();

                    //Obtiene el jugador que tiene el turno
                    jugadorConTurno = partida.getJugadorConTurno();

                    //Si puede jugar
                    if( jugadorConTurno.getTurnosSinTirar() == 0 )
                    {

                        //Enseña quien es el jugador con el turno
                        interfaz.escribeMensajeSistema(
"*************************************" );
                        interfaz.escribeMensajeSistema( "Juega : " +
jugadorConTurno.getNombre() );

                        //Si es el jugador local
```

```
                    if( jugadorConTurno == partida.getJugadorLocal() )
                    {
                        ////Habilita el boton de lanzamiento de dados
                        interfaz.enableBotonDado();
                        interfaz.escribeMensajeSistema( "Lanza los dados
pulsando\nel boton 'Lanzar dados'" );

                        //Pone a false el flag de boton pulsado
                        interfaz.setBotonDadoPulsado( false );

                        //Espera hasta que el usuario pulse el boton de
lanzamiento de dados
                        while( !interfaz.isBotonDadoPulsado() ) {}

                    } else {

                        //Mientras el jugador que tiene el turno no haya
enviado
                        //el valor de su lanzamiento de dado encriptado
                        while( !jugadorConTurno.getDado().isValorCifrado() ) {}

                    }

                    //Se lanza el dado de forma conjunta...
                    partida.getJugadorLocal().getDado().lanzar();

                    //lanza el dado y se lo envia a todos cifrado
                    partida.enviaATodos( Mensajes.DADOCIFRADO );
                    partida.enviaATodos( new Base64().encode(
partida.getJugadorLocal().getDado().getValorCifrado() ) );
                    //Espera recibir de todos el dado cifrado
                    partida.recibeDeTodos( Mensajes.DADOCIFRADO );

                    //envia a todos la clave utilizada para encriptar el dado
                    partida.enviaATodos( Mensajes.DADOCLAVE );
                    partida.enviaATodos( new Base64().encode(
partida.getJugadorLocal().getDado().getValorClave() ) );
                    //Espera recibir de todos las claves utilizadas para
cifrar los dados
                    partida.recibeDeTodos( Mensajes.DADOCLAVE );

                    //Calcula el valor del lanzamiento de dados compartido
                    int valorDados = partida.calculaDadoConjunto();
                    interfaz.getDado().simulaLanzamiento( valorDados );

                    //Notifica el resultado del dado
                    interfaz.escribeMensajeSistema( "Resultado de los dados: "
+ String.valueOf(valorDados) );

                    //-------------------------
                    //ACTUALIZACION DE POSICION
                    //-------------------------

                    int nuevaCasilla;

                    //el tablero calcula la nueva casilla
                    nuevaCasilla = interfaz.getTablero().calculaNuevaCasilla(
jugadorConTurno.getCasilla(), valorDados );
                    //Mueve la ficha del tablero
                    interfaz.getTablero().mueveFicha(
jugadorConTurno.getTurno(), nuevaCasilla, 500 );
                    //Actualiza la nueva casilla del jugador
                    jugadorConTurno.setCasilla( nuevaCasilla );

                    //Si la casilla avanza automaticamente (OCA, PUENTE,
DADOS, MUERTE)
                    if( interfaz.getTablero().isMovimientoAutomatico(
nuevaCasilla ) )
```

```
                    {
                        //el tablero calcula la nueva casilla
                        nuevaCasilla = interfaz.getTablero().getNuevaCasilla(
nuevaCasilla );
                        //Mueve la ficha del tablero
                        interfaz.getTablero().mueveFicha(
jugadorConTurno.getTurno(), nuevaCasilla, 250 );
                        //Actualiza la nueva casilla del jugador
                        jugadorConTurno.setCasilla( nuevaCasilla );
                    }

                    //Comprueba las bonificaciones de la casilla
                    jugadorConTurno.setTurnosSinTirar(
interfaz.getTablero().getTurnosSinTirar( nuevaCasilla ) );
                    jugadorConTurno.setVuelveATirar(
interfaz.getTablero().getVuelveATirar( nuevaCasilla ) );
                    jugadorConTurno.setGanador(
interfaz.getTablero().isCasillaFinal( nuevaCasilla ) );

                    //Enseña quien hay conectado para actualizar el valor de
la casilla
                    interfaz.showQuienHayConectado( partida.getJugadores() );

                } else {

                    //enseña el mensaje de turno sin tirar
                    interfaz.escribeMensajeSistema(
"*************************************" );
                    interfaz.escribeMensajeSistema( "Juega : " +
jugadorConTurno.getNombre() );
                    interfaz.escribeMensajeSistema( "Turno sin tirar..." );
                    jugadorConTurno.setTurnosSinTirar(
jugadorConTurno.getTurnosSinTirar()-1 );

                }

                //Si el jugador activo es ganador, finaliza partida
                if( jugadorConTurno.isGanador() ) partida.setFinalizada(true);
                else
                {
                    //Si el jugador tiene que volver a tirar (ha caido en un
dado, oca, o puente)
                    if ( jugadorConTurno.getVuelveATirar() )
                        //restaura el flag de volver a tirar a false
                        jugadorConTurno.setVuelveATirar( false );
                    else
                        //Cambia de jugador segun su turno
                        partida.cambiaTurno();
                }

