ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

PROYECTO FIN DE CARRERA

DESARROLLO DE UN VIDEOJUEGO MULTIJUGADOR BLUETOOTH PARA DISPOSITIVOS MÓVILES

INGENIERÍA DE TELECOMUNICACIÓN

MÁLAGA, 2011 ALBERTO MARTÍNEZ FERNÁNDEZ

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

Titulación: Ingeniería de Telecomunicación

Reunido el tribunal examinador en el día de la fecha, constituido por:

D./Dª. _____

D./Dª. _____

D./Dª. _____

para juzgar el Proyecto Fin de Carrera titulado:

DESARROLLO DE UN VIDEOJUEGO MULTIJUGADOR BLUETOOTH PARA DISPOSITIVOS MÓVILES

del alumno/a D. Alberto Martínez Fernández

dirigido por D. Francisco Javier González Cañete

ACORDÓ POR _____ OTORGAR LA

CALIFICACIÓN DE _____

Y, para que conste, se extiende firmada por los componentes del tribunal, la presente diligencia

Málaga, a _____ de _____ de _____

El/La Presidente/a El/La Vocal El/La Secretario/a

Fdo.: _____ Fdo.: _____ Fdo.: _____

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

DESARROLLO DE UN VIDEOJUEGO MULTIJUGADOR BLUETOOTH PARA DISPOSITIVOS MÓVILES

REALIZADO POR:

Alberto Martínez Fernández

DIRIGIDO POR:

Francisco Javier González Cañete

DEPARTAMENTO DE: Tecnología Electrónica.

TITULACIÓN: Ingeniería de Telecomunicación.

PALABRAS CLAVE: Aplicaciones móviles, videojuego, Arkanoid, J2ME, Bluetooth, multihilo.

RESUMEN:

Debido al auge que está experimentando el sector del ocio interactivo, así como a la creciente implantación de terminales móviles con mayores prestaciones y numerosas posibilidades de conectividad, se plantea el presente proyecto, cuyo objetivo es desarrollar un videojuego multijugador mediante Bluetooth para dispositivos móviles basado en el título comercial para un solo jugador, "Arkanoid". La idea de juego consiste en eliminar ladrillos manejando una nave que puede desplazarse lateralmente para hacer rebotar una pelota que impactará sobre ellos para destruirlos. Adicionalmente, se cuenta con una serie de power-ups, o potenciadores, para alcanzar este objetivo. Para esta versión, se contemplan dos modos de juego: uno cooperativo en el que los dos jugadores colaborarán en la consecución de un objetivo común; y otro no cooperativo, o deathmatch, donde cada jugador buscará sumar la mayor puntuación posible y evitar la continuidad del otro en la partida. El videojuego ha sido desarrollado empleando la tecnología J2ME de Java.

Málaga, Diciembre de 2011

Agradecimientos

Las primeras líneas de este apartado son de obligada dedicatoria a mi familia, especialmente a mis padres, Felipe y Mari Carmen, por habérmelo dado todo en esta vida, por ser el mejor ejemplo de trabajo, sacrificio y honradez y por haber creído en mí y en mis posibilidades más que yo mismo en determinados momentos.

También quiero expresar mi agradecimiento a los que habéis colaborado de forma desinteresada con vuestras sugerencias y ayudas para que este proyecto llegara finalmente a buen puerto, en especial a Silvia y Antonio. De igual modo, quisiera agradecer a mi tutor, Francis, su apoyo y su disponibilidad durante el desarrollo de este proyecto.

Y a todos mis amigos y compañeros, que habéis sabido estar ahí y apoyarme, pero en especial a unos cuantos que, aun a riesgo de olvidarme de alguien en la siempre incómoda tarea de enumerarlos, considero justo que sean mencionados en estas líneas:

A Lidia, por tu paciencia, por todas esas horas aguantándome, en los buenos momentos y en los no tan buenos, por tener siempre un hueco para mí.

A Alejandro Blanco, periodista, amante del deporte, gran comunicador, mejor persona. Gracias por ser especial y ser una inspiración en todo lo que haces.

A Macarena, por dejarme entrar en tu vida, por confiar en mí y hacerme sentir alguien importante.

A José Manuel del Pino, compañero al que he tenido la gran suerte de conocer en mis últimos años de universidad, noble y humilde como muy pocos.

A mis italianos, en especial a Anna Gangemi, Giuseppe, Claudia, Natalia y Salvo, por esa amistad que nos une y hace que me sienta próximo a vosotros a pesar de la distancia. Dentro de muy poco volveremos a disfrutarla juntos.

Arduo hallarás pasar
sobre el agudo filo de la navaja
y penoso es, dicen los sabios,
el camino de la salvación
Upanishad Kathara

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1: INTRODUCCIÓN

La industria de los videojuegos es una de las más activas y en auge de todas las del sector del entretenimiento. Iniciada por Atari, en 1972, con la publicación de un sencillo juego de tenis de mesa conocido como "Pong" [1] y popularizada en 1978 por Taito, con el lanzamiento al mercado del título "Space Invaders", puede decirse que, salvo la excepción de la crisis del sector en 1983, debida al exceso de producción [2], apenas ha parado de crecer año tras año, convirtiéndose en la actualidad en una de las industrias mejor consolidadas en España, siendo el cuarto país de Europa –y sexto del mundo– en ventas.

En la actualidad, la aDeSe (Asociación española de Distribuidores y Editores de Software de Entretenimiento) [3], cuyos miembros representan el 80% del negocio de los videojuegos en España, recoge en su informe anual de 2010 que el sector de los videojuegos facturó 1.245 millones de euros, una cifra un 5.2% menor a la registrada en 2009, pero con la que España mantiene su posición como cuarto mercado europeo en consumo. Con estas cifras, la industria del videojuego genera más beneficios que la de la música y el cine juntas [4], concretamente un 53% en 2009, frente al 47% restante, repartido entre cine en DVD y Blu-Ray, entradas de cine y discos de música.

Para poder mantener este ritmo de crecimiento, el sector ha tenido que ir adaptándose y modernizándose a la misma velocidad que evolucionan las nuevas tecnologías y las tendencias de un mercado tan cambiante como el del entretenimiento.

La omnipresencia de los terminales móviles en nuestras vidas cotidianas ha sido vista por parte de los desarrolladores como una buena oportunidad para explotar sus capacidades cada vez mayores de ejecutar videojuegos y expandir su mercado a este tipo de dispositivos. Lejos ya de los antiguos terminales con pantallas monocromo y pocas líneas de texto simultáneas, los móviles actuales tienen pantallas de mayor resolución con miles de colores, así como múltiples posibilidades de conectividad (Wi-Fi, Bluetooth, etc.) que permiten que el jugar ya no tenga que ser una experiencia

solitaria. De hecho, 4 de cada 10 usuarios de móvil se conectan a Internet por Wi-Fi en España. Esto supone un incremento del 17% frente a 2010 [5].

Además, en la actualidad no es necesario acudir a las gamas más altas (y, por tanto, más costosas) de los fabricantes para encontrarnos con terminales de esas características. El tiempo medio de vida de un móvil es de 9 meses en Japón, 15 meses en Europa y 18 meses en EEUU [6]. Tenemos una sociedad que mantiene actualizado constantemente su teléfono a los últimos modelos, y que incluso los dispositivos que se adquieren sin desembolso de dinero para el usuario, ya vienen equipados con muchas de las últimas novedades y mejoras en conectividad.

La variedad de títulos disponibles para teléfonos móviles actualmente es muy elevada, viéndose beneficiada de la multitud de consolas portátiles existentes (PSP, Nintendo DS, N–Gage) y la similitud de desarrollar para ese tipo de plataformas con limitaciones. Podemos encontrar títulos de muy diversos géneros, desde juegos arcade, de estrategia o simulación, pasando por juegos de aventura.

Los arcades se dieron a conocer de la mano de las máquinas recreativas de videojuegos, las cuales llegaron a gozar de cierta popularidad durante la década de los ’70 y los ’80, disponibles en los salones recreativos, bares, restaurantes y centros comerciales. La primera máquina arcade de monedas fue “Computer Space”, lanzada en noviembre de 1971 por Nutting Associates. Creada por Nolan Bushnell y Ted Dabney, que más tarde fundarían Atari gracias al dinero que obtuvieron al descubrir este negocio [7]. Un año más tarde, en 1972, lanzan desde Atari la recreativa “Pong”, ya mencionada en párrafos anteriores. El éxito fue rotundo. En la figura 1.1 se puede ver el aspecto que presentaba este videojuego.

Figura 1.1. Imagen del videojuego Pong

Al ver el mercado que estaban generando estas máquinas de videojuegos, muchas compañías se sumaron para incrementar sus ingresos. En 1974, Atari, junto a Kee Games, presentan "Tank", el primer videojuego que almacenaba gráficos en chips de memoria ROM. En 1975 salió "Gunfight", el primer juego japonés licenciado para Estados Unidos en el que Midway rediseñó la máquina original incluyendo por primera vez un microprocesador. El juego trataba de una lucha con pistolas entre dos vaqueros al más puro estilo de las películas del oeste.

Steve Jobs y Steve Wozniak diseñan "Breakout" para Atari un poco antes de fundar Apple. El juego es una versión del "Pong", que se convertiría en el predecesor del "Arkanoid". Consiste en destruir los bloques de situados en la parte superior de la pantalla con la ayuda de una pelota y de una paleta situada en el extremo inferior que simulaba una raqueta de front-tenis que el jugador podía desplazar de izquierda a derecha para evitar que la pelota cayera por la parte inferior de la pantalla. La figura 1.2 muestra una captura de pantalla de "Breakout".

Figura 1.2. Imagen del videojuego Breakout para Atari 2600

La recreativa del "Breakout" fue lanzada al mercado en 1976 [8], y dos años después, en 1978, se distribuyó la versión para la videoconsola Atari 2600 –posteriormente, en 1983, también se lanzaría la versión para MSX–. Su funcionamiento era simple: comenzaba con ladrillos dispuestos en ocho filas y un color por cada dos filas (de abajo a arriba: amarillo, verde, naranja y rojo). A base de rebotes con las paredes y con la paleta manejada por el jugador, éste tenía que eliminar el máximo número posible de ladrillos. Si la pelota salía de la pantalla –por el extremo inferior, sin que el jugador pudiera hacer que rebotara en la paleta–, se perdía una vida. Inicialmente, el jugador disponía de 3 vidas para completar dos pantallas de ladrillos. Las puntuaciones por cada ladrillo eliminado iban acorde a su ubicación en la pantalla, y por

tanto, a la dificultad para alcanzar a un ladrillo de ese tipo. Así, por eliminar un ladrillo amarillo, se obtenía 1 punto, por uno verde 3, por uno naranja 5 y por uno rojo 7. Además, la velocidad de la pelota se incrementa tras efectuar cuatro y ocho impactos, y tras hacer contacto con las filas naranja y roja, y la nave se encoge hasta la mitad de su tamaño original una vez la pelota ha alcanzado la pared de la parte superior de la pantalla –habiéndose abierto hueco entre los ladrillos para llegar hasta ahí–.

El éxito de "Breakout" se tradujo en el desarrollo de su secuela dos años después para la consola Atari 5200: "Super Breakout". A diferencia de su predecesor, éste fue un videojuego basado en microprocesador –"Breakout" se implementó con puertas lógicas–, por lo que es un juego que puede ser emulado con el programa de código abierto y gratuito MAME (*Multiple Arcade Machine Emulator*) [9] [10].

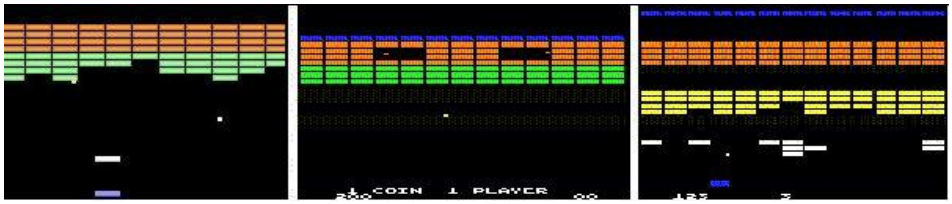
"Super Breakout" mostraba un aspecto en esencia similar a su predecesor, aunque aportó una serie de nuevos modos de juego entre los que poder elegir para comenzar una partida:

- Double: consiste en utilizar dos paletas al mismo tiempo, una situada encima de la otra, y aportando cada una una pelota. Se considera que se pierde una vida cuando ambas pelotas se pierden –es decir, se salen de la pantalla por el extremo inferior de la misma– y la puntuación se duplica mientras el jugador es capaz de mantener en juego las dos pelotas, evitando que ninguna de ellas se pierda.

- Cavity: mantiene el modo de juego original de "Breakout", con su pelota y su paleta, pero hay otras dos pelotas confinadas entre ladrillos que el jugador debe liberar. La puntuación se triplicará mientras el jugador sea capaz de mantener libres las tres pelotas.

- Progressive: al igual que el modo de juego *Cavity*, utiliza una pelota y una paleta, pero los bloques de ladrillos descienden gradualmente con los rebotes de la pelota sobre la paleta, de manera que se acercan progresivamente a ésta.

La figura 1.3 muestra la introducción de los nuevos modos de juego en "Super Breakout".



(a) **(b)** **(c)**

Figura 1.3. Imágenes de los 3 nuevos modos de juego para Super Breakout (a) Double, (b) Cavity, (c) Progressive

Posteriormente, a mediados de los años '90, para la consola Atari Jaguar apareció una nueva versión, conocida como "Breakout 2000", que básicamente consistía en una adaptación en 3D del juego original. El juego incluía algunas características que se darían a conocer algunos años antes con el "Arkanoid", tales como los *power-ups* o potenciadores, ladrillos irrompibles, rompibles tras varios impactos… pero también aprovechaba las nuevas posibilidades que le brindaba el nuevo entorno tridimensional, como el uso de ladrillos apilados que ocupaban los huecos que iban dejando los ladrillos inferiores tras ser eliminados por la pelota.

"Breakout" llegó a lanzarse para los IBM PC y para la videoconsola Play Station, e incluso hay disponible una versión para iPhone y iPod Touch. Por su parte, "Super Breakout" se adaptó en 2010 para la consola Xbox 360 y para Windows PC. De esta saga de videojuegos surgieron multitud de variaciones y de clones, pero sin duda el que logró mayor calado y se asentó definitivamente en el mercado fue "Arkanoid" (figura 1.4), uno de los juegos insignia del mundo de los videojuegos, desarrollado por Taito en 1986 [11] [12].

Figura 1.4. Portada de Arkanoid para la versión de Atari ST

"Arkanoid" fue uno de los primeros arcades en incluir un pequeño argumento. En él, controlamos la nave *Vaus* en lugar de la paleta que controlábamos en "Breakout" y al iniciar una nueva partida aparece un pequeño texto introductorio en inglés, que traducido dice:

> *La era y el tiempo de esta historia son desconocidos. Después de que la nave nodriza "Arkanoid" fuera destruida, una cápsula "Vaus" se separó de ella, sólo para ser atrapada en el espacio por alguien…*

En esencia, los objetivos y el modo de juego siguen siendo los mismos que en "Breakout", pero ahora se trata de un título con un aspecto gráfico mucho más vistoso, con más niveles y al que se supo explotar la jugabilidad añadiendo nuevas funcionalidades con los *power-ups*.

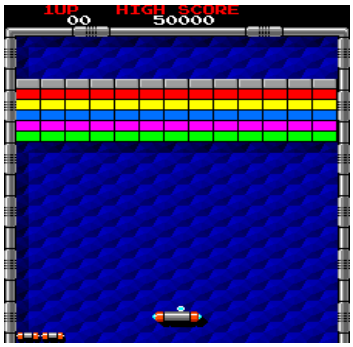


Figura 1.5. Imagen de una partida de Arkanoid

El videojuego original se componía de 33 niveles, cada uno con su propio estilo y distribución de los ladrillos en pantalla. En el último nivel, el jugador se enfrenta al principal enemigo del juego, Doh, al cual deberá vencer tras golpearle repetidas veces con la pelota. La figura 1.5 muestra el aspecto que presentaba el primer nivel.

Los ladrillos podían romperse al primer impacto o tras varios impactos, y también podían ser irrompibles. Algunos de los ladrillos rompibles desprendían una especie de cápsula de un determinado color tras ser destruidos por la pelota. Cada color indicaba el tipo de *power-up* asociado a la cápsula (el azul ensancha la nave, el rojo le proporciona capacidad de disparar proyectiles verticalmente, el rosa le permite pasar directamente al siguiente nivel, el celeste proporciona tres bolas, el plomo otorga una

vida extra al jugador, el verde captura la pelota en lugar de hacerla rebotar y el naranja ralentiza la velocidad de la pelota). Si la nave del jugador recogía la cápsula, adquiría el *power-up* asociado a ella, y su efecto le duraba mientras mantuviera al menos una pelota en juego o no recogiera otra cápsula que lo cancelara.

Además, a partir de cierto momento comienzan a aparecer en cada nivel por la parte superior de la pantalla unas figuras que representan enemigos, cuyo movimiento aleatorio podrá estorbar al jugador, modificando inesperadamente la trayectoria de la pelota al impactar contra alguno de ellos. Tras el impacto, mueren, pero por cada enemigo eliminado se incorpora otro nuevo desde el extremo superior.

La popularidad que adquirió el título hizo que rápidamente inundara el mercado de las computadoras de la época, la mayoría de 8 bits, como los *Atari 400/800*, *Apple II*, *ZX Spectrum*, y *MSX* en 1986, y *Amstrad CPC 464* y *Commodore 64* en 1987, y poco después haría lo propio en las de 16 bits, como *Commodore Amiga* y *Atari ST* en 1987 y *Apple IIGS* e *IBM-PC* en 1988. También se distribuyó una versión del juego original para NES (Nintendo Entertainment System).