            }
        } catch( Exception e ) {
            //Enseña un mensaje de error
            interfaz.escribeMensajeError( e.toString() );
        }
    }
  }
}
```

### 15.6.8 Clase Jugador.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.io.*;
import java.net.*;
import javax.net.ssl.*;
import javax.crypto.*;

/**
 * Clase que representa la "imagen" del estado de un jugador.
 * En esta clase se recogen los parametros normales de un jugador como su
nombre, su direccionIP y
 * su puerto de conexion, y a través del socket asociado se puede enviar
informacion, mensajes y
 * demás. A la vez que contiene el socket de comunicacion, también contiene el
stream de salida y
 * de entrada de información.
 * Es una clase que extiende el uso de la clase Thread, es decir, es un hilo
independiente del flujo
 * de ejecucion normal. Esto es así ya que esta clase, recibe de su homologo
en el otro extremo mensajes
 * de peticion, con lo cual, inmediatamente despues de recibir un mensaje,
puede actualizar valores
 * sin necesidad de hacer esperar al programa principal. También es debido a
que el objeto creado se
 * ha de esperar indefinidamente a la recepcion de algun mensaje, con lo cual,
bloquearia al programa
 * principal.
 */
public class Jugador extends Thread
{
    //Nombre del jugador al cual se esta conectado
    private String nombreJugador;
    //Direccion IP del jugador al cual se esta conectado
    private String direccionIP;
    //Puerto asociado a este jugador
    private int puerto;
    //Socket por donde se comunica con este jugador
    private SSLSocket socket;
    //Streams de entrada y salida de informacion del jugador.
    private StreamEntrada i;
    private StreamSalida o;
    //Numero de casilla donde esta el jugador
    private int casilla;
    //Representa el turno de juego que tiene asignado el jugador
    //Tambien dependiendo del turno asignado, controlará una ficha u otra
    private int turno;
    //Representa el numero de turnos que se tiene que quedar sin jugar.
    private int turnosSinTirar;
    //Indica si vuelve a tirar, despues de un lanzamiento de dados
    private boolean vuelveATirar;
    //Indica el numero identificador de la ficha dentro de la imagen del
tablero.
    private int identificadorFicha;
    //Dado del jugador
    private Dado dado;
    //Flag que representa la decision de comenzar una partida (solo válido para
el iniciador de partida)
    private boolean comenzarPartida;
    //Flag que representa la decision de comenzar una partida (solo válido para
el iniciador de partida)
    private boolean ganador;
```

```
    /**
     * Metodo constructor.
     * Inicializa un jugador sin asignarle turno de juego y lanza el metodo de
ejecucion del hilo.
     */
    public Jugador( String pN, String pD, int pP, SSLSocket pS, StreamEntrada
pI, StreamSalida pO, int pT ) throws Exception
    {
        //Asigna las propiedades del objeto creado
        nombreJugador = pN;
        direccionIP   = pD;
        puerto = pP;
        socket = pS;
        i = pI;
        o = pO;
        turno = pT;

        //Variables de estado
        turnosSinTirar = 0;
        casilla = 0;
        vuelveATirar = false;
        dado = new Dado( 6 );
        comenzarPartida = false;

        //Lanza la ejecucion del hilo mediante el metodo run()
        start();
    }

    /**
     * Metodo de lectura
     * Obtiene el nombre del jugador
     * @return String nombre del jugador
     */
    public String getNombre()
    {
        return nombreJugador;
    }

    /**
     * Metodo de lectura
     * Obtiene la direccion IP asignada a este jugador
     * @return String direccion IP del jugador, en forma XXX:XXX:XXX:XXX
     */
    public String getDireccionIP()
    {
        return direccionIP;
    }

    /**
     * Metodo de lectura
     * Obtiene el puerto asignado a este jugador.
     * @return int puerto asignado a la comunicacion con este jugador
     */
    public int getPuerto()
    {
        return puerto;
    }

    /**
     * Metodo de lectura
     * Obtiene el socket seguro (SSL) asignado a la comunicacion con este
jugador
     * @return  SSLSocket socket asociado a la comunicacion con este jugador
     */
    public SSLSocket getSocket()
    {
        return socket;
```

```
    }

    /**
     * Metodo de lectura
     * Obtiene la casilla donde se encuentra este jugador
     * @return int casilla donde esta el jugador
     */
    public int getCasilla()
    {
        return casilla;
    }

    /**
     * Metodo de escritura
     * Establece la posicion del jugador en una casilla determinada
     * @param int numero de casilla asignada al jugador
     */
    public void setCasilla( int pValor )
    {
        casilla = pValor;
    }

    /**
     * Metodo de lectura
     * Obtiene el identificador del turno del jugador.
     * @return int identificador del turno del jugador
     */
    public int getTurno()
    {
        return turno;
    }

    /**
     * Metodo de escritura
     * Establece el turno del jugador. Si ya tenia un turno asignado, borra la
ficha
     * que le correspondia y añade la nueva una vez establecido el turno
     * @param int turno que se le quiere asignar a este jugador
     */
    public void setTurno( int pTurno )
    {
        //Asigna el turno de juego
        turno = pTurno;
    }

    /**
     * Metodo de escritura
     * Escribe un mensaje al jugador, siempre y cuando el socket siga operativo
     * @param String mensaje a escribir al jugador
     */
    public void escribe( String pLinea ) throws Exception
    {
        //Si el socket sigue abierto, escribe el mensaje por el stream de salida
        if( socket != null )    o.escribeLinea( pLinea );
    }

    /**
     * Metodo de lectura
     * lee del stream de lectura asignado a este jugador.
     * @return String mensaje leido del stream de entrada asignado a este
jugador
     */
    public String lee() throws Exception
    {
        //Intenta leer del stream de entrada, bloqueando del flujo normal de
ejecucion
        return i.leeLinea();
    }
```

```
    /**
     * Metodo de ejecucion
     * Mediante el método run de este Thread se pueden recibir mensajes
     * desde el otro lado del socket asociado a este. Representa la mitad de la
conexion.
     * Es el flujo normal de ejecucion de este hilo. Es bloqueante ya que hasta
que no
     * ha leido algun mensaje no sigue con el flujo normal de ejecucion.
     */
    public void run()
    {
        try {

            //Si el socket esta abierto y esta asignado
            if( socket != null ) {

                while( !socket.isClosed() ) {

                    //Lee el mensaje de este jugador (en el otro extremo)
                    String mensaje = lee();

                    //Mensaje de peticion de comienzo de partida
                    if( mensaje.equals( Mensajes.COMIENZA_PARTIDA) ) {
                        //Pone el flag de comenzar partida a verdadero
                        comenzarPartida = true;
                    }

                    //Mensaje de peticion de recepcion de dados cifrados
                    if( mensaje.equals( Mensajes.DADOCIFRADO ) ) {
                        //Pone el valor del dado cifrado recogido del stream de
entrada.
                        //Primero lo descodifica desde Base64.
                        dado.setValorCifrado( new Base64().decode( lee() ) );
                    }

                    //Mensaje de peticion de recepcion de clave de dados
                    if( mensaje.equals( Mensajes.DADOCLAVE ) ) {
                        //Pone el valor de la clave recogida del stream de
entrada.
                        //Primero la descodifica desde Base64.
                        dado.setValorClave( new Base64().decode( lee() ) );
                    }
                }
            }

        //Si ha habido error
        } catch( Exception e1 ) {
            //Mensaje de cierre de comunicaciones
            Juego.interfaz.escribeMensajeError( "La comunicacion con el jugador
" + getNombre() + " se ha perdido, error: "+e1.getMessage() );
            Juego.interfaz.escribeMensajeError( "Se cerrará la comunicacion con
este jugador!" );
            //Cierra la comunicacion con el otro extremo (socket, input y
output)
            try{
                i.close();
                o.close();
                socket.close();
            } catch( Exception e2 ) {
                //Error al cerrar la comunicacion
                Juego.interfaz.escribeMensajeError( "No se puede cerrar la
comunicacion con el jugador. Terminar la partida por posibles inestabilidades
en el juego!" );
            }
        }
    }
```

```
    /**
     * Metodo de lectura.
     * Obtiene el numero de turnos sin tirar de este jugador
     * @return int numero de turnos sin tirar
     */
     public int getTurnosSinTirar()
     {
       return turnosSinTirar;
     }

    /**
     * Metodo de escritura.
     * Actualiza el numero de turnos sin tirar de este jugador
     * @param int numero de turnos sin tirar
     */
     public void setTurnosSinTirar( int pValor )
     {
       turnosSinTirar = pValor;
     }

    /**
     * Metodo de lectura.
     * Obtiene si ha de volver a tirar este jugador.
     * @return boolean si ha de volver a tirar, devuelve cierto. Falso en otro
caso.
     */
     public boolean getVuelveATirar()
     {
       return vuelveATirar;
     }

    /**
     * Metodo de escritura.
     * Actualiza el flag de volver a tirar.
     * @param boolean flag de volver a tirar.
     */
     public void setVuelveATirar( boolean pValor )
     {
       vuelveATirar = pValor;
     }

    /**
     * Metodo de lectura.
     * Obtiene el valor del flag de comenzar la partida.
     * Este flag se actualiza si el jugador del otro extremo le avisa del
comienzo de la partida.
     * @return boolean verdadero si le ha llegado el aviso de comenzar la
partida.
     */
     public boolean isComenzarPartida()
     {
       return comenzarPartida;
     }

    /**
     * Metodo de escritura.
     * Borra los valores del dado de este jugador. Limpia estos valores para el
siguiente lanzamiento
     * del dado.
     */
    public void resetDado()
    {
       dado.resetValorCifrado();
       dado.resetValorClave();
    }

    /**
     * Metodo de lectura.
```

```
     * Obtiene el dado de este jugador.
     * @return Dado dado del jugador.
     */
    public Dado getDado()
    {
        return dado;
    }

    /**
     * Metodo de escritura.
     * Actualiza el valor del flag de ganador de la partida.
     * @param boolean verdadero si se quiere hacer ganador a este jugador,
falso en otro caso.
     */
    public void setGanador( boolean pValor )
    {
        ganador = pValor;
    }

    /**
     * Metodo de lectura.
     * Obtiene el valor del flag que indica si este jugador a ganado la
partida.
     * @return boolean si el jugador es ganador, devuelve verdadero, falso en
otro caso.
     */
    public boolean isGanador()
    {
        return ganador;
    }

    /**
     * Metodo de lectura.
     * Obtiene la representacion en un String del este jugador.
     * El String contendrá el nombre del jugador, la direccion IP, el puerto,
la casilla y el turno.
     * @return String representacion en un String del jugador.
     */
    public String toString()
    {
        return getNombre()+"
["+getDireccionIP()+":"+String.valueOf(getPuerto()+"] Casilla "+getCasilla()+"
Turno "+getTurno() );
    }
}
```