Además, después de "Arkanoid" se desarrollaron otras tres secuelas para el mercado de las máquinas arcade: "Arkanoid – Revenge of Doh" en 1987 y diez años más tarde, en 1997, "Arkanoid – Doh it again" y "Arkanoid Returns", cada una con pequeñas novedades con respecto al título original, pero respetando siempre la idea y concepto de juego de la primera entrega. "Arkanoid – Revenge of Doh", por ejemplo, incluía ladrillos que se desplazaban y ladrillos que volvían a aparecer después de ser eliminados, además de nuevos *power-ups*, algunos de ellos no precisamente para ayudar al jugador, como el que reducía el tamaño de la nave. También añadía otros, como el de disponer de 8 pelotas, el de tener 3 pelotas *invencibles* –cuando una caía, aparecía otra nueva en pantalla, de manera que siempre había 3– o el de usar dos naves, una junto a la otra. "Arkanoid Returns", por su parte, renovó ligeramente el aspecto gráfico de las versiones anteriores y también añadió algún *power-up* nuevo, como el de la pelota que no rebota al colisionar con los ladrillos. Alguna de estas secuelas llegó también a las videoconsolas, como "Arkanoid – Doh it again", que se distribuyó para *Nintendo Super NES* en 1997. En la figura 1.6 se muestran capturas de pantalla de "Arkanoid – Revenge of Doh" y "Arkanoid Returns".

(a) **(b)**

Figura 1.6. Secuelas de Arkanoid

(a) Arkanoid – Revenge of Doh, (b) Arkanoid Returns

Posteriormente, "Arkanoid" siguió conquistando plataformas, lanzándose versiones para *Nintendo DS* ("Arkanoid DS"), *Xbox 360* ("Arkanoid Live") o *Nintendo Wii* ("Arkanoid Plus!"). Como particularidad, la versión para *Nintendo DS* permite el manejo de la nave a través del lápiz táctil proporcionado con la consola, por lo que, en la práctica, se puede regular la velocidad de desplazamiento de ésta, según la arrastremos con el lápiz con mayor o menor velocidad. Además, incluye más de 130 fases en el modo clásico y añade nuevos modos de juego, como el modo misión, donde se deben cumplir requisitos especiales para terminar las fases, un modo versus para jugar contra la CPU o contra uno o más amigos –permite hasta 4 jugadores– a través de Wi-Fi y la posibilidad de conectarse vía Wi-Fi para buscar nuevos retos [13].

En definitiva, "Arkanoid" ha sido y es uno de los videojuegos de mayor renombre y reconocido como un clásico del entretenimiento electrónico, con una larga lista de versiones, adaptaciones, clones y juegos inspirados en él a sus espaldas. Su sencilla idea de juego, su dificultad y la astucia a la hora de decidir qué *power-ups* utilizar en cada momento lo han convertido en uno de los títulos más adictivos.

En este proyecto se desarrollará una versión multijugador –en concreto, para 2 jugadores– de este videojuego, adaptada para dispositivos móviles. Se parte del "Arkanoid", tanto a nivel de lógica del juego como de estética, tomando como referencia su aspecto gráfico (la nave, los ladrillos, las paredes laterales, las cápsulas que contienen los *power-ups*…). Pero ahora, al ser multijugador, constará de dos naves, una abajo (controlada por el usuario local) y otra arriba (que responderá a las instrucciones recibidas desde el dispositivo remoto). Así pues, para que en ambos

terminales el usuario local siempre controle la nave del extremo inferior, la representación en pantalla en un terminal, corresponderá a la misma representación en el terminal remoto, pero invirtiendo el eje vertical. La aplicación necesitará ejecutarse al mismo tiempo desde dos terminales diferentes, por lo que será necesario un proceso de sincronización entre ellos para garantizar la coherencia en la ejecución del programa en los dos dispositivos.

El videojuego, cuyos textos y opciones de menú estarán traducidos al español, inglés e italiano, constará de dos modos básicos de juego: uno cooperativo y uno no cooperativo, o *deathmach*. En el modo cooperativo, cada jugador intentará mantener en juego en todo momento tanto su pelota como la del otro jugador, contribuyendo entre los dos a conseguir un objetivo común, el de completar tantos niveles como les sea posible. En el modo *deathmach*, por el contrario, los jugadores lucharán uno contra el otro, penalizando con una vida menos si la pelota del otro usuario rebota sobre la nave propia. En este modo, por tanto, no se trata de llegar lo más lejos posible, sino de durar más en la partida que el contrincante. Ambos modos se desarrollarán sobre un conjunto de 10 niveles, cada uno con una disposición diferente para los ladrillos. Al terminar el décimo, se continúa por el primer nivel de nuevo, y así sucesivamente hasta que la partida concluya, bien por el agotamiento de las vidas por parte de alguno de los jugadores, bien porque alguno de ellos haya decidido abandonar la partida.

Asimismo, el videojuego incluye, como ocurre en el "Arkanoid" original, una serie de *power-ups* que podrán servir de ayuda al jugador o podrán complicarle la tarea. Podrán hacer que la pelota se adhiera a la nave –en lugar de hacerla rebotar–, aumentar o reducir el tamaño de la nave o la rapidez en sus desplazamientos, dotar a la nave de capacidad para disparar, otorgar vidas extra, acelerar o frenar la velocidad de la pelota, triplicar el número de pelotas o cancelar el rebote de la pelota tras el impacto con un ladrillo. Todos ellos se analizarán con mayor detenimiento en el capítulo de *Diseño de la Aplicación*.

El presente documento está estructurado en diversos capítulos para poder explicar con detalle todas las partes del proyecto:

1. Introducción: es el presente capítulo, en el que se hace una somera descripción y justificación del proyecto, así como de los objetivos a conseguir.

2. Tecnologías y herramientas empleadas: en este capítulo se detallan las herramientas software y hardware utilizadas, comparándolas con otras similares y justificando su elección en base a sus ventajas e inconvenientes.

3. Descripción de la aplicación: aquí se expone en profundidad la lógica de la aplicación, aclarando el funcionamiento de los menús, las diferentes opciones y la lógica del juego.

4. Diseño de la aplicación: en este apartado se explica en detalle cómo se ha implementado la aplicación para lograr la funcionalidad deseada.

5. Conclusiones y líneas futuras: en el último capítulo se hace un balance del proyecto, así como el estudio de posibles mejoras o ampliaciones.

CAPÍTULO 2: TECNOLOGÍA Y HERRAMIENTAS EMPLEADAS

2.1. INTRODUCCIÓN

A lo largo de este capítulo se detallarán las tecnologías que se han estimado oportunas para la realización de este proyecto, así como las herramientas software que se han utilizado, la labor para las que fueron empleadas y el motivo de su elección frente a otras alternativas.

Cabe reseñar que todas las herramientas que se emplearon en el presente proyecto son de libre uso y distribución, siguiendo la filosofía general de búsqueda de la estandarización y el mínimo coste global, tanto para el desarrollador como para el cliente final.

2.2. JAVA

Java se creó como una herramienta de programación para ser usada en un proyecto de *set-top-box* (es el nombre con el que se conoce el dispositivo encargado de la recepción y decodificación de señal de televisión analógica o digital) en una pequeña operación denominada *the Green Project* en Sun Microsystems en el año 1991 [14]. Java tuvo unos comienzos dubitativos, hasta que en 1994, en Sun Microsystems comenzaron a ver a Internet como el medio interactivo que pensaban era la televisión por cable y reconocieron que la mejor forma de enfocar su desarrollo futuro era orientándolo hacia la Web. A partir de entonces, Java floreció como uno de los lenguajes con más implantación, gozando actualmente de una excelente salud y de infinidad de desarrolladores, herramientas y una base de código extensísima.

Las características que hacen de Java un lenguaje tan popular son:

- Es independiente de la plataforma. Es decir, una aplicación una vez escrita, puede ser ejecutada en cualquier dispositivo, tal como reza el axioma de Java: "*write once, run anywhere*". Esto es así porque cuando se compila una

aplicación Java, se genera un código intermedio llamado *bytecode*, que ha de ser interpretado más tarde por la Máquina Virtual Java (JVM – *Java Virtual Machine*), ésta sí dependiente del sistema operativo, presente en la gran mayoría de sistemas. En la figura 2.1 se muestra la diferencia entre la compilación tradicional dependiente de la máquina y la generación de los *bytecodes* multiplataforma.

- Seguridad: Esto es en parte debido a la generación de los *bytecodes*, ya que en Java, por su propio diseño, no se permite acceder a la máquina físicamente más que a través de librerías que sí dependen del sistema operativo y que no son estándares de Java.

- Es software libre, y los IDE (Entornos de Desarrollo Integrados, del inglés *Integrated Development Environment*) más conocidos para programar en Java, como NetBeans y Eclipse, son gratuitos.

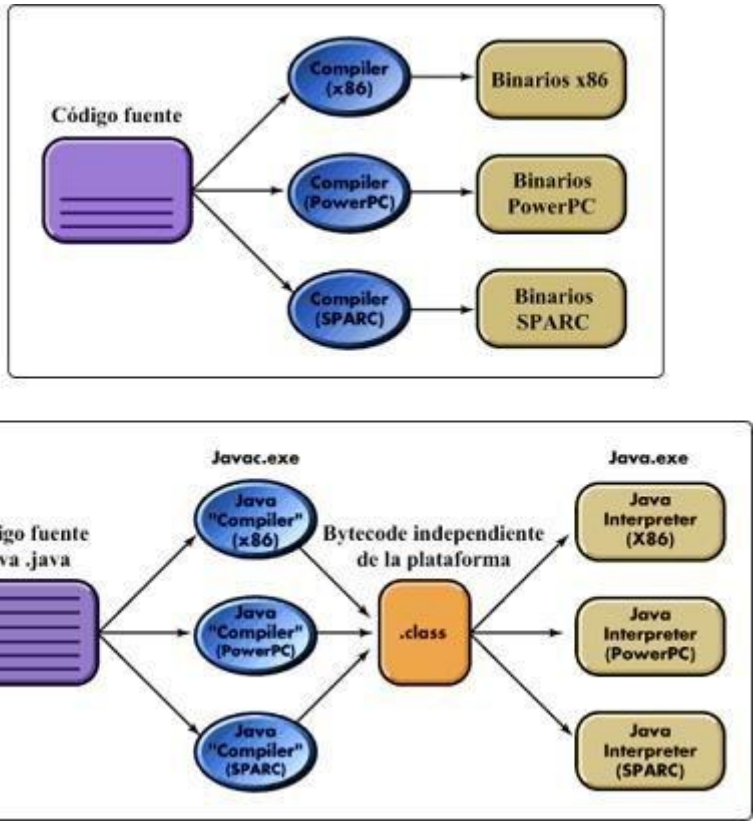


Figura 2.1. Diferencias entre los compiladores tradicionales y el compilador Java

2.3. J2ME

J2ME (*Java 2 Micro Edition*) fue diseñado por Sun Microsystems como un subconjunto de las especificaciones de Java, que definen una versión minimizada de la plataforma *Java 2 Standard Edition* (J2SE), que se ajusta especialmente bien a dispositivos con pocos recursos o con capacidades restringidas, como pueden ser teléfonos móviles, PDA (agendas electrónicas), o sistemas embebidos.

2.3.1. ARQUITECTURA J2ME: CONFIGURACIONES Y PERFILES

Como se acaba de comentar, J2ME está pensado para cubrir un gran número de dispositivos con características muy diversas, y, por lo tanto, demanda una arquitectura capaz de gestionar de forma adecuada las funcionalidades disponibles en cada familia de dispositivos que lo soportan. Debido a la diferencia de restricciones y capacidades en todos los dispositivos soportados, J2ME se estructura en configuraciones, perfiles y ciertos paquetes opcionales. Además de estos elementos, J2ME también incluye el propio lenguaje de programación, y una máquina virtual.

En la figura 2.2 se puede ver cómo se estructura en la actualidad la plataforma J2ME. En la base del diagrama, se sitúan dos configuraciones básicas, que trabajan sobre una máquina virtual:

- CLDC (*Connected Limited Device Configuration*) es la configuración estándar para los dispositivos inalámbricos con capacidades más limitadas, y también es la más extendida. Esta configuración proporciona un nivel mínimo de funcionalidades para desarrollar aplicaciones para un determinado conjunto de dispositivos como los teléfonos móviles o PDA de capacidades limitadas. Por tanto, CLDC provee de un conjunto de clases esenciales para construir aplicaciones para este tipo de dispositivos. En este caso, se trabaja sobre una máquina virtual llamada KVM (*Kilobyte Virtual Machine*), que es más pequeña y por lo tanto menos potente que la dedicada a entornos mayores (JVM).

- CDC (*Connected Device Configuration*) es otra configuración pensada para dispositivos con una capacidad de proceso relativamente mayor (por

ejemplo, un mínimo de 2 megabytes de memoria, así como capacidades de red más potentes que las correspondientes a CLDC). Además, en CDC y su perfil correspondiente, la máquina virtual que se utiliza es la misma que la utilizada en entornos superiores (JVM), como el ya nombrado J2SE y J2EE (*Java 2 Enterprise Edition*).

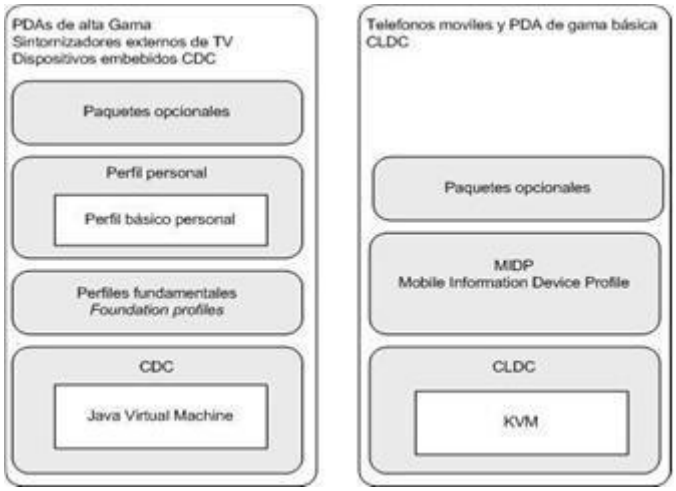


Figura 2.2. Plataforma J2ME, configuraciones y perfiles

Por encima de las configuraciones, están los perfiles. El perfil es un grupo más específico de API (del inglés *Application Programming Interface*, Interfaz de Programación de Aplicaciones) desde el punto de vista del dispositivo. Es decir, la configuración da cobertura a una familia de dispositivos más o menos amplia, y el perfil se orienta a un determinado tipo de dispositivos dentro de esa familia. El perfil añade una serie de funcionalidades, además de las que ya aporta la configuración, por lo que al final, entre unas aportaciones y otras, se tiene un conjunto de funcionalidades adecuadas al grupo específico de dispositivos en cuestión. Cabe reseñar que las aplicaciones desarrolladas sobre un determinado perfil van a ser portables a cualquier dispositivo que soporte ese perfil. Sobre una configuración pueden existir diversos perfiles.

Sobre CDC se encuentran los perfiles FP (*Foundation Profiles*, Perfiles Fundacionales), que son los perfiles de más bajo nivel, y están pensados para elementos con capacidades de red pero sin interfaz de usuario, por lo que son especialmente adecuados para sistemas embebidos. Estos perfiles se pueden combinar con el que tienen justo por encima en el diagrama, el perfil PP (*Personal Profile*, Perfil Personal), para dispositivos que requieran de una interfaz gráfica de usuario. Además, hay

disponible también una versión reducida del perfil PP, que se denomina PBP (*Personal Basic Profile*, Perfil Básico Personal) y está diseñado para un entorno de trabajo que necesite un nivel básico de presentación gráfica.

Por otra parte, sobre CLDC se presenta la especificación MIDP (*Mobile Information Device Profile*, Perfil para Dispositivos de Información Móviles), que está especialmente pensada para trabajar sobre teléfonos móviles y PDA. Dentro de la especificación, se define un dispositivo MIDP como un dispositivo pequeño de recursos limitados, móvil y con una conexión inalámbrica. Es decir, las funcionalidades que provee hacen referencia a conexiones de red limitadas en velocidad y estabilidad, interfaz de usuario reducida y almacenamiento local de información en el dispositivo.

2.3.2. MIDLETS

Será sobre este último, el perfil MIDP, y su configuración subyacente (CLDC), sobre los que se trabaje en el presente proyecto. El motivo es que es el perfil que está disponible en teléfonos móviles y PDA, y a día de hoy es el perfil más extendido e implantado. La última versión disponible de la especificación MIDP es la 3.0, aunque buscando una mayor compatibilidad en la aplicación desarrollada se ha utilizado la versión MIDP 2.0. En esta versión se incluyen una serie de elementos exclusivos que hacen más fácil el desarrollo de videojuegos, que no estaban en las versiones anteriores.

2.3.2.1. MIDP 2.0

Los requisitos mínimos que ha de cumplir un dispositivo para poder ejecutar aplicaciones MIDP 2.0 son:

- Pantalla de, al menos, 96x54 píxeles, 1 bit (blanco y negro), proporción del aspecto del píxel (aproximadamente 1:1).
- Entrada por teclado o pantalla táctil.
- 256 KB de memoria no volátil para la aplicación MIDP (sin tener en cuenta la requerida por CLDC), 8 KB de memoria no volátil para los datos persistentes creados en la aplicación, y 128 KB de memoria volátil para el entorno de ejecución de la máquina virtual.
- Conexión bidireccional, acceso inalámbrico posiblemente no permanente, con ancho de banda limitado.
- Capacidad para reproducir tonos, ya sea por hardware o software.

En la figura 2.3 se muestra la arquitectura de alto nivel que provee MIDP 2.0.