### 15.6.9 Clase Mensajes.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

/**
 * Clase que inicializa constantes de mensajes de peticion de accion entre
jugadores.
 */
public class Mensajes
{
   //Define las constantes de mensajes de conexion y peticion de accion
   public static final String OK                 = "OK";
//Mensaje de OK
   public static final String NOK                = "NOK";
   //Mensaje de No OK
   public static final String NUEVO_JUGADOR      = "NUEVO";
   //Peticion de nuevo jugador
   public static final String INVITADO_JUGADOR   = "INVITADO";
   //Peticion de jugador invitado
   public static final String COMIENZA_PARTIDA = "COMIENZAPARTIDA";
   //Peticion de comienzo de partida
   public static final String DADOCIFRADO      = "DADOCIFRADO";
   //Valor de dado cifrado
   public static final String DADOCLAVE        = "DADOCLAVE";        //Valor
de clave de cifrado
}
```

### 15.6.10 Clase Partida.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.io.*;
import java.util.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;

/**
 * Clase que representa una partida del juego de la OCA.
 * Junto con la clase Juego (principal) es la clase de mas importancia en el
juego.
 * Esta clase contiene la informacion de la partida en juego, numero de
jugadores de la partida,
 * el jugador local a este ordenador, variables para establecer comunicaciones
y recibir
 * conexiones a través del protocolo SSL y flags de estado de la partida.
 * Su funcionamiento es vital para la aplicacion. Se puede inicializar esta
clase si se inicia
 * una partida (como iniciador) o enrolandose a una ya inicializada.
 * Proporciona metodos para esperar conexiones de jugadores entrantes.
 */
public class Partida
{
    //Jugador local
    private Jugador miJugador;
    //Lista de jugadores activos
    private List jugadores;
    //Numero de jugadores que hay en activo
    private int numeroDeJugadores;
    //Turno de juego
    private int turno;
    //Variables para la seguridad SSL
    private SecureRandom numeroAleatorioSeguro;
    private KeyStore almacenPrivado;
    private KeyStore almacenPublico;
    private SSLContext contextoSSL;
    //Socket servidor
    private SSLServerSocket socketServidor;
    //Indicador de jugador iniciador.
    private boolean iniciada;
    //Indicador de partida finalizada
    private boolean finalizada;

    /**
     * Metodo constructor.
     * Inicia una partida.
     * Inicializa el array de jugadores de la partida, en principio vacios y
pone a 0 el numero de jugadores.
     * Inicializa los almacenes de claves privada y publica de este jugador.
Los almacenes deben
     * contener la clave privada de este jugador y la clave publica con la que
se identifica hacia los
     * demas jugadores, así como las claves publicas de todos los demas
jugadores (para hacer la autenticacion).
     * Crea un numero aleatorio seguro para inicializar el contexto SSL.
     * Inicializa el contexto SSL con los certificados y claves
privadas/publicas.
     * Crea un socket servidor para esperar conexiones de jugadores remotos y
crea al jugador local con
     * el turno numero 0.
     * @param String nombre del jugador local
```

```
     * @param int puerto desde donde se escuchan conexiones
     * @param String fichero que contiene las claves privadas y publicas del
cliente y servidor,
     *        para cuando se actua de cliente y cuando se actua de servidor.
     * @param String tipo del fichero anterior
     * @param String password del fichero anterior
     * @param String fichero que contiene la clave publica y certificado del
cliente y servidor,
     *        para cuandi se actua como cliente y como servidor.
     * @param String tipo del fichero anterior
     * @param String password del fichero anterior
     */
    public Partida( String nombreLocal,
                    int puertoLocal,
                    String pfPrivado,
                    String ptPrivado,
                    String ppPrivado,
                    String pfPublico,
                    String ptPublico,
                    String ppPublico ) throws Exception
    {

        //Inicializa los flags y variables de la partida
        iniciada = true;              //Flag de partida iniciada por el jugador
local (si el jugador local es iniciador)
        finalizada = false;           //Flag de partida finalizada
        turno = 0;                       //Turno actual de juego (en este caso,
ahora tiene el turno el jugador con turno 0)

        //Inicializa el array de jugadores conectados
        jugadores = new ArrayList();  //Nuevo array
        numeroDeJugadores = 0;        //Numero de jugadores a 0

        //Crea un numero aleatorio seguro para contexto SSL
        numeroAleatorioSeguro = new SecureRandom();
        numeroAleatorioSeguro.nextInt();

        //Setup los almacenes con claves privadas y publicas
        almacenPrivado = KeyStore.getInstance( ((ptPrivado == null) ? "JKS" :
ptPrivado) );
        almacenPublico = KeyStore.getInstance( ((ptPublico == null) ? "JKS" :
ptPublico) );
        almacenPrivado.load( new FileInputStream( ((pfPrivado == null) ?
"almacenprivado.jks"  : pfPrivado) ), ((pfPrivado == null) ? "password" :
ppPrivado).toCharArray() );
        almacenPublico.load( new FileInputStream( ((pfPublico == null) ?
"almacenpublico.jks" : pfPublico) ), ((pfPublico == null) ? "password" :
ppPublico).toCharArray() );

        //Setup de las factorias
        //Trust manager para los certificados firmados por una CA de la clave
publica
        TrustManagerFactory tmf = TrustManagerFactory.getInstance( "SunX509" );
        tmf.init( almacenPublico );
        //Key Manager para las claves privada/publica del jugador
        KeyManagerFactory kmf = KeyManagerFactory.getInstance( "SunX509" );
        kmf.init( almacenPrivado , ((pfPrivado == null) ? "password" :
ppPrivado).toCharArray() );

        //Inicializa conexto SSL
        contextoSSL = SSLContext.getInstance( "SSL" );
        contextoSSL.init( kmf.getKeyManagers(), tmf.getTrustManagers(),
numeroAleatorioSeguro );

        //Setup del server socket
        SSLServerSocketFactory sf = contextoSSL.getServerSocketFactory();

        //Crea al jugador local
```

```
        miJugador = new Jugador( nombreLocal,
InetAddress.getLocalHost().getHostAddress(), puertoLocal, null, null, null, 0
);

        //Crea el socket servidor
        socketServidor = (SSLServerSocket)sf.createServerSocket( puertoLocal );
    }

    /**
     * Metodo constructor.
     * Se conecta a una partida existente.
     * Inicializa el array de jugadores de la partida, en principio vacios y
pone a 0 el numero de jugadores.
     * Inicializa los almacenes de claves privada y publica de este jugador.
Los almacenes deben
     * contener la clave privada de este jugador y la clave publica con la que
se identifica hacia los
     * demas jugadores, así como las claves publicas de todos los demas
jugadores (para hacer la autenticacion).
     * Crea un numero aleatorio seguro para inicializar el contexto SSL.