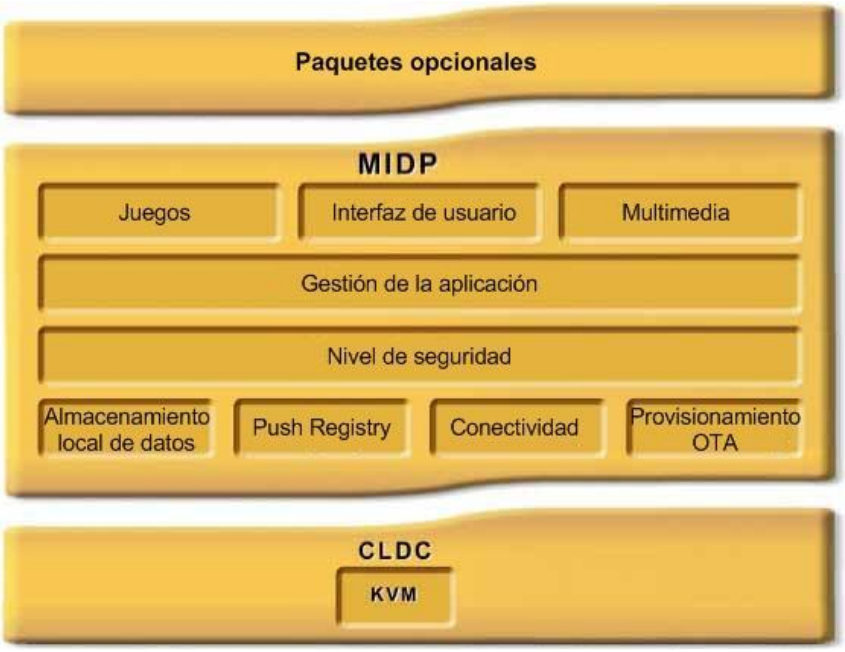


Figura 2.3. Arquitectura de alto nivel de MIDP 2.0

Las aplicaciones J2ME que han sido desarrolladas bajo la especificación MIDP se denominan *MIDlets*. Las clases de un *MIDlet* se almacenan en el formato definido por los *bytecodes* Java dentro de un fichero *.class*. Estas clases han de pasar por un proceso de preverificación en tiempo de desarrollo que garantice que no realizarán ninguna operación no permitida cuando estén funcionando en el terminal móvil. Este proceso no se puede hacer en tiempo de ejecución por las limitaciones inherentes a la máquina virtual que se utiliza (KVM) que, liberada de esta tarea, pasa a ser más pequeña y eficiente. La preverificación se realiza después de la compilación, y el resultado es la nueva clase.

2.3.2.2. Estructura de ficheros

Los *MIDlets* están pensados para poder ser descargados a través de Internet. Este tipo de descarga recibe el nombre de OTA (*Over The Air*), y está compuesta por un archivo con extensión *.jar* (*Java Archive*) en el que se empaqueta la aplicación en sí (clases del *MIDlet*, clases de apoyo, recursos como imágenes o sonidos), así como un fichero denominado manifiesto; y un fichero descriptor con extensión *.jad* (*Java Application Descriptor*).