     * Inicializa el contexto SSL con los certificados y claves
privadas/publicas.
     * Crea un socket servidor para esperar conexiones de jugadores remotos y
crea al jugador local con
     * el turno numero 0.
     * @param String nombre del jugador local
     * @param int puerto desde donde se escuchan conexiones
     * @param String fichero que contiene las claves privadas y publicas del
cliente y servidor,
     *        para cuando se actua de cliente y cuando se actua de servidor.
     * @param String tipo del fichero anterior
     * @param String password del fichero anterior
     * @param String fichero que contiene la clave publica y certificado del
cliente y servidor,
     *        para cuandi se actua como cliente y como servidor.
     * @param String tipo del fichero anterior
     * @param String password del fichero anterior
     * @param String direccion IP donde conectarse
     * @param int puerto donde conectarse (donde el servidor remoto escucha)
     */
    public Partida( String nombreLocal,
                    int puertoLocal,
                    String pfPrivado,
                    String ptPrivado,
                    String ppPrivado,
                    String pfPublico,
                    String ptPublico,
                    String ppPublico,
                    String pDireccionIP,
                    int pPuerto ) throws Exception
    {

        //Indicador de iniciador de partida
        iniciada = false;
        finalizada = false;
        turno = 0;

        //Inicializa el array de jugadores conectados
        jugadores = new ArrayList();
        numeroDeJugadores = 0;

        //Escribe mensaje por la interfaz de conexion SSL
        Juego.interfaz.escribeMensajeSistema( "Preparando conexion SSL..." );

        //Crea un numero aleatorio seguro
        numeroAleatorioSeguro = new SecureRandom();
        numeroAleatorioSeguro.nextInt();
```

```
        //Setup los almacenes con claves privadas y publicas
        almacenPrivado = KeyStore.getInstance( ((ptPrivado == null) ? "JKS" :
ptPrivado) );
        almacenPublico = KeyStore.getInstance( ((ptPublico == null) ? "JKS" :
ptPublico) );
        almacenPrivado.load( new FileInputStream( ((pfPrivado == null) ?
"almacenprivado.jks"  : pfPrivado) ), ((pfPrivado == null) ? "password" :
ppPrivado).toCharArray() );
        almacenPublico.load( new FileInputStream( ((pfPublico == null) ?
"almacenpublico.jks" : pfPublico) ), ((pfPublico == null) ? "password" :
ppPublico).toCharArray() );

        //Setup de las factorias
        TrustManagerFactory tmf = TrustManagerFactory.getInstance( "SunX509" );
        tmf.init( almacenPublico );
        KeyManagerFactory kmf = KeyManagerFactory.getInstance( "SunX509" );
        kmf.init( almacenPrivado , ((pfPrivado == null) ? "password" :
ppPrivado).toCharArray() );

        //Inicializa conexto SSL
        contextoSSL = SSLContext.getInstance( "SSL" );
        contextoSSL.init( kmf.getKeyManagers(), tmf.getTrustManagers(),
numeroAleatorioSeguro );

        //Inicializa las factorias para el socket servidor y cliente
        SSLServerSocketFactory sf = contextoSSL.getServerSocketFactory();
        SSLSocketFactory sfc = contextoSSL.getSocketFactory();

        //Añade al jugador local
        miJugador = new Jugador( nombreLocal,
InetAddress.getLocalHost().getHostAddress(), puertoLocal, null, null, null, -1
);

        //Escribe mensaje por la interfaz
        Juego.interfaz.escribeMensajeSistema( "Conectando con la direccion: " +
pDireccionIP );

        //Variables auxiliares
        String nom, dir, prt, tur;
        String nom2, dir2, prt2, tur2;

        //Se crea un socket seguro a traves del protocolo SSL hacia la direccion
indicada
        SSLSocket nuevoSocket = (SSLSocket)sfc.createSocket( pDireccionIP,
pPuerto );

        //Se crean los streams de comunicacion
        StreamEntrada i = new StreamEntrada( nuevoSocket.getInputStream() );
        StreamSalida  o = new StreamSalida( nuevoSocket.getOutputStream() );

        //Escribe mensaje por la interfaz
        Juego.interfaz.escribeMensajeSistema( "Enviando informacion de mi
jugador..." );

        //Protocolo de comunicacion de nuevo usuario, se envia mensaje de
peticion de nuevo jugador
        o.escribeLinea( Mensajes.NUEVO_JUGADOR );

        //Envia informacion de mi jugador
        o.escribeLinea( miJugador.getNombre() );
        o.escribeLinea( miJugador.getDireccionIP() );
        o.escribeLinea( String.valueOf( miJugador.getPuerto() ) );

        //Avisa al jugador por medio de la interfaz
        Juego.interfaz.escribeMensajeSistema( "Recibiendo informacion del
jugador conectado..." );

        //Recibe informacion del jugador remoto
```

```
        nom = i.leeLinea();
        dir = i.leeLinea();
        prt = i.leeLinea();
        tur = i.leeLinea();

        //Avisa al jugador por medio de la interfaz
        Juego.interfaz.escribeMensajeSistema( "Recibiendo informacion de los
jugadores de la partida..." );

        //Leemos la informacion de los jugadores conectados al remoto
        int enrolados = Integer.parseInt( i.leeLinea() );
        for( int j = 0 ; j < enrolados; j++ )
        {
            //Lee los datos del jugador de la lista del remoto
            nom2 = i.leeLinea();
            dir2 = i.leeLinea();
            prt2 = i.leeLinea();
            tur2 = i.leeLinea();

            //Establece un socket hacia el jugador
            SSLSocket invitadoSocket = (SSLSocket)sfc.createSocket( dir2,
Integer.parseInt( prt2 ) );

            //Se crean los streams de comunicacion
            StreamEntrada i2 = new StreamEntrada(
invitadoSocket.getInputStream() );
            StreamSalida  o2 = new StreamSalida(
invitadoSocket.getOutputStream() );

            //Escribe el mensaje de jugador invitado
            o2.escribeLinea( Mensajes.INVITADO_JUGADOR );

            //Envia informacion de mi jugador
            o2.escribeLinea( miJugador.getNombre() );
            o2.escribeLinea( miJugador.getDireccionIP() );
            o2.escribeLinea( String.valueOf( miJugador.getPuerto() ) );

            //Creamos al nuevo jugador
            numeroDeJugadores++;
            jugadores.add( new Jugador(nom2, dir2, Integer.parseInt( prt2 ),
invitadoSocket, i2, o2, Integer.parseInt( tur2 ) ) );
        }

        //Creamos al nuevo jugador
        numeroDeJugadores++;
        Jugador nuevoJugador = new Jugador(nom, dir, Integer.parseInt( prt ),
nuevoSocket, i, o, Integer.parseInt( tur ) );
        jugadores.add( nuevoJugador );

        //Avisa al jugador remoto que hemos acabado de conectarnos con todos los
jugadores de la partida
        nuevoJugador.escribe( Mensajes.OK );

        //Establecemos el turno, que será el último
        miJugador.setTurno( numeroDeJugadores );

        //Setup del socket servidor
        socketServidor = (SSLServerSocket)sf.createServerSocket( puertoLocal );

        //Avisa al jugador de que se ha conectado a la partida
        Juego.interfaz.escribeMensajeSistema( "Protocolo finalizado. Partida
creada." );
    }

    /**