En el fichero de manifiesto (de extensión *.mf*), se describe el contenido del fichero *.jar*. En la figura 2.4 se muestra el contenido de un fichero de manifiesto.

```
MIDlet-1: BArkanoid, , principal.BArkanoid
MIDlet-Vendor: Vendor
MIDlet-Name: BArkanoid
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

Figura 2.4. Contenido de un fichero de manifiesto *.mf*

El programa encargado de manejar la descarga de *MIDlets* y de controlar los estados por los que éstos pasan mientras se están ejecutando en el dispositivo se denominan AMS (*Application Management Software*). Estos estados son los siguientes: Localización, Instalación, Ejecución, Actualización y Borrado.

El fichero descriptor *.jad* proporciona la información requerida por el AMS para llevar a cabo la descarga y asegurarse de que ese *MIDlet* es adecuado para el dispositivo en el que se quiere instalar. Es importante destacar que hay que tener cuidado con la procedencia de los ficheros *.jar* que se instalan en los dispositivos, ya que si ésta no es fiable, podrían contener código malicioso no verificado que escapase al control de la máquina virtual, pudiendo causar perjuicios al usuario (por ejemplo, mediante el envío de mensajes SMS no autorizados sin que el usuario se percate).

2.3.2.3. Ciclo de vida de un *MIDlet*

Durante su ciclo de vida activo, un *MIDlet* pasa por varios estados, desde que se crea hasta que se destruye, devolviendo todos los recursos que le había proporcionado el sistema. En la figura 2.5 se muestra un diagrama con estos estados y los métodos que regulan la transición de un estado a otro.

Los estados posibles en los que se puede encontrar un *MIDlet* son:

- Detenido (*Paused*), estado en el que se encuentra un *MIDlet* que ha sido creado, pero aún no ha sido ejecutado por primera vez el método *startApp()*. También se puede provocar este estado a través de una llamada al método *pauseApp()*. En este estado, el *MIDlet* mantiene los mínimos recursos posibles y admite cualquier modificación asíncrona.

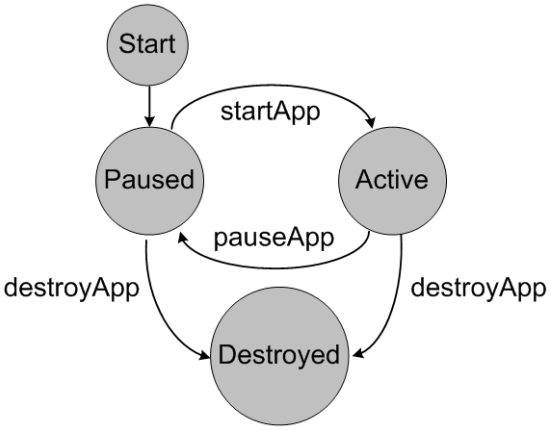


Figura 2.5. Ciclo de vida de un *MIDlet*

- Activo (*Active*), estado de ejecución del *MIDlet* en el que se entra tras la llamada a *startApp()*.
- Destruido (*Destroyed*), estado provocado por la invocación del método *destroyApp()*. Una vez que el *MIDlet* entra en este estado, no podrá hacer una transición a ningún otro estado, habrá terminado toda su actividad.

2.4. ENTORNOS DE DESARROLLO (IDE)

Un entorno de desarrollo integrado o IDE (*Integrated Development Environment*) es un programa compuesto por un conjunto de herramientas que tienen como misión facilitar la labor del desarrollo a un programador. Es un entorno de programación que incluye un editor de código, un compilador, un depurador y un constructor de la interfaz gráfica. Existen IDE configurables que soportan una variedad de lenguajes de programación. La elección de un IDE adecuado puede suponer la diferencia de permitir al programador concentrarse en el desarrollo de sus aplicaciones, en lugar de tener que lidiar con las particularidades del entorno.

Siguiendo la filosofía del presente proyecto en cuanto al uso de software libre siempre que sea posible, las alternativas disponibles que permiten integrar el desarrollo de aplicaciones Java y J2ME son:

- Eclipse [15]. Esta herramienta desarrollada originalmente por IBM, emplea módulos o *plugins* para proporcionar su funcionalidad, a diferencia de otros

entornos monolíticos en los que están incluidas todas las funcionalidades, las necesite el usuario o no. Eclipse soporta de forma nativa el desarrollo de aplicaciones Java. En cuanto al desarrollo de aplicaciones J2ME, lo soporta por medio de la instalación del *plugin* EclipseME.

- NetBeans [16]. Esta herramienta de Sun soporta el desarrollo de todos los tipos de aplicación Java (J2SE, J2EE y J2ME con el paquete opcional *Mobility Pack*). Está estructurada en módulos que posibilitan extender la funcionalidad cuando sea necesario, sin tener de entrada un entorno demasiado pesado.

Ambas plataformas conforman unos entornos de desarrollo maduros, con una gran base de usuarios que contribuyen a crear tutoriales, manuales y *plugins* para facilitar el desarrollo. Como ambos entornos se adecúan bastante bien a los requerimientos de este proyecto, la decisión entre uno u otro se basa en pequeños detalles, como por ejemplo la mayor facilidad de NetBeans para comenzar a utilizarse inmediatamente al proporcionar al usuario todo lo necesario en una única descarga, o algunos errores que se encontraron en Eclipse a la hora de añadir librerías externas a la aplicación. Es por ello que finalmente se eligió NetBeans como IDE para llevar a cabo el desarrollo de la aplicación.

2.5. EMULADORES

Una buena forma de detectar y corregir errores de código antes de empezar a probar la aplicación sobre móviles reales, es hacer uso de los emuladores. Si bien no garantizan que las aplicaciones funcionen exactamente igual que en los teléfonos móviles, sin su ayuda el desarrollo habría sido más lento y mucho más complicado. Gracias a los emuladores, es posible, entre otras cosas, hacer un seguimiento paso a paso de la ejecución del código del programa, lo cual facilita en gran medida su depuración.

Durante la elaboración de este proyecto, se ha hecho uso del emulador Sun Wireless Toolkit [17]. Este emulador viene proporcionado por Sun dentro del paquete en el que se distribuyen las herramientas necesarias para desarrollar aplicaciones MIDP

sobre la configuración CLDC. Al estar desarrollado por Sun en base a las especificaciones de MIDP, no tiene en cuenta las particularidades que se dan en los terminales físicos reales, por lo que no debe considerarse como fiable por completo, ya que puede ejecutar una aplicación sin ningún error, y sin embargo no está garantizado el correcto funcionamiento de ésta más tarde cuando se instala en un teléfono móvil. Por otra parte, una característica de este emulador es la de poder ejecutar múltiples instancias sobre la misma máquina, por lo que fue de gran ayuda para realizar las pruebas en las que fue necesario implicar a más de un terminal, como es el caso del juego multijugador. Por último, cabe destacar que su consumo de memoria es reducido, lo que facilita también la ejecución de múltiples instancias y la rapidez en la corrección de errores. Se puede ver una imagen de este emulador en la figura 2.6.

Figura 2.6. Emulador Sun Wireless Toolkit

2.6. BLUETOOTH

Bluetooth es la solución que se ha utilizado en el presente proyecto para dar soporte multijugador a la aplicación. Es una tecnología de radio de corto alcance, que

permite conectividad inalámbrica entre dispositivos remotos. Se diseñó pensando básicamente en tres objetivos: pequeño tamaño, mínimo consumo y bajo precio.

Opera en la banda libre de radio ISM (en inglés, *Industrial, Scientific and Medical*, banda industrial científico-médica) a 2.4 GHz. Su máxima velocidad de transmisión de datos es de 1 Mbps. El rango de alcance Bluetooth depende de la potencia empleada en la transmisión. La mayor parte de los dispositivos que usan Bluetooth transmiten con una potencia nominal de salida de 0 dBm, lo que permite un alcance de unos 10 metros en un ambiente libre de obstáculos.

El desarrollo de aplicaciones Bluetooth programando en Java es posible gracias a la especificación JSR 82 [18]. Ésta esconde la complejidad del protocolo Bluetooth detrás de unos *API's* que permiten centrarse en el desarrollo, en vez de los detalles de bajo nivel de Bluetooth.

JSR 82 intenta ofrecer las siguientes capacidades:

- Registro de servicios.
- Descubrimiento de dispositivos y servicios.
- Establecer conexiones *RFCOMM* (*Radio Frequency Communication*), *L2CAP* (*Logical Link Control and Adaptation Protocol*) y *OBEX* (*OBject EXchange*) entre dispositivos.
- Usar dichas conexiones para enviar y recibir datos (las comunicaciones de voz no están soportadas).
- Manejar y controlar las conexiones de comunicación.
- Ofrecer seguridad a dichas actividades.

2.7. GIMP

Para la edición de las imágenes que aparecen en el videojuego, ha sido necesario recurrir a un software específico para tal fin. Una vez más, se ha hecho uso de un programa libre y gratuito como es GIMP (*GNU Image Manipulation Program*) [19] –en concreto, la versión 2.6– disponible bajo la licencia pública general de GNU.

La primera versión de GIMP se desarrolló inicialmente en sistemas Unix [20] y fue pensada para GNU/Linux como una herramienta libre para trabajar con imágenes. No obstante, actualmente es el programa de manipulación de gráficos disponible en más sistemas operativos (los ya mencionados Unix y GNU/Linux, además de Microsoft Windows, Mac OS X, FreeBSD y Solaris, entre otros) y con el tiempo se ha convertido en una alternativa gratuita al popular Adobe Photoshop para un gran número de usos. De hecho, GIMP es también conocido por ser quizás la primera gran aplicación libre para usuarios no profesionales o expertos. Productos originados anteriormente, como GCC, el núcleo Linux, etc., eran principalmente herramientas de programadores para programadores. GIMP es considerado por algunos como una prueba de que el proceso de desarrollo de software libre puede crear aplicaciones que los usuarios comunes, no avanzados, pueden usar de manera productiva.

GIMP sirve para procesar gráficos y fotografías digitales. Los usos típicos –los empleados en la elaboración de este proyecto– incluyen la creación de gráficos y logos, el cambio de tamaño, recorte y modificación de fotografías digitales, la modificación de colores, la combinación de imágenes usando un paradigma de capas, la eliminación o alteración de elementos no deseados en imágenes o la conversión entre distintos formatos de imágenes. La figura 2.7 muestra una captura de pantalla durante la elaboración de las imágenes de las naves del videojuego que se ha desarrollado.

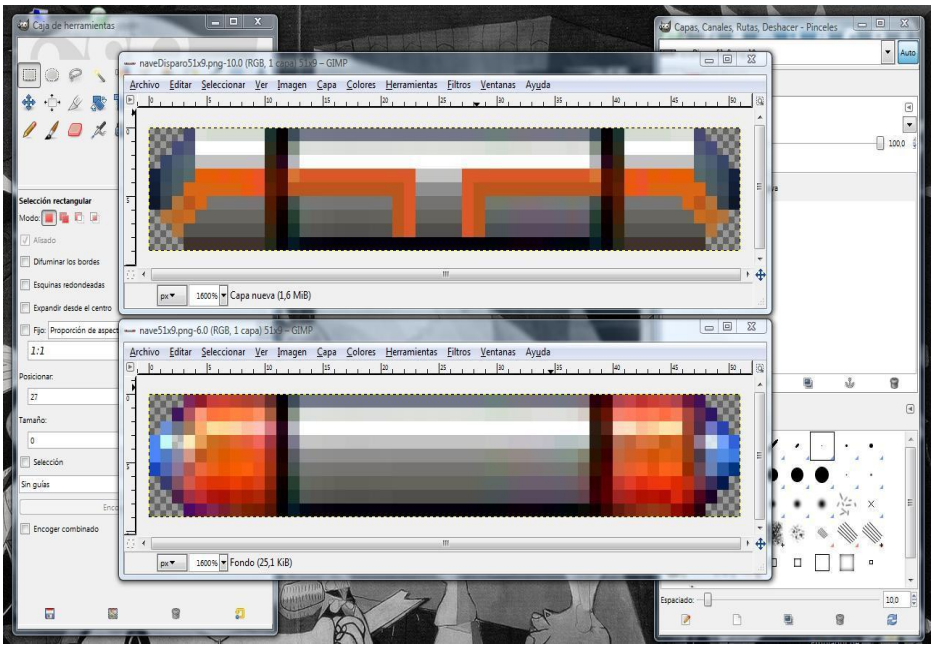


Figura 2.7. Edición de imágenes con GIMP

Por último, aunque ninguna de las siguientes características han sido necesarias para la realización de este proyecto, cabe señalar que este programa también permite crear imágenes animadas sencillas, la manipulación de vectores y la edición avanzada de video.

2.8. SOUNDTAP

Cualquier videojuego reciente para dispositivos móviles incorpora efectos de sonido de mayor o menor complejidad, pequeñas melodías o incluso los hay que incluyen hasta composiciones musicales creadas expresamente para ese título en concreto.

En el videojuego que se ha desarrollado en el presente proyecto se precisaba reproducir los efectos de sonido y la melodía del videojuego original en el que se basa, "Arkanoid". Para la incorporación de dichos efectos se ha utilizado una herramienta de grabación de sonido que permitiera recoger los diferentes sonidos presentes en "Arkanoid" y posteriormente separar cada uno de ellos y editarlos de manera independiente. SoundTap [21] es un programa de licencia gratuita, que registra en un fichero de audio cualquier sonido que se reproduzca en la tarjeta de sonido del equipo sobre el que se está ejecutando. Su facilidad de uso, su bajo consumo de memoria y, por supuesto, el hecho de ser un software gratuito han sido los factores que han influido decisivamente en la elección de esta herramienta. En la figura 2.8 puede verse el aspecto que presenta esta aplicación.

Figura 2.8. La herramienta de grabación de audio SoundTap

Normalmente, este tipo de programas permiten ser descargados junto con una o varias aplicaciones que complementan su funcionalidad. Para realizar el proceso de separación y edición de cada efecto de sonido registrado, se ha utilizado una de estas herramientas adicionales: WavePad Sound Editor. En la figura 2.9 puede verse una captura de pantalla de la ejecución de este programa.

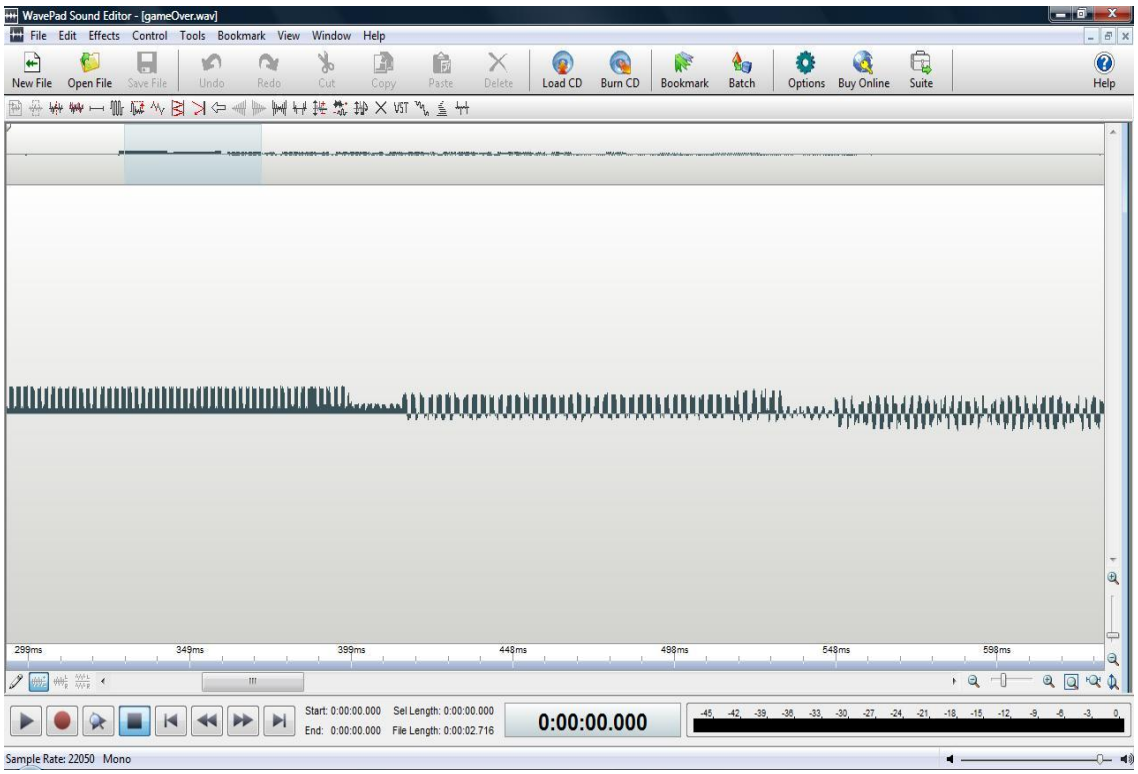


Figura 2.9. Editor de audio WavePad Sound Editor

Los sonidos han sido extraídos de la versión de “Arkanoid” para MAME [22].

2.9. M.A.M.E.

Como ya se comentó en el capítulo de introducción, las máquinas recreativas llegaron a ser bastante conocidas allá por los ’70 y ’80. Sin embargo, debido a la rentabilidad y avance de la tecnología en las consolas y ordenadores que han llegado a tener un hardware superior al arcade, las máquinas recreativas han ido perdiendo popularidad hasta casi desaparecer. A pesar de ello, la gran cantidad de nostálgicos y aficionados del género ha logrado mantener vivo el interés por experimentar aquella sensación de jugar en una máquina arcade, llegando incluso a construirlas ellos mismos

(existen diversas comunidades y páginas web dedicadas a esto), rehabilitando armazones o creándolos nuevos y habilitando un ordenador para simular la CPU de las viejas máquinas.

Otra opción menos laboriosa, aunque también más alejada del manejo de una máquina recreativa real, es la de emular su funcionamiento en un ordenador personal. Para ello, hay que recurrir a un emulador. MAME [9] es el emulador de máquinas recreativas más popular, es el acrónimo de *Multiple Arcade Machine Emulator* (en español, emulador de múltiples máquinas recreativas). Es un programa de código abierto y gratuito si se utiliza sin ánimo de lucro y actualmente emula la mayoría de los juegos de recreativas del siglo XX, en total más de 5.000 juegos distintos emulados, la mayoría en múltiples versiones [23]. MAME se desarrolla principalmente en versiones para Windows y DOS, pero existen versiones para otras plataformas, como Linux, Mac OS, AmigaOS o QNX, e incluso de forma no oficial para consolas como Nintendo DS, PlayStation 2 o Nintendo Wii. En la figura 2.10 se puede ver el aspecto que presenta su interfaz gráfica.

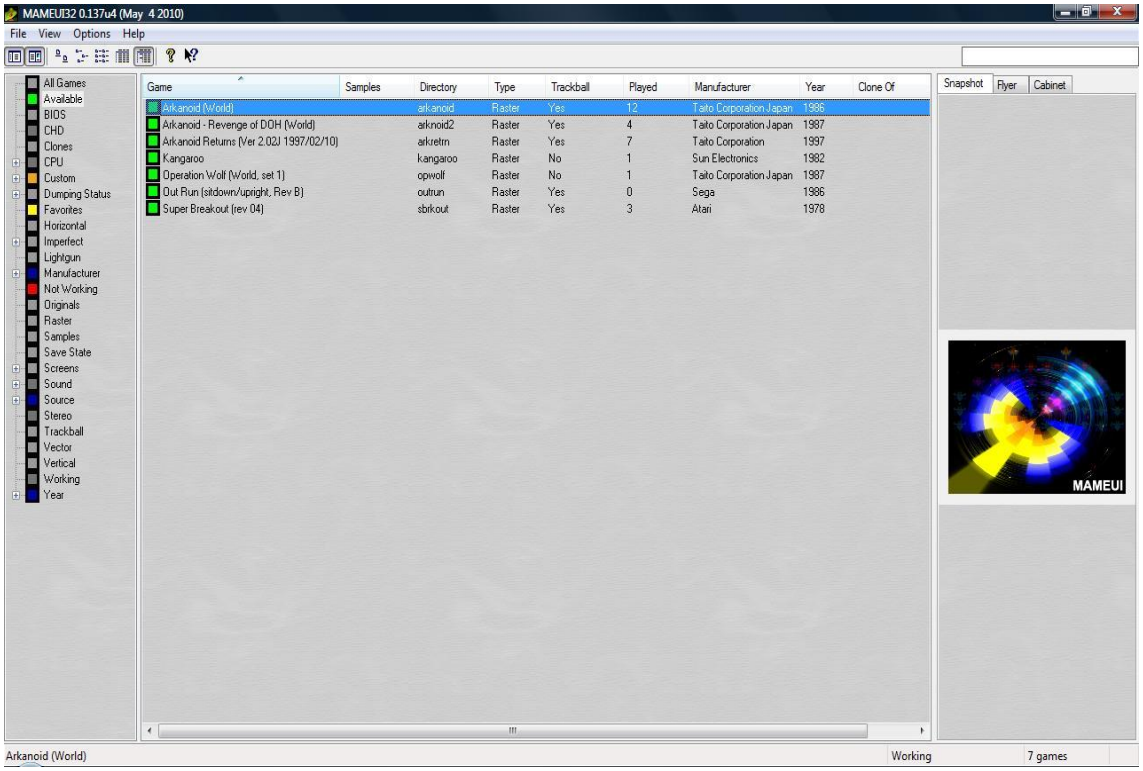


Figura 2.10. El emulador MAME

2.10. MODELADO UML

2.10.1. INTRODUCCIÓN

A la hora de afrontar el diseño de un sistema complejo, como primera opción se podría proceder sin más que realizar un simple análisis previo y corregir los eventuales errores de diseño conforme se hacen patentes sus consecuencias. El problema es que dichas correcciones no se resolverían hasta entrar ya en etapas más avanzadas, donde el coste que supondría rectificar sería demasiado elevado.

Por ello, resulta conveniente realizar una planificación previa un poco más elaborada, que permita hacernos una idea y visualizar cómo es o queremos que sea un sistema. Necesitamos, por tanto, *modelar* el sistema. En cualquier rama de la ingeniería, se ha encontrado útil la representación de los diseños de una forma gráfica. No obstante, la falta de estandarización en la manera de representar gráficamente un modelo impedía que los diseños gráficos ya realizados pudieran ser compartidos entre distintos diseñadores. Era necesario, por tanto, un lenguaje que no sólo permitiera comunicar ideas a otros diseñadores, sino que también sirviera de apoyo en los procesos de análisis de un problema. Con este objetivo se creó el Lenguaje Unificado de Modelado (UML, del inglés *Unified Modeling Language*).

UML se ha convertido en el estándar ansiado durante tanto tiempo por profesionales de la ingeniería, permitiendo representar y modelar la información con la que se trabaja en la fase de análisis y, especialmente, en la de diseño. Es un lenguaje gráfico que proporciona una notación muy expresiva que se usa para visualizar, especificar, construir y documentar un sistema.

Como su propio nombre indica, UML es un lenguaje de modelado. Un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema es capturar las partes esenciales del mismo. Para facilitar este modelado, el diseñador realiza una abstracción y la plasma en una notación gráfica, lo que se conoce como un modelado visual. Este modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar. Además, el modelado visual que proporciona es independiente del lenguaje usado en la implementación, de forma que los diseños realizados utilizando UML

pueden implementarse en cualquier lenguaje que soporte sus posibilidades (principalmente lenguajes orientados a objetos).

Siguiendo una definición estricta, UML es, ante todo, un lenguaje. Esto es, proporciona un vocabulario y unas reglas para permitir una comunicación. Este lenguaje ofrece las facilidades necesarias para crear y leer un modelo, pero no dice cómo crearlo, ya que de eso se han de ocupar las metodologías de desarrollo.

Además de todo lo anterior, UML es un método formal de modelado, por lo que permite un mayor rigor en la especificación y posibilita a su vez realizar una verificación y validación del modelo realizado. Incluso se pueden automatizar determinados procesos, de modo que se puede generar código a partir del modelo y viceversa, lo que permite que ambos estén en todo momento actualizados, con lo que se mantiene la visión de la estructura del proyecto desde el más alto nivel.

UML tiene una gran variedad de objetivos, aunque se pueden resumir así sus funciones:

- Visualizar. Posibilita expresar de una forma gráfica un sistema de forma que otro lo pueda entender.
- Especificar. Permite precisar cuáles son las características de un sistema antes de ser implementado.
- Construir. A partir de los modelos especificados, se pueden construir los sistemas diseñados.
- Documentar. Los propios elementos gráficos sirven como documentación del sistema desarrollado, y pueden servir para su futura revisión.

Un modelo UML está compuesto por tres clases de bloques de construcción:

- Elementos: son abstracciones de cosas reales o ficticias (objetos, acciones, etc).
- Relaciones: sirven para ligar los elementos entre sí.
- Diagramas: son colecciones de elementos con sus relaciones.

Por tanto, en un diagrama se tiene la representación gráfica de un conjunto de elementos con sus relaciones. Para poder representar correctamente el sistema, UML ofrece una amplia variedad de diagramas para visualizar el sistema desde distintas perspectivas. En la figura 2.11 se pueden ver los diversos tipos de diagrama que proporciona UML.

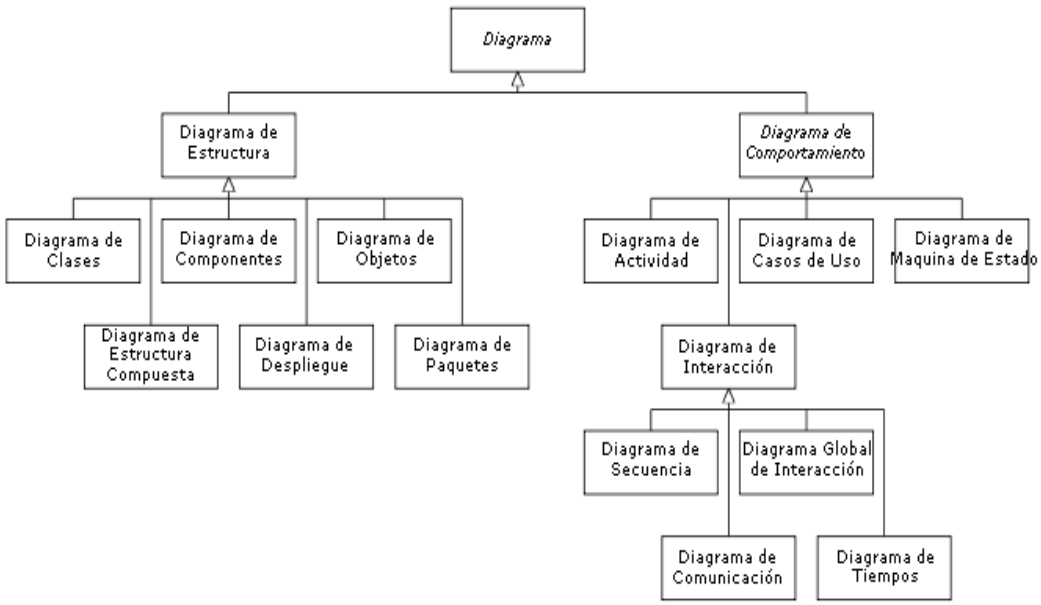


Figura 2.11. Tipos de diagramas UML

Los diagramas más interesantes (y por ello, los más usados) son los de clases, de estados y de secuencia. Debido a sus características particulares y a los diferentes puntos de vista que proponen, que se exponen a continuación, serán los diagramas que se utilizarán más adelante para realizar el diseño del videojuego que propone este proyecto.

2.10.2. DIAGRAMA DE CLASES

Un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos. Los diagramas de clases abarcan la vista de diseño estática de un sistema, como se puede ver en el ejemplo de un aeropuerto, en la figura 2.12.

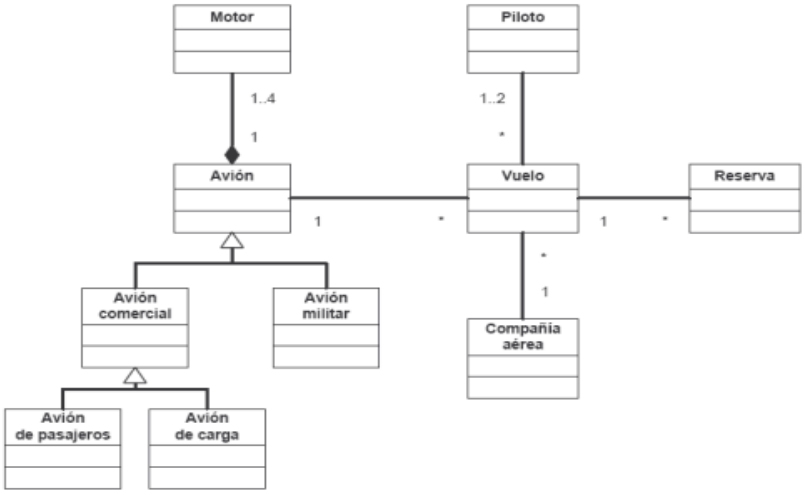


Figura 2.12. Diagrama de clases de un aeropuerto

2.10.3. DIAGRAMA DE ESTADOS

Los diagramas de estados muestran una máquina de estados y se utilizan para modelar los aspectos dinámicos de un sistema. Se pueden usar para modelar un caso de uso, una clase o un sistema completo. Estos diagramas pueden ser útiles para modelar la vida de un objeto, como se puede ver en la figura 2.13.

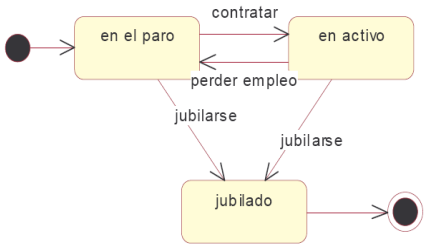


Figura 2.13. Diagrama de estados de la vida de un trabajador

Al tratarse del modelado de los comportamientos dinámicos de un sistema, la mayoría de las veces, esto supone modelar el comportamiento de objetos reactivos. Un objeto reactivo es aquel para el que la mejor forma de caracterizar su comportamiento es señalar cuál es su respuesta a los eventos lanzados desde fuera de su contexto. Estos objetos tienen un ciclo de vida bien definido, cuyo comportamiento se ve afectado por su pasado.

Estos diagramas también son importantes para construir sistemas ejecutables a través de ingeniería directa e inversa.

2.10.4. DIAGRAMA DE SECUENCIA

Los diagramas de secuencia son los diagramas más efectivos para modelar la interacción entre objetos en un sistema. El diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada método de la clase. Este diagrama contiene detalles sobre la implementación, incluyendo los objetos y las clases que se usan para implementar el escenario, así como los mensajes intercambiados entre los objetos. Estos mensajes se dibujan cronológicamente desde la parte superior del diagrama hasta la inferior. En la figura 2.14 se puede ver un ejemplo de este tipo de diagramas.

Figura 2.14. Diagrama de secuencia de un proceso de trabajo

CAPÍTULO 3: DESCRIPCIÓN DE LA APLICACIÓN

3.1. INTRODUCCIÓN

Este capítulo se centra en la descripción de la aplicación a nivel de usuario. El objetivo principal es explicar su funcionamiento sin precisar detalles funcionales de diseño o desarrollo técnico. En cierta manera, debe conformar una especie de pormenorizado manual de usuario donde se especifican todos los aspectos del juego.

Se comenzará por hacer un recorrido completo por el sistema de pantallas que conforman la aplicación, que permita dar a conocer cada uno de los menús, opciones, estados e imágenes que el usuario podrá encontrarse durante la ejecución del programa.

Como ya se mencionó en el capítulo de introducción, este videojuego toma como base el videojuego desarrollado por Taito en 1986, “Arkanoid”. Aquí se desarrolla una versión para dispositivos móviles, para dos jugadores y con dos modos de juego. Después de haber presentado las pantallas y menús de la aplicación, se dedicará una sección para precisar con mayor detalle los aspectos que caracterizan y distinguen cada modo de juego, así como una descripción minuciosa sobre el funcionamiento del mismo, indicando la manera de proceder para jugar.

Posteriormente, en las secciones sucesivas del presente capítulo, se especificarán y analizarán los diferentes elementos que integran el videojuego en sí: se comentarán los diferentes tipos de ladrillo que podrán aparecer durante una partida, los tipos de *power-ups* implementados, se presentarán los niveles que componen el videojuego y se justificará su elección y el orden en que se deberán completar y, por último, se tratará el apartado sonoro de la aplicación, indicando los diferentes efectos sonoros y melodías que lo componen, así como el momento en que son ejecutados.

3.2. PANTALLAS Y MENÚS

En cualquier videojuego, antes de iniciar una partida conviene siempre configurar algunos parámetros, tales como la resolución de la pantalla, la configuración

de teclado, tarjeta gráfica, idioma... Dependiendo de la complejidad y del tipo de videojuego que se trate, estos parámetros serán más o menos numerosos y sofisticados. En el videojuego desarrollado, dado que se trata de la adaptación de un juego de los años ochenta al entorno de los dispositivos móviles, tan sólo hay que preocuparse por configurar unos pocos que serán comentados a lo largo del presente capítulo. En todo caso, será necesario pasar por una serie de menús y pantallas preliminares que permitan definirlos antes de establecer el inicio de la partida.

En este apartado se irán mostrando las diferentes pantallas que constituyen la aplicación del videojuego, así como la navegación que se realiza a través de ellas.

La aplicación comienza con una pantalla en la que se pregunta si el jugador desea habilitar el sonido a la misma o no (figura 3.1). Así, siempre dispondrá de la opción de poder empezar la aplicación sin sonido si se encuentra en un entorno en el que es preferible mantener silencio. En cualquier caso, posteriormente se podrá modificar desde el menú correspondiente la decisión tomada en este punto.

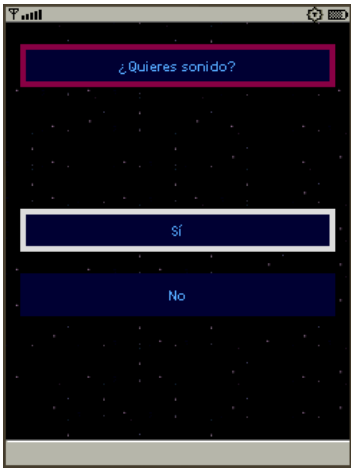


Figura 3.1. Pantalla de habilitación de sonido

Con las flechas arriba y abajo del terminal móvil se podrá desplazar sobre las dos opciones disponibles en este menú –y en general, en el resto de menús– y con el botón central o de disparo, aceptar la opción que finalmente se haya escogido.

Tras la pantalla de configuración de sonido, se muestra por pantalla un mensaje de advertencia de que es necesario habilitar la comunicación por Bluetooth, activando la

visibilidad del dispositivo, para poder jugar al videojuego (figura 3.2). En caso de que no estuviese habilitado, no sería posible la creación del servicio de conexión a la partida creada, ni tampoco la conexión de un dispositivo remoto a esa partida, lógicamente.

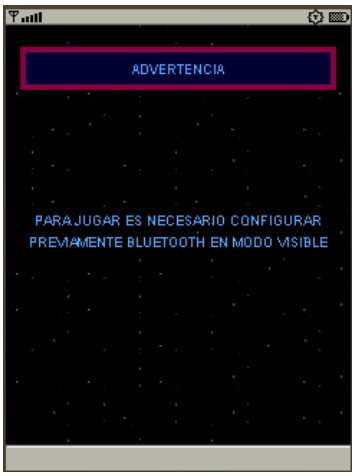


Figura 3.2. Mensaje de advertencia de activación de la visibilidad Bluetooth

Después de esto, la aplicación muestra la pantalla de presentación del videojuego (figura 3.3), con la melodía del mismo si se ha habilitado el sonido, o en silencio, en caso contrario. En la parte inferior de la pantalla se advierte el mensaje de *Pulse cualquier tecla para continuar*.

Tras pulsar cualquier tecla, se avanza hasta la pantalla del menú principal del juego. En ella estarán las siguientes opciones de selección:

- Iniciar Partida
- Unirse a una Partida
- Opciones
- Salir

Estas opciones se muestran en la figura 3.4. De nuevo, el menú puede recorrerse mediante el uso de las flechas arriba y abajo, escogiendo la opción deseada con el botón de disparo.

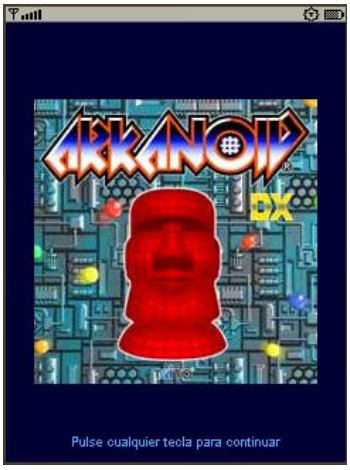


Figura 3.3. Pantalla de presentación de la aplicación

Figura 3.4. Menú principal de la aplicación

A continuación, se describirá brevemente la función de cada una de estas opciones:

- *Iniciar Partida* crea una nueva partida a la que otro jugador podrá unirse desde su terminal.

- *Unirse a una Partida*, como es de esperar, permite al jugador unirse a la partida previamente creada desde otro terminal.

- *Opciones* permite acceder al control de otros parámetros de la aplicación, en concreto el idioma y el sonido, además de la consulta de las máximas puntuaciones y la ayuda de la aplicación.

- Por último, está también disponible desde este menú la posibilidad de salir de la aplicación, sin más que elegir la opción *Salir*.

3.2.1. INICIAR PARTIDA

Al iniciar una nueva partida, aparecerá en pantalla un menú para elegir el modo de juego: cooperativo o *deathmatch* (figura 3.5). En caso de que se desee dar marcha atrás, se puede volver al menú anterior a través de la opción *Volver*.

Figura 3.5. Pantalla de elección de modo de juego

Tras elegir el modo de juego deseado, el terminal registra un servicio de manera que otro terminal pueda hacer una petición de cliente a dicho servicio. Así, al iniciar la nueva partida, el dispositivo se quedará en espera de que un terminal remoto solicite el servicio que está ofreciendo. En la figura 3.6 se puede observar la pantalla que aparece una vez se ha confirmado la creación del servicio.