     * Metodo que espera conexiones al socket servidor creado, un tiempo
establecido.
     * Si se conecta algun jugador, lo devuelve, si no devuelve null.
```

```
     * Cuando algun jugador se conecta, lee el mensaje que le han enviado:
     * NUEVO_JUGADOR : Un nuevo jugador se quiere conectar, para eso:
     *   Se lee la informacion del jugador conectado.
     *   Se envia informacion del jugador local.
     *   Se envia informacion de los jugadores que hay conectados a la partida
(lista de jugadores conectados)
     *   Se crea al jugador que se acaba de conectar asignandole el nuevo
turno.
     * JUGADOR_INVITADO : Un jugador que ya posee la lista de jugadores de la
partida, me quiere conocer:
     *   Se lee la informacion del jugador conectado.
     *   Se crea al jugador asignandole el turno adecuado.
     */
    public Jugador esperaJugador( int pTiempo ) throws Exception
    {
        Jugador nuevoJugador = null;

        //Aceptamos jugadores, durante el tiempo de espera
        socketServidor.setSoTimeout( pTiempo );

        try {

            //Variables auxiliares
            String nom, dir, prt, tur;

            //Acepta conexiones entrantes
            SSLSocket socketNuevoJugador = (SSLSocket)socketServidor.accept();

            Juego.interfaz.escribeMensajeSistema( "Se esta conectando un nuevo
jugador..." );

            //Stream de entrada
            StreamEntrada i = new
StreamEntrada(socketNuevoJugador.getInputStream());
            StreamSalida o  = new
StreamSalida(socketNuevoJugador.getOutputStream());

            //Lee el mensaje que ha recibido
            String comando = i.leeLinea();

            //Se atiende la peticion de nuevo jugador
            if( comando.equals( Mensajes.NUEVO_JUGADOR ) ) {

                //Si el jugador local es el iniciador de la partida
                if( iniciada )
                {

                    Juego.interfaz.escribeMensajeSistema( "Es un jugador nuevo..."
);

                    //Recibe la informacion del jugador remoto
                    nom = i.leeLinea();
                    dir = i.leeLinea();
                    prt = i.leeLinea();

                    Juego.interfaz.escribeMensajeSistema( "Enviando informacion de
mi jugador.." );

                    //Envia informacion de mi jugador
                    o.escribeLinea( miJugador.getNombre() );
                    o.escribeLinea( miJugador.getDireccionIP() );
                    o.escribeLinea( String.valueOf( miJugador.getPuerto() ) );
                    o.escribeLinea( String.valueOf( miJugador.getTurno() ) );

                    Juego.interfaz.escribeMensajeSistema( "Enviando informacion de
los jugadores enrolados..." );

                    //Envia informacion de mi jugadores activos
```

```
                o.escribeLinea( String.valueOf( numeroDeJugadores ) );

                //Enviamos informacion de todos los jugadores conectados
                for( int iterador = 0 ; iterador < numeroDeJugadores ;
iterador++ )
                {
                    //Enviamos informacion del jugador Conectado
                    o.escribeLinea(
((Jugador)jugadores.get(iterador)).getNombre() );
                    o.escribeLinea(
((Jugador)jugadores.get(iterador)).getDireccionIP() );
                    o.escribeLinea( String.valueOf(
((Jugador)jugadores.get(iterador)).getPuerto()) );
                    o.escribeLinea( String.valueOf(
((Jugador)jugadores.get(iterador)).getTurno()) );
                }

                //Creamos al nuevo jugador
                numeroDeJugadores++;
                nuevoJugador = new Jugador(nom, dir, Integer.parseInt( prt ),
socketNuevoJugador, i, o, numeroDeJugadores);
                jugadores.add( nuevoJugador );

                Juego.interfaz.escribeMensajeSistema( "Esperando a que el
jugador se ponga\nen contacto con los demas jugadores..." );

                //Hasta que el jugador al que se le ha pasado la lista de
jugadores de la partida, no se
                //ponga en contacto con todos los demás, no se puede seguir,
ya que si no existe la posibilidad
                //de que el jugador local quiera comenzar la partida y se
bloqueen las comunicaciones
                //while( !nuevoJugador.lee().equals( Mensajes.OK ) ) {}

                Juego.interfaz.escribeMensajeSistema( "Nuevo jugador
enrolado!" );

            } else {

                //Envia no conforme, si se lee un mensaje de nuevo jugador y
el local no es iniciador de
                //partida
                o.escribeLinea( Mensajes.NOK );
            }

        }

        //Se atiende la peticion de jugador invitado por alguien
        if( comando.equals( Mensajes.INVITADO_JUGADOR ) ) {

            Juego.interfaz.escribeMensajeSistema( "Recibiendo informacion de
nuevo jugador invitado..." );

            //Recibe informacion del jugador invitado
            nom = i.leeLinea();
            dir = i.leeLinea();
            prt = i.leeLinea();

            //Creamos al nuevo jugador
            numeroDeJugadores++;
            nuevoJugador = new Jugador(nom, dir, Integer.parseInt( prt ),
socketNuevoJugador, i, o, numeroDeJugadores);
            jugadores.add( nuevoJugador );

            Juego.interfaz.escribeMensajeSistema( "Jugador invitado
enrolado!" );
        }
```

```
        } catch( SocketTimeoutException ste ) {
            //Si ha pasado el tiempo de espera de conexiones devuelve null
            return null;
        }
        //Devuelve al jugador que se acaba de conectar
        return nuevoJugador;
    }

    /**
     * Metodo de lectura.
     * Devuelve un array con todos los jugadores de esta partida.
     * @return Jugador[] array de jugadores de la partida
     */
     public Jugador[] getJugadores() throws Exception
     {
        //Crea un array con el numero de jugadores conectados contando con el
        //jugador local
        Jugador[] aux = new Jugador[ jugadores.size()+1 ];
        //Inserta el jugador local
        aux[0] = miJugador;
        //Recorre la lista de jugadores y lo añade al array
        for( int i = 0 ; i < jugadores.size(); i++ )
            //Si el socket del jugador sigue activo
            if( !((Jugador)jugadores.get(i)).getSocket().isClosed() ) aux[i+1] =
(Jugador)jugadores.get(i);
            //Si se ha perdido la comunicacion se elimina al jugador
            else  eliminaJugador( i );
        return aux;
     }

    /**
     * Metodo de lectura.
     * Obtiene al jugador local de esta partida.
     * @return Jugador jugador local
     */
     public Jugador getJugadorLocal() throws Exception
     {
        return miJugador;
     }

    /**
     * Metodo de lectura.
     * Devuelve al jugador que tenga el turno de la partida. Depende de la
propiedad de clase "turno".
     * @return Jugador jugador con el turno actual.
     */
    public Jugador getJugadorConTurno() throws Exception
    {
        //inicializa un jugador que contendrá al jugador con el turno
        Jugador jugadorConTurno = null;
        //Mientras no se encuentre al jugador
        while( jugadorConTurno == null )
        {
            //Mira quien de los jugadores conectados tiene el turno
            for( int i = 0 ; i < jugadores.size(); i++ )
                //Si el socket no esta cerrado
                if( !((Jugador)jugadores.get(i)).getSocket().isClosed() )
                {
                    //Si el turno del jugador coincide con el turno de la partida
                    if( ((Jugador)jugadores.get(i)).getTurno() == turno )
                    {
                        jugadorConTurno = (Jugador)jugadores.get(i);
                        break;
                    }
                //Si se ha perdido la comunicacion se elimina al jugador
                } else  eliminaJugador( i );

            //Mira si el turno lo tiene el jugador local
```

```
            if( miJugador.getTurno() == turno )
                jugadorConTurno = miJugador;

            //El jugador no esta en la lista (se ha perdido la comunicacion)
            if( jugadorConTurno == null )
                cambiaTurno();
        }
        return jugadorConTurno;
    }

    /**
     * Metodo de escritura.
     * Cambia el turno de juego. El turno va rotando dependiendo del numero de
jugadores
     * activos en la partida. Cuenta con los jugadores desconectados, ya que si
un jugador
     * tiene un turno mayor a uno que se ha perdido, no tendría nunca el turno
de juego.
     */
    public int cambiaTurno() throws Exception
    {
        //se hace un modulo numerodejugadores+2 porque se cuenta el jugador
local
        turno = (turno + 1) % (numeroDeJugadores+1);
        //Devuelve el turno calculado
        return turno;
    }

    /**
     * Metodo de escritura.
     * Envia a todos los jugadores de esta partida un mensaje (String)
     * @param String mensaje a enviar a todos los jugadores.
     */
    public void enviaATodos( String pValor ) throws Exception
    {
        //Para todos los jugadores activos en la partida
        for( int i = 0 ; i < jugadores.size(); i++ )
            //Si el socket no esta cerrado, no se ha perdido la comunicacion con
el jugador
            if( ! ((Jugador)jugadores.get(i)).getSocket().isClosed() )
                ((Jugador)jugadores.get(i)).escribe( pValor );
            //Si se ha perdido la comunicacion se elimina al jugador
            else  eliminaJugador( i );
    }

    /**
     * Mensaje de lectura.
     * Lee de todos los jugadores un mensaje. Hasta que no se lee el mensaje
esperado
     * de todos los jugadores no vuelve al flujo normal de ejecucion.
     * @param String el mensaje que se quiere leer de los jugadores
     */
    public void recibeDeTodos( String pValor ) throws Exception
    {
        //Si el mensaje que se quiere leer es el valor del dado cifrado
        if( pValor.equals( Mensajes.DADOCIFRADO ) )
        {
            //Para todos los jugadores de la partida
            for( int i = 0 ; i < jugadores.size(); i++ )
                //Si el socket sigue abierto
                if( !((Jugador)jugadores.get(i)).getSocket().isClosed() )
                    //Mientras no este establecido el valor del dado cifrado de
este jugador
                    while( !((Jugador)jugadores.get(i)).getDado().isValorCifrado()
) {}
                //Si se ha perdido la comunicacion se elimina al jugador
                else  eliminaJugador( i );
        }
```

```
        //Si el mensaje que se quiere leer es la clave utilizada para cifrar el
dado
        if( pValor.equals( Mensajes.DADOCLAVE ) )
        {
            //Recorre todos los jugadores de la partida
            for( int i = 0 ; i < jugadores.size(); i++ )
                //Si el socket sigue abierto
                if( !((Jugador)jugadores.get(i)).getSocket().isClosed() )
                    //Leer mientras no este establecido el valor de la clave
utilizada por jugador
                    while( !((Jugador)jugadores.get(i)).getDado().isValorClave() )
{}
                //Si se ha perdido la comunicacion se elimina al jugador
                else  eliminaJugador( i );
        }
    }

    /**
     * Metodo de escritura.
     * Borra los valores del dado de todos los jugadores.
     */
    public void resetDadoDeJugadores() throws Exception
    {
        //Recorre la lista de todos los jugadores
        for( int i = 0 ; i < jugadores.size(); i++ )
            //Si el socket sigue abierto
            if( ! ((Jugador)jugadores.get(i)).getSocket().isClosed() )
            {
                //Borra los valores del dado del jugador
                ((Jugador)jugadores.get(i)).resetDado();
            }
            //Si se ha perdido la comunicacion se elimina al jugador
            else eliminaJugador( i );
    }

    /**
     * Metodo de borrado.
     * Elimina al jugador con indice determinado y reasigna turnos de juego de
cada jugador.
     * @param int indice del array que contiene al jugador a eliminar
     */
    public void eliminaJugador( int indice ) throws Exception
    {
        //Eliminamos al jugador
        jugadores.remove(indice);
        //Enseña quien hay conectado por la interfaz del juego
        Juego.interfaz.showQuienHayConectado( getJugadores() );
    }

    /**
     * Metodo de lectura.
     * @return verdadero, si el jugador local es el iniciador de la partida.
Falso en otro caso.
     */
    public boolean isIniciada()
    {
        return iniciada;
    }

    /**
     * Metodo de lectura.
     * @return boolean verdadero, si la partida a finalizado. Falso en otro
caso.
     */
    public boolean isFinalizada()
    {
        return finalizada;
    }
```

```
    /**
     * Metodo de escritura.
     * Establece el flag de partida finalizada.
     * @param boolean valor del flag de partida finalizada.
     */
    public void setFinalizada( boolean pValor )
    {
        finalizada = pValor;
    }

    /**
     * Metodo de calculo.
     * Calcula el valor del dado de la partida, entre todos los jugadores.
     * Descifra el dado de todos los jugadores y lo suma junto con el dado del
jugador
     * local. Después hace un modulo 6 para calcular el valor del dado
conjunto.
     */
    public int calculaDadoConjunto() throws Exception
    {
        //Pone a 0 el valor acumulado del dado
        int valorAcumulado = 0;
        //Recorre la lista de jugadores activos
        for( int i = 0 ; i < jugadores.size(); i++ )
            //Si el socket esta abierto
            if( !((Jugador)jugadores.get(i)).getSocket().isClosed() )
            {
                //Suma el acumulado del dado, descifrando el dado de este jugador
                valorAcumulado +=
((Jugador)jugadores.get(i)).getDado().descifraDado();
            }
            //Si se ha perdido la comunicacion se elimina al jugador
            else  eliminaJugador( i );

        //Suma el valor del dado del jugador local
        valorAcumulado += miJugador.getDado().getValorEnClaro();
        //Hace un modulo 6.
        valorAcumulado = valorAcumulado%6;

        //Retorna el valor calculado, teniendo en cuenta que si el resultado es
0, representa un 6.
        return ( ((valorAcumulado == 0) ? 6 : valorAcumulado) );
    }
}
```