Llegados a este punto, hay dos opciones posibles: esperar a que un terminal se conecte al servicio que se ha creado, o volver al menú principal. Si un dispositivo remoto localiza el terminal y solicita el servicio ofrecido, la partida comienza automáticamente en ambos terminales. Si, por el contrario, se desea volver al menú principal, habría que seleccionar la opción *Volver* que aparece en pantalla.

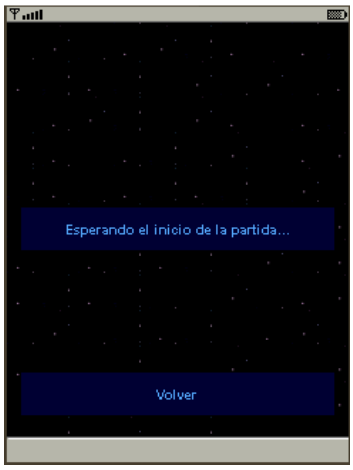


Figura 3.6. Creando una nueva partida

3.2.2. UNIRSE A UNA PARTIDA

Como ya se ha comentado, en lugar de crear una nueva partida, es posible unirse a una partida ya existente. Para ello, hay que escoger la opción *Unirse a una Partida*. En este caso, se tendrá la responsabilidad de localizar el dispositivo que ofrezca el servicio necesario para poder unirse a la partida que ya se ha creado. Por tanto, una vez se ha tomado la decisión de unirse a una partida, es preciso un proceso de búsqueda de dispositivos (figura 3.7).

De todo esto se deduce que, para que pueda haber partida, es necesario que un terminal la cree y el otro se una a ella. No es posible pretender iniciar una nueva partida si ambos terminales la crean o si ambos quieren unirse a una que nadie ha creado.

Figura 3.7. Pantalla de búsqueda de dispositivos remotos

Seleccionando la opción *Descubrir Dispositivos*, se inicia, pues, la búsqueda de dispositivos remotos (figura 3.8). Nuevamente, con la opción *Volver* se regresa a la pantalla del menú inicial.

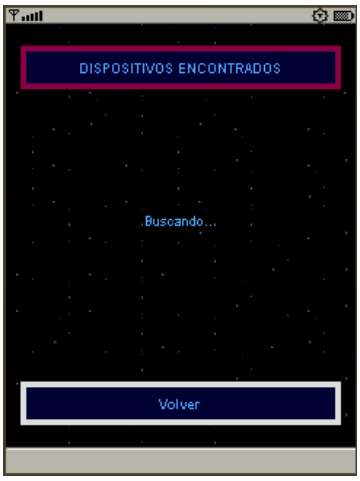


Figura 3.8. Búsqueda de dispositivos remotos

Tras la búsqueda, se presentará por pantalla la lista de dispositivos encontrados, como se muestra en la figura 3.9. El nombre del dispositivo encontrado que aparece en la imagen, *Wireless Toolkit*, responde al *friendly name* que por defecto asigna el emulador a cada instancia que se ejecuta de él. En un caso real, aparecerían los *friendly names* con los que los usuarios de los dispositivos encontrados quieren que se les localice (por ejemplo, *Alberto*, *Nokia N95*, *K750i*,…).

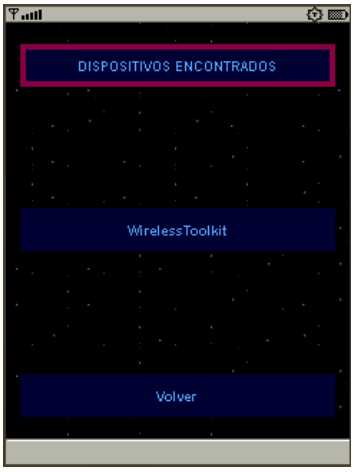


Figura 3.9. Pantalla con los dispositivos encontrados

En caso de que no encuentre ningún dispositivo, se advertirá por pantalla el correspondiente mensaje (figura 3.10). Pulsando cualquier tecla se volverá a la pantalla de búsqueda de dispositivos remotos.

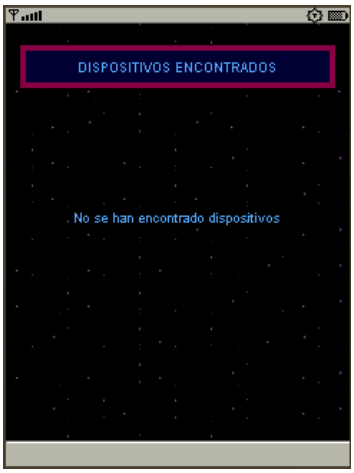


Figura 3.10. Búsqueda sin resultados

Igualmente, si aun habiendo detectado dispositivos, no se selecciona ninguno, aparecerá por pantalla el conveniente mensaje de aviso (figura 3.11). En este caso, pulsando cualquier tecla se volvería a la pantalla con la lista de dispositivos encontrados.

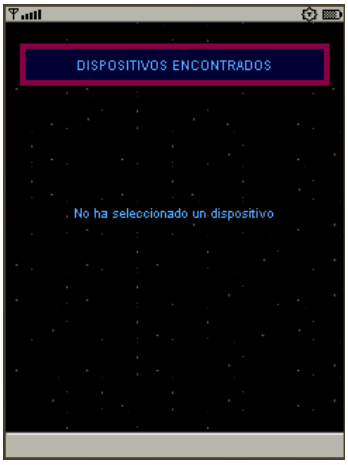


Figura 3.11. No selección de un dispositivo

Si por el contrario el dispositivo remoto que se ha seleccionado para conectarse no soporta el servicio que se necesita para establecer la comunicación *Bluetooth* necesaria entre ambos terminales para iniciar la partida multijugador, aparecerá por

pantalla el mensaje que se muestra en la figura 3.12. Nuevamente, al pulsar una tecla se volvería al menú con la lista de dispositivos remotos encontrados.

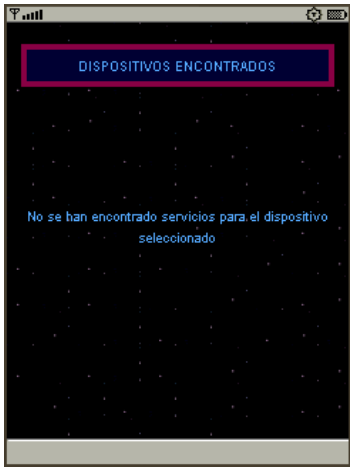


Figura 3.12. Servicios no encontrados en el dispositivo remoto

3.2.3. OPCIONES

Puede que, antes de comenzar a jugar, se desee configurar el idioma, habilitar el sonido si no se había hecho al iniciar la aplicación, o deshabilitarlo si ya estaba activado y se prefiere deshabilitarlo, consultar las máximas puntuaciones o echar un vistazo rápido al menú de ayuda para obtener más información sobre la aplicación. En tal caso, como ya se ha comentado antes, habría que seleccionar *Opciones*. El aspecto que presenta el menú que aparece tras seleccionar esta opción se muestra en la figura 3.13.

Figura 3.13. Pantalla del menú Opciones

Al seleccionar *Ayuda*, aparecerá en pantalla un texto que explica los objetivos del juego y la manera de proceder para conseguirlos (figura 3.14).

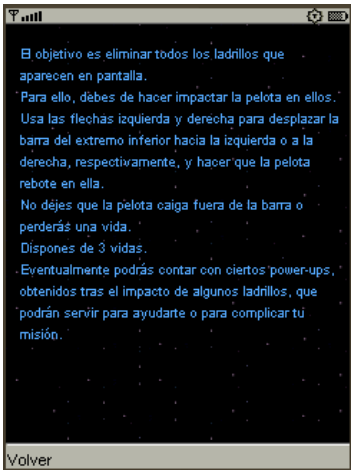


Figura 3.14. Ayuda de la aplicación

Seleccionando el botón de *Volver*, se regresa al menú *Opciones*.

Cabe añadir también que, si las dimensiones de la pantalla no son lo suficientemente grandes, el menú *Ayuda* dispone de *scroll*, es decir, se puede seguir leyendo el texto sin más que desplazándose con las flechas arriba y abajo, según convenga. Cuando quede texto que leer hacia abajo, aparecerá una pequeña flecha hacia abajo en la parte inferior de la pantalla indicando que es posible seguir leyendo, e igualmente se podrá ver otra flecha hacia arriba en la parte superior cuando quede texto sin mostrar por la parte de arriba.

Al pulsar la opción *Activar Sonido*, se habilitará el sonido de la aplicación y en el menú de opciones se cambiará esa opción por la de *Desactivar Sonido* (figura 3.15). Igualmente, al pulsar *Desactivar Sonido*, se cancelará el sonido del programa y la opción pasará a ser *Activar Sonido*.

Si se desea cambiar el idioma de la aplicación, habría que seleccionar la opción *Idioma*. En el nuevo menú que se presenta por pantalla (figura 3.16), aparecen tres idiomas disponibles: inglés, español e italiano.

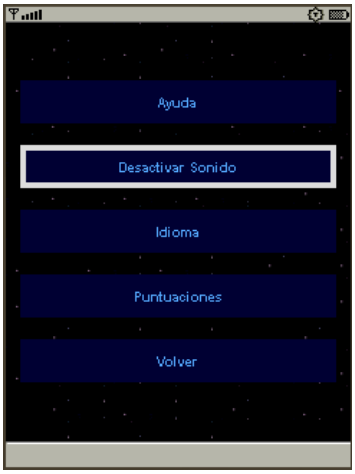


Figura 3.15. Activar/Desactivar sonido

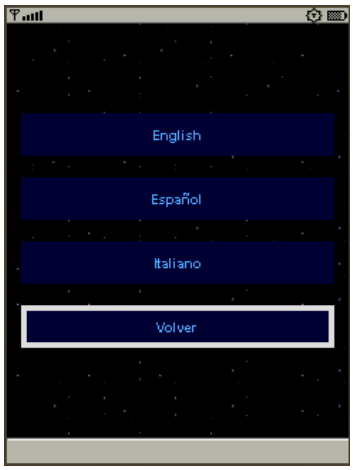


Figura 3.16. Pantalla del menú Idioma

Tras seleccionar el idioma deseado, se puede retornar al menú *Opciones* seleccionando la opción *Volver*, ahora ya escrita en el idioma finalmente elegido (*Back* en inglés, *Dietro* en italiano). También es posible volver al menú *Opciones* sin haber seleccionado ningún idioma, en cuyo caso se mantendría la configuración de idioma que había antes de entrar a dicho menú.

Se pueden ver las mejores puntuaciones obtenidas en el videojuego, accediendo al menú que aparece tras seleccionar la opción *Puntuaciones*. El aspecto que presenta dicho menú se muestra en la figura 3.17.

Figura 3.17. Pantalla del menú Puntuaciones

Como se puede observar, hay tres opciones disponibles: *Cooperativo*, para consultar las puntuaciones más altas en el modo cooperativo; *Deathmatch*, para las del modo *deathmatch*; y *Volver* para regresar al menú *Opciones*.

En el modo cooperativo, la puntuación se registra a nombre de los dos jugadores, puesto que ambos han *cooperado* conjuntamente para lograr el mismo objetivo. En el modo *deathmath*, sin embargo, la máxima puntuación queda registrada sólo a nombre del que ha conseguido mantenerse en la partida, y por tanto se considera el vencedor de ella. En la figura 3.18 se muestran las puntuaciones por defecto de cada modo de juego.

(a) **(b)**

Figura 3.18. Puntuaciones por defecto para los dos modos de juego
(a) Cooperativo, (b) Deathmatch

Finalmente, la última opción del menú *Opciones* es la opción *Volver*, con la que se volvería al menú principal.

3.2.4. SALIR

Por último, si desde el menú principal se pulsa la opción *Salir*, aparecerá la pantalla de confirmación de que se desea abandonar la aplicación (figura 3.19).

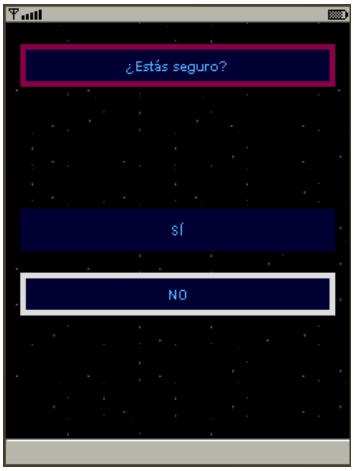


Figura 3.19. Saliendo de la aplicación

Pulsando la opción *No*, se vuelve al menú principal de la aplicación. Por el contrario, al pulsar *Sí*, se abandona definitivamente el programa. Como suele ser habitual en la mayoría de las aplicaciones informáticas, el cursor se sitúa inicialmente sobre el *No*, por si se ha seleccionado *Salir* de forma accidental.

3.2.5. PANTALLAS DE LA PARTIDA

Una vez se ha establecido el inicio de la partida, lo que aparece en pantalla es directamente el primer nivel (figura 3.20).

El juego durará mientras haya algún jugador que disponga de vidas y ninguno de ellos abandone la partida (pulsando el botón *Salir*). En el caso en que la partida acabe de forma natural, es decir, agotando la vida de al menos uno de los dos jugadores, se mostrará por pantalla la puntuación obtenida (figura 3.21).

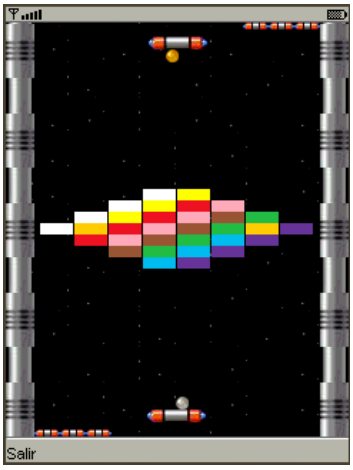


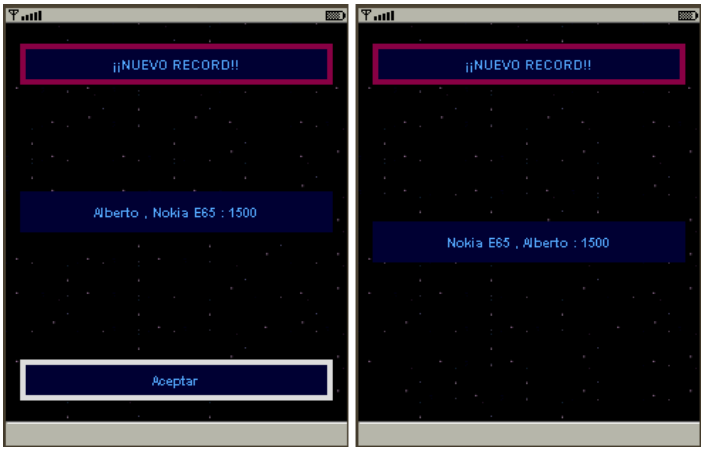
Figura 3.20. Iniciando una partida

(a) **(b)**

Figura 3.21. Puntuación final obtenida, mostrada en ambos terminales
(a) terminal que inició la partida, (b) terminal que se unió

Si dicha puntuación supera alguna de las registradas como máximas puntuaciones hasta el momento en ese modo de juego, lo que se mostrará será una pantalla indicando que se ha logrado alcanzar un nuevo record (figura 3.22).

En cualquier caso, se establezca o no un nuevo record, junto con la puntuación obtenida se especifica el jugador o jugadores (dependiendo del modo de juego, *deathmatch* o cooperativo, respectivamente) que han alcanzado esa puntuación, identificados por los *friendly names* de sus dispositivos.



(a) **(b)**

Figura 3.22. Nuevo record obtenido, mostrado en ambos terminales (a) terminal que inició la partida, (b) terminal que se unió

Tras esto, pulsando *Aceptar* desde el dispositivo del jugador que ha creado la partida, aparecerá la pantalla de *Game Over*, haciendo referencia a que la partida se ha terminado (figura 3.23).

Figura 3.23. Pantalla de fin de partida (game over)

Preguntará si se desea comenzar una nueva partida o si, por el contrario, se da por finalizado el juego. En el primer caso, pulsando la opción *Sí*, se daría comienzo a una nueva partida, por lo que en pantalla aparecerá de nuevo el correspondiente menú (mostrado anteriormente en la figura 3.5) para elegir el modo de juego (cooperativo o *deathmatch*). Tras haber elegido el modo de juego deseado se iniciará una nueva partida en ambos terminales. En el segundo caso, pulsando la opción *No*, se volvería al menú inicial en ambos terminales (figura 3.4).

Por otra parte, si la partida acaba de forma abrupta, por abandono de uno de sus jugadores, aparecerá el menú principal del videojuego en el terminal del que abandona, y en el otro extremo se advertirá que el dispositivo remoto ha abandonado la partida (figura 3.24).

Figura 3.24. Desconexión del dispositivo remoto

Tras pulsar en *Aceptar*, aparecerá a continuación el menú principal de la aplicación.

3.3. INSTRUCCIONES DE JUEGO

En primer lugar, como ya se ha comentado en la sección anterior, el videojuego admite dos tipos de partida: en modo cooperativo y en modo *deathmatch*. En ambos casos, como ya se ha mencionado en anteriores ocasiones, el objetivo no es otro que el de eliminar todos los ladrillos rompibles de cada nivel que se presente por pantalla. En la sección 3.4 se detallarán los tipos de ladrillos existentes en el videojuego.

3.3.1. MODO COOPERATIVO

En este modo de juego, los dos jugadores colaborarán el uno con el otro para lograr un mismo objetivo: llegar lo más lejos posible en la partida. Para ello, procurarán mantener en pantalla tanto la pelota del propio jugador como la del otro, haciéndolas rebotar sobre sus respectivas naves, puesto que ambas contribuirán a eliminar los ladrillos de cada nivel. Si la nave del extremo –superior o inferior– al que se dirige la

pelota no llegase a hacerla rebotar, ésta se saldría de la pantalla y el jugador al que pertenecía dicha pelota perdería una vida. Cuando uno de los dos jugadores se quede sin vidas, la puntuación que se registra es la obtenida conjuntamente por los dos, es decir, la suma de todos los puntos obtenidos por eliminar los ladrillos entre ambos.

3.3.2. MODO DEATHMATCH

Por su parte, en el modo *deathmatch*, el objetivo es el de durar más en la partida que el otro jugador. Por tanto, no interesará mantener en juego también la pelota del otro jugador. Al contrario, se penalizará con una vida menos si ésta rebota sobre la nave propia. A diferencia del modo cooperativo, en caso de que la pelota del jugador alcance el extremo opuesto (el extremo donde está la nave del otro jugador) y la nave no alcance a hacerla rebotar, la pelota rebotaría y permanecería en pantalla. Solo se perderá cuando se dirija hacia el extremo donde está la nave controlada por el mismo jugador y ésta no llegase a hacerla rebotar. Cuando uno de los dos jugadores se quede sin vidas, la puntuación que se registra es la obtenida por el otro jugador, es decir, el que ha conseguido mantenerse con vida en la partida, y por tanto, el que ha ganado.

Una vez se haya seleccionado el modo de juego e iniciado la partida, cada jugador manejará la nave que se encuentra en la parte inferior de la pantalla de su dispositivo, desplazándola a izquierda y derecha con las flechas izquierda y derecha respectivamente. Adicionalmente, podrá usar el botón de disparo para lanzar la pelota. En caso de que no lo hiciera, transcurridos unos segundos la pelota se lanzará automáticamente.

Con la pelota en movimiento, cada jugador tratará de impactar su pelota con los ladrillos rompibles presentes en el nivel que se muestra por pantalla, haciéndola rebotar sobre su nave. Eventualmente, podrá contar con ciertos *power-ups* o potenciadores, que le permitirán agilizar la eliminación de estos ladrillos, o bien podrán entorpecer su cometido. Estos *power-ups* se podrán obtener tras la eliminación de algunos ladrillos rompibles, los cuales desprenderán una cápsula hacia el extremo de la nave a la que pertenece la pelota que ha impactado en dicho ladrillo. El jugador que maneje esa nave deberá colocarla debajo de la cápsula para recogerla si quiere activar el *power-up* asociado a ella. Los tipos de *power-ups* implementados para este videojuego, así como

sus principales características, efectos y las cápsulas que los identifican se describirán con mayor detalle en la sección 3.5.

Una vez se haya superado un nivel, la partida avanzará hacia el siguiente nivel hasta un total de diez. Si tras completar el último nivel la partida sigue activa, se vuelve al primero, y así sucesivamente hasta que termine la partida o hasta que uno de los dos jugadores abandone. Estos niveles se comentarán más en detalle en la sección 3.6.

3.4. LADRILLOS

Cada nivel consta de un conjunto de ladrillos que hay que eliminar para pasar al siguiente nivel. Dichos ladrillos pueden ser de tres tipos: rompibles tras un impacto, rompibles tras dos impactos e irrompibles. Naturalmente, los ladrillos irrompibles no tendrán que ser destruidos para pasar a la siguiente pantalla.

3.4.1. LADRILLOS ROMPIBLES TRAS UN IMPACTO

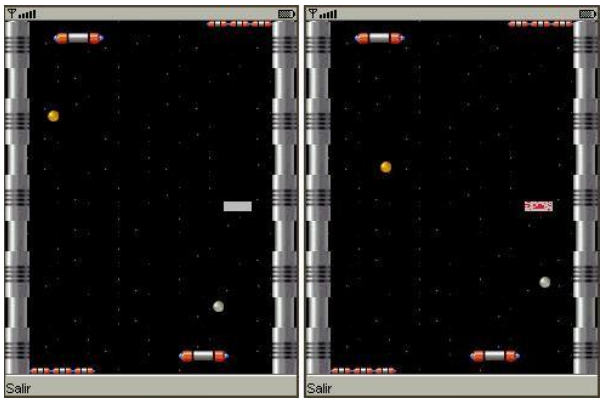
Pueden ser de diferentes colores: amarillo, azul, blanco, marrón, morado, rojo, rosa o verde. Son eliminados después de ser golpeados con una pelota o con un proyectil que haya disparado una nave con el *power-up* de disparo activado. En la figura 3.25 puede verse un ejemplo de este tipo de ladrillos, concretamente el ladrillo verde.

Figura 3.25. Ladrillo rompible tras 1 impacto: ladrillo verde

Por cada ladrillo eliminado de este tipo, el jugador obtiene 100 puntos.

3.4.2. LADRILLOS ROMPIBLES TRAS DOS IMPACTOS

Se identifican con el color gris. Al igual que los ladrillos rompibles tras un impacto, éstos pueden ser golpeados por una pelota o por un proyectil. Sin embargo, son necesarias un total de dos colisiones para eliminarlos de la pantalla. Tras el primer impacto, el ladrillo cambia de aspecto para indicar que ya ha sido golpeado y que solo hace falta otro impacto más para eliminarlo definitivamente. En la figura 3.26 se puede apreciar este tipo de ladrillo en sus dos posibles estados: sin haber recibido ningún impacto y después de recibir el primero de los dos impactos necesarios para su eliminación.



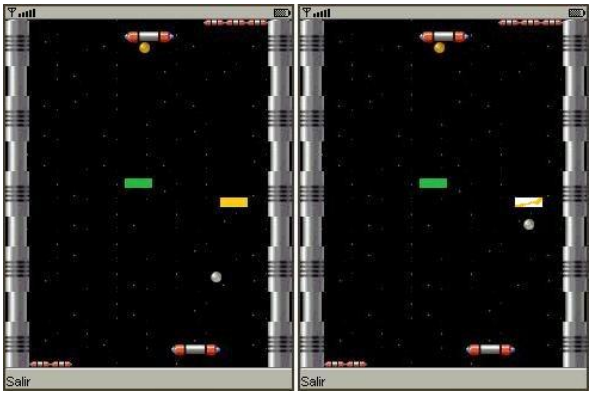
(b) **(b)**

Figura 3.26. Ladrillo rompible tras 2 impactos
(a) aspecto inicial (b) aspecto tras el primer impacto

Al necesitar dos impactos para eliminarlo, la puntuación que obtiene el jugador por destruir este tipo de ladrillos se duplica, y son, por tanto, 200 puntos.

3.4.3. LADRILLOS IRROMPIBLES

Como su propio nombre indica, estos ladrillos no se pueden destruir, independientemente del número de impactos que reciba durante la partida. Se distinguen por ser de color dorado y por brillar durante un instante tras ser alcanzados por una pelota o por un proyectil. En la figura 3.27 se observa este tipo de ladrillo en su estado normal, es decir, cuando no recibe impacto, y en el instante inmediatamente posterior a un impacto, momento en el que emite un breve destello luminoso.



(a) **(b)**

Figura 3.27. Ladrillo irrompible

(a) aspecto en reposo (b) aspecto tras ser impactado

3.5. POWER-UPS

En el capítulo de introducción ya se comentó que en el "Arkanoid" original existían los denominados *power-ups* o potenciadores, cuyo efecto podría ayudar al jugador o perjudicarle, según qué tipo de *power-up*. En esta adaptación multijugador para dispositivos móviles se han implementado una serie de potenciadores, cada uno asociado a una determinada cápsula, que se obtiene, al igual que en el videojuego original, tras eliminar el ladrillo que la contiene. En el presente apartado se detallarán las características, los efectos y las cápsulas que identifican a los *power-ups* implementados para este videojuego.

3.5.1. ATRAPAR LA PELOTA

Consiste en que la pelota no rebota al alcanzar la nave, sino que se queda adherida a ésta. El jugador cuya nave haya recogido la cápsula asociada a este *power-up* podrá atrapar la pelota con su nave, desplazarse y ejecutar el lanzamiento de la misma pulsando el botón de disparo o esperando a que se lance de forma automática transcurridos unos segundos. En la figura 3.28 se muestra la cápsula que identifica a este *power-up* y una imagen de la pelota adherida a la nave.

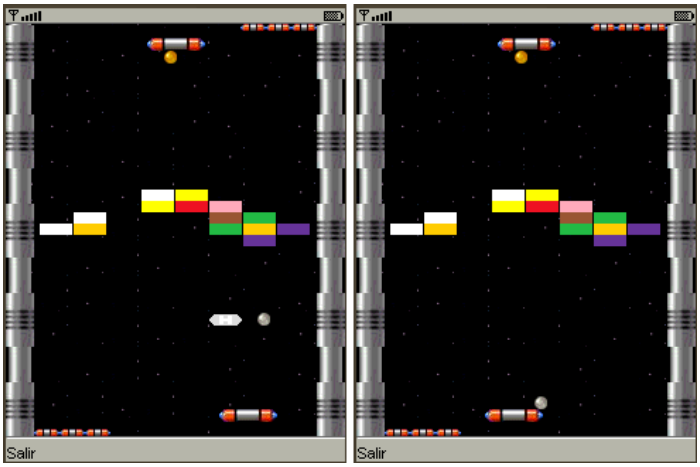


Figura 3.28. Power-up que atrapa la pelota

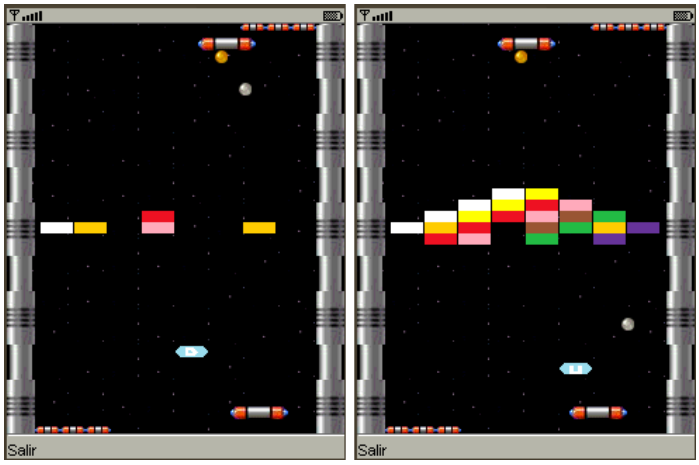
Se puede observar que la cápsula es de un color gris claro, con una *H* (del inglés, *hold*) inscrita en ella.

Este *power-up* puede ser útil en aquellos casos en los que haya ladrillos de difícil acceso, cuyo impacto sobre ellos simplemente a base de hacer rebotar la pelota sobre la nave resulte complicado. Si al jugador le da tiempo suficiente de preparar la nave para que la pelota se pose sobre una posición favorable, este *power-up* puede ser un poderoso aliado.

3.5.2. DISMINUIR / AUMENTAR LA VELOCIDAD DE LA PELOTA

La velocidad a la que se mueve la pelota que controla el jugador puede dificultarle la tarea de mantenerla en pantalla si es muy elevada o ayudarle a no perderla si es moderada. En el juego hay dos *power-ups* relacionados con este parámetro, uno que la reduce y otro que la aumenta. En la figura 3.29 pueden observarse las cápsulas que distinguen a estos dos *power-ups*.

Ambas comparten el mismo color azul celeste de fondo y se distinguen por la letra que lleva inscrita: una *D* (del inglés, *down*) para el *power-up* que disminuye la velocidad de la pelota y una *U* (*up*) para el que la aumenta.

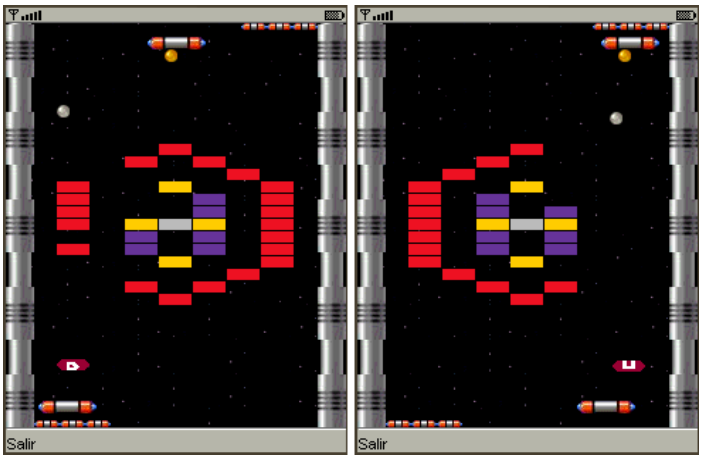


(b) **(b)**

Figura 3.29. Power-ups que disminuyen / aumentan la velocidad de la pelota
(a) disminuye la velocidad (b) aumenta la velocidad

3.5.3. DISMINUIR / AUMENTAR LA VELOCIDAD DE LA NAVE

Al igual que para la velocidad de la pelota, la velocidad con la que se mueve la nave al desplazarla a izquierda y derecha puede ser alterada mediante *power-ups* (figura 3.30). En concreto hay dos, uno para aumentar esa velocidad y otro para disminuirla. Cuanto mayor sea esta velocidad, con mayor antelación se podrá ubicar la nave en una posición que permita hacer rebotar la pelota sobre ella, y por tanto, más difícil será que el jugador pierda una vida al perder la pelota por el extremo inferior. Asimismo, si esta velocidad se reduce, más difícil será mantener en juego una pelota, por lo que será más probable que el jugador pierda una vida.



(a) **(b)**

Figura 3.30. Power-ups que disminuyen / aumentan la velocidad de la nave
(a) disminuye la velocidad (b) aumenta la velocidad

En la figura 3.30 se pueden ver las cápsulas de color rojo oscuro que identifican a estos *power-ups*. De nuevo, se distinguen con las iniciales *D* y *U*.

3.5.4. TRIPLICAR EL NÚMERO DE PELOTAS

Cuando el jugador solo dispone de una pelota, solo tiene que preocuparse de mantener en pantalla a esa pelota (y eventualmente la pelota del otro jugador, si están jugando en modo cooperativo), pero también será solo una pelota la que impactará contra los ladrillos del nivel en el que se encuentren. Con más pelotas, la tarea de mantenerlas todas en pantalla se complica, evidentemente, pero a cambio las posibilidades de impacto sobre los ladrillos se multiplican. En este caso, el jugador no perderá una vida mientras mantenga en pantalla al menos una pelota. Existe un *power-up* cuyo efecto consiste justamente en otorgar tres pelotas al jugador que recoge su cápsula. En la figura 3.31 se observa la cápsula, de color verde y con una *T* (del inglés, *three*) inscrita, que identifica a este *power-up*, así como las tres pelotas que aparecen en pantalla una vez la nave ha recogido dicha cápsula.

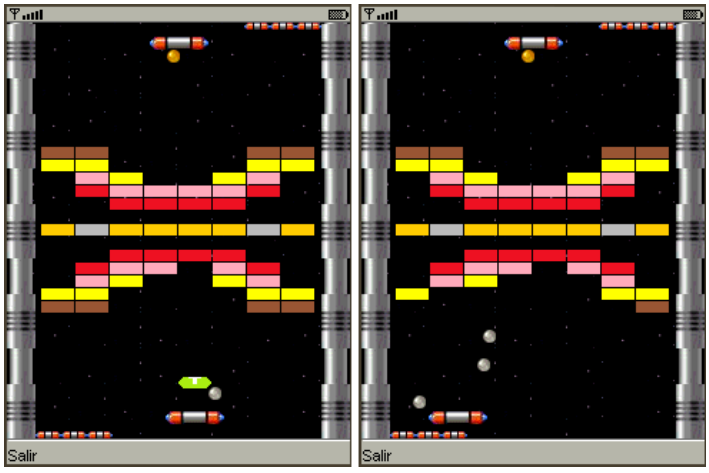


Figura 3.31. Power-up que otorga 3 pelotas al jugador

3.5.5. DISMINUIR / AUMENTAR EL ANCHO DE LA NAVE

En todos los niveles se empieza con un ancho de la nave predeterminado. Sin embargo, este parámetro también puede ser modificado con los *power-ups*. En particular, la nave podrá ser ensanchada o estrechada. Con una nave más amplia, será más fácil evitar que la pelota escape de la pantalla, y por tanto, será más difícil que el jugador pierda una vida. En la figura 3.32 se muestra la cápsula que identifica al *power-*

up que ensancha la nave y el efecto de aumento de su tamaño tras haber recogido la cápsula.

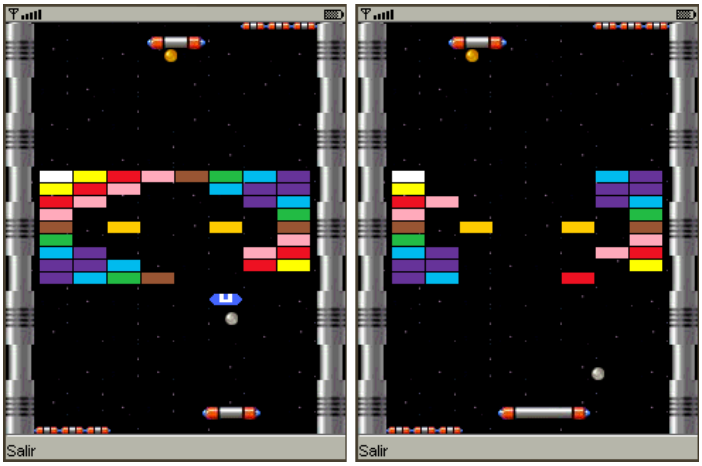


Figura 3.32. Power-up que ensancha la nave

Igualmente, si la nave fuese más estrecha sería más complicado provocar que la pelota rebotase sobre ella, y sería pues, más difícil mantenerla en pantalla. En la figura 3.33 se expone la cápsula asociada a este *power-up* y su consiguiente efecto de reducción del tamaño de la nave.

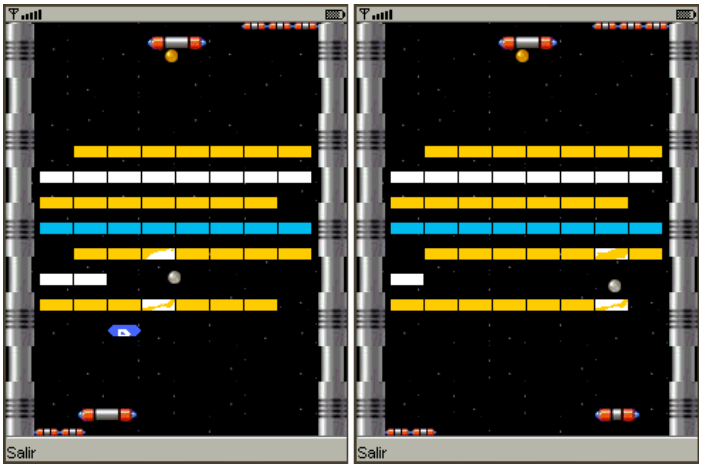


Figura 3.33. Power-up que estrecha la nave

Al igual que en casos anteriores, ambos potenciadores comparten el mismo color para la cápsula, en este caso el azul, y se distinguen por la letra que las acompaña: *U* para el que ensancha la nave y *D* para el que la estrecha.

3.5.6. NO REBOTE

Cuando una pelota impacta contra un ladrillo, independientemente de si lo elimina o no, rebota. Sin embargo, existe un *power-up* cuyo efecto consiste precisamente en alterar esta condición y cancelar el rebote, permitiendo así que la pelota continúe su avance. En la figura 3.34 puede observarse la cápsula que se asocia a este potenciador y el efecto que produce cuando se adquiere.

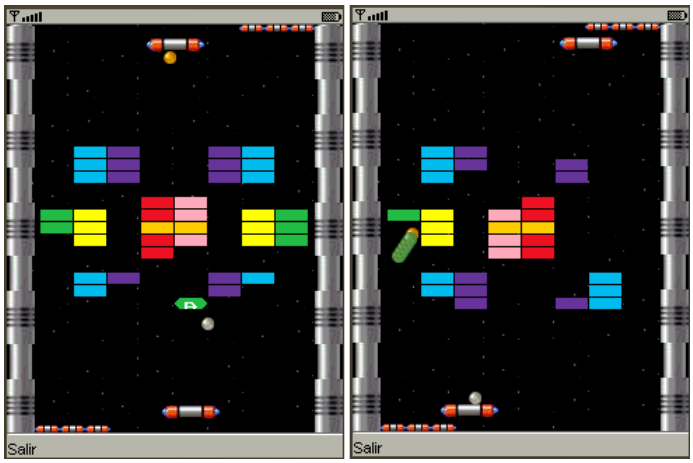


Figura 3.34. Power-up que deshabilita el rebote con los ladrillos

Se puede apreciar que la cápsula que lo identifica es de color verde y lleva la letra *B* (del inglés, *bounce*, rebote) inscrita. Al recoger dicha cápsula, la pelota empieza a dejar una estela de color verde que indica que dicho *power-up* está activo y por tanto, no rebotará con ningún ladrillo rompible, tan solo contra los ladrillos irrompibles.

Este tipo de *power-up* resulta especialmente útil cuando el nivel contiene amplios bloques compuestos de ladrillos rompibles, ya que pueden ser eliminados rápidamente con unos pocos rebotes de la pelota sobre la nave.

3.5.7. DISPARO

Uno de los *power-ups* más emblemáticos del "Arkanoid" era aquel que permitía al jugador disparar proyectiles como complemento para destruir los ladrillos de un nivel. En el videojuego desarrollado se ha implementado esta característica, a semejanza del juego original. Así, aquí también se dispondrá de la misma capacidad de disparar proyectiles al recoger la correspondiente cápsula sin más que pulsar la tecla de disparo.

En la figura 3.35 se puede apreciar el aspecto que presenta tanto la cápsula como los proyectiles que dispara la nave una vez ha recogido la cápsula.

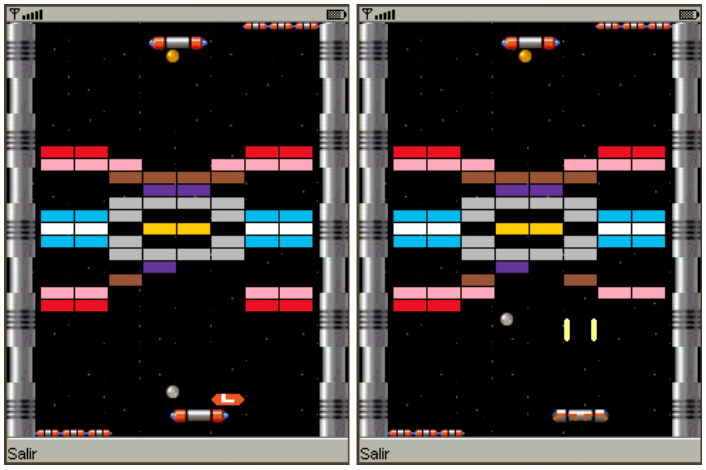


Figura 3.35. Power-up que permite disparar proyectiles

Se observa que la cápsula que distingue a este *power-up* es de color rojo con una *L* (de *láser*) inscrita en ella. Además, los proyectiles que se disparan son estrechos y alargados y de color amarillento, como en el videojuego original. Es de notar también que el aspecto de la nave cambia cuando se activa este *power-up*, pasando a ser como se muestra en la imagen de la derecha de la figura.

3.5.8. VIDA EXTRA

Cuando la pelota de un jugador se sale de la pantalla, éste pierde una vida. Cuando pierde todas las vidas, ha perdido la partida. Sin embargo, el juego proporciona también una forma de ganarse una vida, y es recogiendo la cápsula asociada al *power-up* que proporciona una vida extra.

En la figura 3.36 se muestra la mencionada cápsula y el efecto de incremento de vidas sobre el jugador que la ha recogido (el que controla la nave de la parte inferior). Puede verse que la cápsula es de color blanco con una *X* (del inglés, *eXtra life*) inscrita en ella y que el jugador que maneja la nave de la parte inferior de la pantalla pasa de tener tres vidas a tener cuatro.

Recoger este *power-up* es siempre una ayuda, lógicamente, y ayuda a prolongar la duración en la partida del jugador que recoge su cápsula.

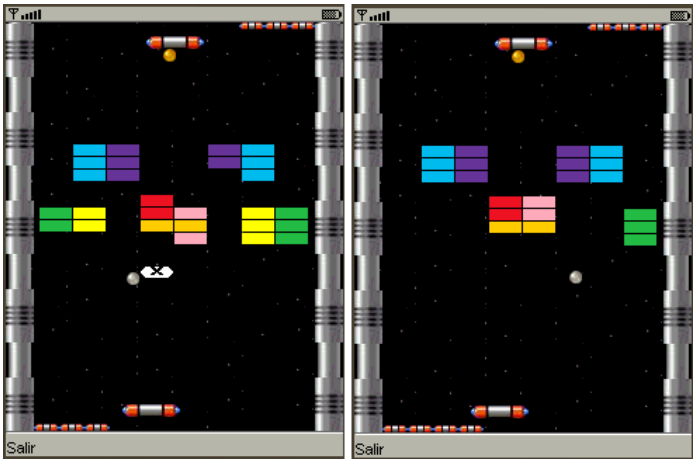


Figura 3.36. Power-up que proporciona una vida extra

3.6. NIVELES

El videojuego se compone de un total de 10 niveles, cuya dificultad se incrementa a medida que se avanza a través de ellos. Dicha dificultad no depende únicamente del número de ladrillos que se encuentren en cada nivel, sino también de la propia distribución de éstos en la pantalla. En la figura 3.37 se presenta un diagrama que representa el número de ladrillos que hay por cada nivel.

Figura 3.37. Distribución del número de ladrillos por nivel

Puede observarse que, pese a que la tónica general es la del incremento del número de ladrillos conforme se progresa en la partida, no necesariamente el número de ladrillos de un nivel es estrictamente mayor que el de su predecesor. En la figura 3.38 se muestra el aspecto que presentan estos niveles, ordenados de izquierda a derecha.

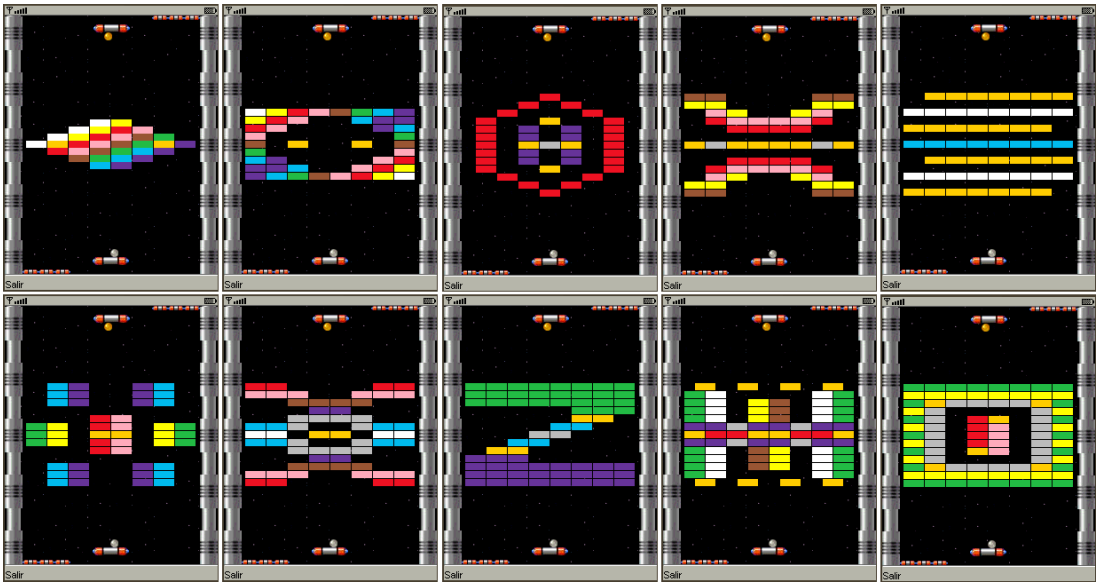


Figura 3.38. Niveles del videojuego

3.7. SONIDO