### 15.6.11 Clase StreamEntrada.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.io.*;

/**
 * Clase que Representa un flujo de entrada de comunicaciones. Subclase de
BufferedReader, con lo
 * que hereda todos sus metodos.
 * Crea un metodo de lectura.
 */
public class StreamEntrada extends BufferedReader
{
    /**
     * Metodo constructor.
     * Llama al metod constructor de la superclase, pasandole el InputStream de
entrada
     */
    public StreamEntrada( InputStream is )
    {
        super( new InputStreamReader(is) );
    }

    /**
     * Metodo de lectura.
     * Lee del flujo de entrada y devuelve el resultado como un String.
     * @return String valor leido de este flujo de entrada.
     */
    public String leeLinea( ) throws IOException
    {
        return readLine();
    }
}
```

### 15.6.12 Clase StreamSalida.

```
/**
 * Copyright Daniel Celemín Garcia 2004
 * Trabajo final de carrera UOC (Universitat Oberta de Catalunya)
 * Seguridad informática
 */

import java.io.*;

/**
 * Clase que Representa un flujo de salida de comunicaciones. Subclase de
DataOutoutStream, con lo
 * cual hereda todos sus metodos.
 * Crea un metodo de escritura, que escribe un String acabado con un salto de
linea.
 */
public class StreamSalida extends DataOutputStream
{
    /**
     * Metodo constructor.
     * Llama al metodo constructor de la superclase, pasandole el OutputStream
de salida
     */
    public StreamSalida( OutputStream os )
    {
        super( os );
    }

    /**
     * Metodo de escritura.
     * Escribe un String acabado en salto de linea por el flujo de salida.
     * @param String mensaje a escribir por el flujo de salida.
     */
    public void escribeLinea( String linea ) throws IOException
    {
        writeBytes( linea + "\n" );
    }
}
```