Como ya se explicó en la sección 3.2, la aplicación dispone de efectos de sonido y melodías, que pueden ser habilitados mediante la opción correspondiente. Si dicha opción está activa al aparecer la pantalla de presentación del videojuego (la que se muestra en la figura 3.3) sonará la melodía principal del mismo, y seguirá sonando hasta que el jugador pulse una tecla. Cuando pulse una tecla, la melodía se interrumpirá de forma abrupta.

Posteriormente, al inicio de la partida, tanto en el terminal que la crea como en el que se une a ella, sonará en ambos la melodía de inicio de partida. Concretamente, en el terminal que la crea, lo hará cuando haya elegido el modo de juego (cooperativo o *deatmatch*) y pase a esperar la conexión del terminal remoto a la partida recién creada (figura 3.6), mientras que en el terminal que se une a ella sonará durante la búsqueda del dispositivo que la ha creado (figura 3.8).

Una vez se inicie la partida, se oirá la melodía de inicio de vida, y se repetirá cada vez que el jugador inicie una nueva vida en el juego, tras haber perdido su pelota.

Asimismo, cada vez que tenga lugar una colisión, sonará el correspondiente efecto de sonido. Dependiendo del tipo de colisión, que podrá ser de una pelota con un ladrillo rompible, un ladrillo irrompible o una nave, o una colisión entre dos pelotas, sonará uno u otro efecto.

Por otra parte, también hay efectos de sonido asociados a determinados *power-ups*. En particular, los asociados a la captura de la cápsula que contiene el *power-up* de ensanchamiento o de estrechamiento de la nave, el que advierte de que se dispone de una vida más tras recoger la cápsula del *power-up* de vida extra, el sonido de la pelota al colisionar con la nave cuando está activo el *power-up* que atrapa la pelota y el sonido que se emite al disparar proyectiles desde la nave cuando se encuentra activo el *power-up* de disparo.

A estos sonidos hay que añadir el efecto correspondiente a cada pérdida de una vida y la melodía de final de la partida, que suena una vez que la partida ha finalizado por el agotamiento de las vidas de al menos uno de los dos jugadores. La melodía suena, pues, cuando se presenta por pantalla la puntuación final obtenida en la partida (figura 3.21).

CAPÍTULO 4: DISEÑO DE LA APLICACIÓN

4.1. INTRODUCCIÓN

En este capítulo se explicará con detenimiento cómo se ha implementado la aplicación para obtener la funcionalidad descrita en el capítulo anterior. Para ello, se indicará de qué manera se han organizado las diferentes clases que componen el programa y cómo se han implementado para contribuir a la funcionalidad global del mismo. Se comenzará por introducir, mediante una pequeña descripción, el marco software en el que se sitúa la aplicación desarrollada.

4.1.1. MARCO SOFTWARE DE LA APLICACIÓN

La aplicación J2ME de videojuego se tendrá que ejecutar en el entorno proporcionado por el dispositivo móvil. Como en la mayoría de las aplicaciones software, la interacción con el usuario se realizará a través del teclado, la pantalla y el altavoz del terminal. El dispositivo responderá a las acciones realizadas por el jugador mediante el teclado, mostrando los resultados de dicha interacción y del desarrollo del juego a través de la pantalla y el altavoz (si el sonido estuviera habilitado). La figura 4.1 muestra el escenario de interacción del jugador con el dispositivo.

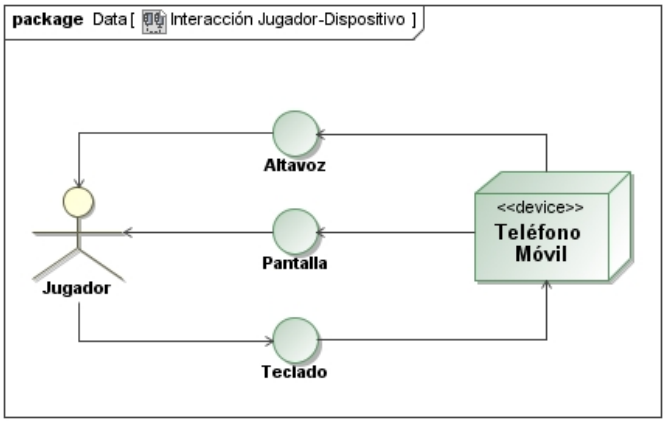


Figura 4.1. Escenario de interacción del jugador con el dispositivo

La arquitectura esquemática del entorno de ejecución software que posee el dispositivo móvil para ejecutar la aplicación e interactuar con el jugador puede apreciarse en la figura 4.2. En ella, se tienen las tres interfaces físicas de comunicación

con el usuario ya señaladas: teclado, pantalla y altavoz; el sistema operativo (SO) y la Máquina Virtual de Java (KVM), con diferentes hilos de aplicaciones que se ejecutan en ésta.

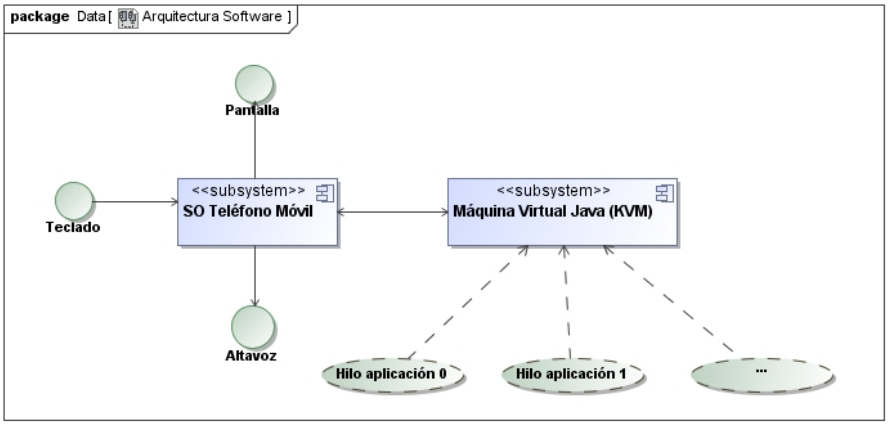


Figura 4.2. Esquema de la arquitectura software de un dispositivo móvil

El SO controla el acceso a las interfaces físicas de comunicación con el usuario. Éste proporciona las rutinas y servicios necesarios a la Máquina Virtual de Java para que las aplicaciones Java puedan hacer uso, a su vez, del teclado, la pantalla y el altavoz. La Máquina Virtual de Java instancia un hilo de ejecución concurrente por cada aplicación J2ME lanzada. De esta forma, se pueden tener varias aplicaciones Java ejecutándose a la vez en el dispositivo. Además, dentro de cada hilo de ejecución de aplicación pueden existir a su vez otros sub-hilos propios de la aplicación que habrán sido contemplados por el programador. La máquina virtual irá alternando su ejecución en función de prioridades y eventos fijados por el programador, pero siempre estarán confinados al entorno propio de su hilo de aplicación.

Como ya se ha mencionado en la introducción al proyecto, la tecnología J2ME sigue un modelo de capas para conseguir un alto grado de portabilidad de las aplicaciones. En la base está el SO del dispositivo móvil; por encima se ubica la máquina virtual de Java (KVM); y por último, antes de llegar a la aplicación, un nivel de API's (*Application Programming Interface*) que proporcionan un conjunto de componentes y utilidades para la programación –estos dos últimos niveles señalados aíslan a la aplicación del SO que pueda estar debajo, con la consiguiente ventaja ya señalada de la portabilidad–. Esta estructura puede verse en la figura 4.3.

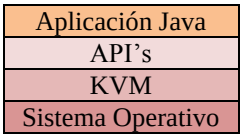


Figura 4.3. Estructura en capas de la tecnología J2ME

Una vez se ha introducido el entorno software sobre el que se implementa el videojuego desarrollado, se abordarán ahora los aspectos particulares del diseño y la implementación de la aplicación desarrollada. En la siguiente sección, se dará una visión de conjunto sobre el programa implementado, indicando la jerarquía de clases y el diagrama de estados de la aplicación. Antes de entrar a analizar en detalle cada una de las clases que dan forma al programa implementado, se justificará en la sección 4.3 la necesidad de ejecutar el mismo en más de un hilo de ejecución de manera simultánea (*multithreading*). Posteriormente, se iniciará el análisis empezando por aquellas clases que procuran una funcionalidad más genérica, como las que se encargan de configurar los apartados gráfico y sonoro de la aplicación o los procesos de lectura y escritura de datos (sección 4.4), para después acercarse progresivamente a la lógica implementada para lograr la funcionalidad del videojuego, contenida en la clase principal, *Juego.java*. Para ello, se analizará en profundidad el mecanismo implementado de establecimiento y liberación de la conexión Bluetooth (sección 4.5), incluyendo el proceso de sincronización entre los dos terminales y los diferentes tipos de mensajes –de sincronización y de datos– que se intercambian entre ellos, así como los diferentes elementos que integran una partida del videojuego (sección 4.6). Finalmente, después de estudiar la lógica del videojuego, en base a los métodos y a los atributos definidos en la clase principal y a la gestión que se hace de los mismos (sección 4.7), se cierra el capítulo presentando el *MIDlet* principal de la aplicación (sección 4.8), desde donde se inicia y se cierra la aplicación desarrollada para el presente proyecto.

4.2. VISION GENERAL DE LA APLICACIÓN

Antes de entrar a analizar en profundidad cada una de las piezas que componen la aplicación desarrollada, conviene establecer un punto de referencia desde el que tener una visión de conjunto sobre la misma. Por un lado, la distribución e identificación de

las clases implementadas, así como las relaciones entre ellas, y por otro, un diagrama que represente de forma esquemática el funcionamiento global del sistema.

4.2.1. CLASES IMPLEMENTADAS

Las clases que integran la aplicación desarrollada en el presente proyecto están estructuradas en paquetes, según se muestra en la figura 4.4.

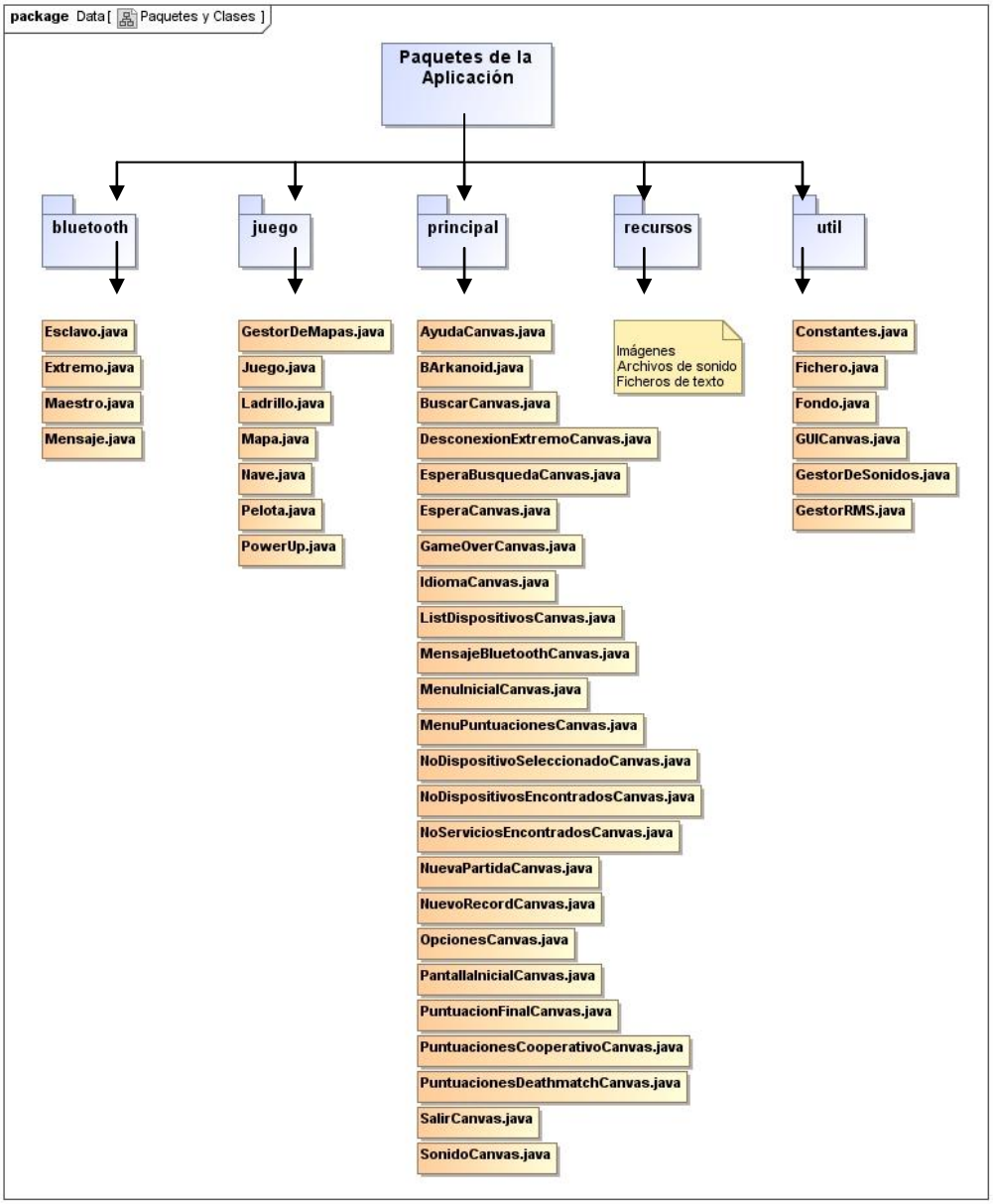


Figura 4.4. Distribución de paquetes y clases implementadas

Por un lado, el paquete *principal* contiene el *MIDlet* de la aplicación (*BArkanoid.java*), junto con las clases que implementan todas las pantallas de menú del

programa. El paquete *juego*, por su parte, aglutina todas las clases encargadas de gestionar los elementos que aparecen durante la partida (ladrillos, naves, pelotas, mapas…), además de la clase donde se ejecuta el bucle principal del juego (*Juego.java*). Por otro lado, en el paquete *bluetooth* se almacenan las clases destinadas a controlar el establecimiento y liberación de la conexión por un enlace Bluetooth entre los dos terminales, así como proporcionar las herramientas necesarias para enviar y recibir mensajes correctamente a través de dicha conexión. En el paquete *util*, hay una serie de clases que resultan ser de utilidad general para todo el código, como una clase que se encarga de gestionar los sonidos de la aplicación, o de proporcionar un formato común a cada pantalla que forme parte de ella, etc. Finalmente, el paquete *recursos* contiene todas las imágenes, sonidos y ficheros de texto que necesita la aplicación. La manera en que estas clases están relacionadas se representa en el diagrama de clases de la figura 4.5.

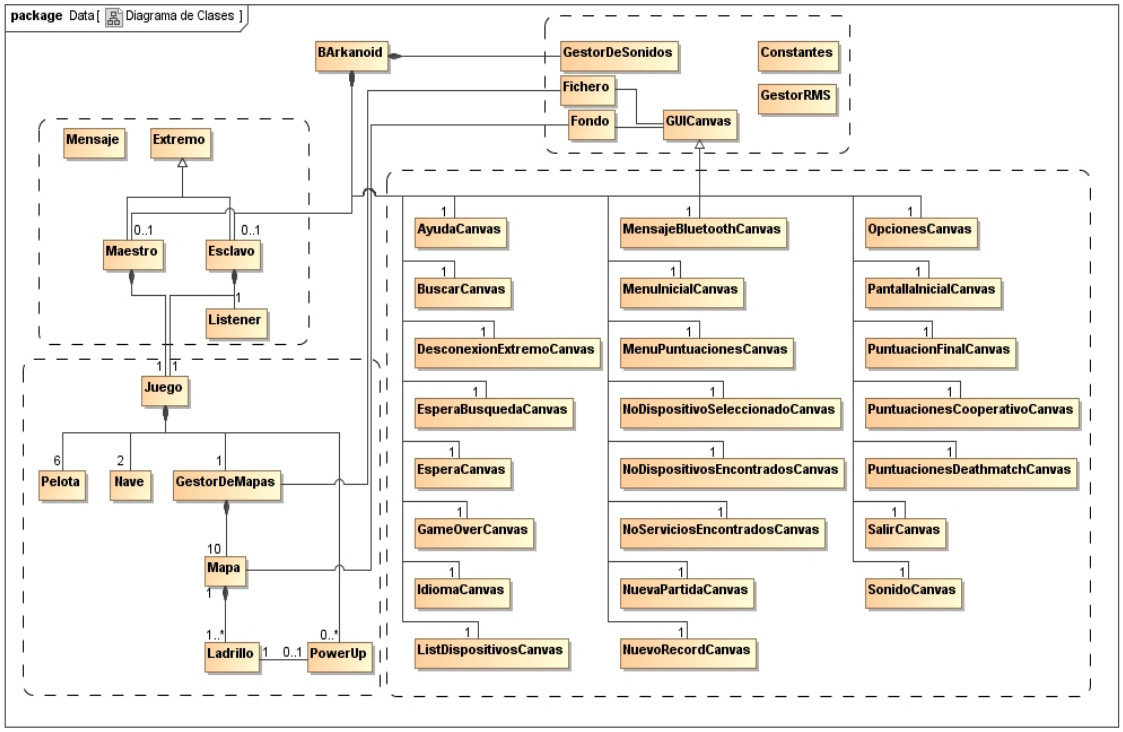


Figura 4.5. Diagrama de clases de la aplicación

Cada una de estas clases se detallará y se analizará detenidamente en las siguientes secciones. Pueden observarse cuatro regiones bien diferenciadas, con recuadros en trazo discontinuo. Cada una de ellas delimita al paquete donde están definidas las clases que hay dentro de él, a excepción de *BArkanoid.java* que, por ser el

MIDlet de la aplicación, se ha situado aparte para facilitar la interpretación del diagrama y destacar a esta clase como el eje central de la implementación.

Cada una de las pantallas del sistema de ventanas de la aplicación, se define, como ya se ha comentado, en el paquete *principal*. El *MIDlet* es quien centraliza la gestión de todas ellas, de manera que desde él se define una instancia para cada una de estas pantallas. Puede observarse, además, que cada una de ellas es una subclase *GUICanvas.java*. Igualmente, deberá crear un maestro o un esclavo en función de si el terminal sobre el que se ejecuta la aplicación desea crear una partida o unirse a una ya creada, respectivamente. Y también definirá un atributo que se encargue de administrar adecuadamente cada uno de los efectos de sonido y melodías que componen el videojuego.

Por otro lado, en el paquete *bluetooth* se definen las dos clases que implementan cada rol dentro de la comunicación por el enlace Bluetooth: *Maestro.java* y *Esclavo.java*, este último con la clase *Listener* implementada dentro de ella (por eso no aparece como un fichero *.java* aparte en el diagrama de clases implementadas). Ambas tendrán en común una serie de características, contenidas en *Extremo.java*, clase de la que heredan. Además, se precisa una clase que defina los diferentes tipos de mensajes que se pueden transmitir entre un extremo y otro de la comunicación. Esta clase es *Mensajes.java*. No necesita instanciarse, simplemente define los posibles mensajes a intercambiar entre ambas partes de la conexión.

Una vez se ha decidido el papel que se va a adoptar en la comunicación, cada extremo lanzará su propia ejecución de la partida a través de una instancia de *Juego.java*. Esta clase será la que controle y administre todos los elementos que contiene una partida del videojuego, es decir, las pelotas, las naves, los mapas y los eventuales *power-ups* asociados a algunos ladrillos del mapa sobre el que se desarrolla la partida en cada momento.

Finalmente, existe una serie de clases que completan la funcionalidad requerida para esta aplicación: *Fondo.java* implementa la imagen de fondo que aparecerá tanto en los menús del programa como en cada mapa; *Fichero.java* proporciona las herramientas y los mecanismos necesarios para interpretar correctamente los ficheros de entrada de la

aplicación y cuya funcionalidad se requerirá para leer los datos de cada pantalla de menú y la información de cada mapa del videojuego; *Constantes.java* define una serie de valores de referencia, de utilidad para el conjunto de la aplicación; y finalmente, *GestorRMS.java* será el encargado de leer y almacenar correctamente las mejores puntuaciones obtenidas en cada modo de juego y el idioma seleccionado para la aplicación.

4.2.2. DIAGRAMA DE ESTADOS DE LA APLICACIÓN

El diagrama de estados que representa de manera esquemática el funcionamiento global de la aplicación es el que se muestra en la figura 4.6. Básicamente, muestra de forma simplificada la descripción del funcionamiento realizada en el capítulo 3.

Tras las primeras pantallas de configuración de sonido, advertencia de Bluetooth activado y la pantalla de presentación con la imagen de la portada del videojuego, se llega al menú principal de la aplicación, desde el que se tiene la posibilidad de crear una partida, unirse a una ya creada, entrar en el menú opciones para realizar otras acciones o salir de la aplicación. En el caso en que se decida iniciar una nueva partida, se elige el modo de juego (cooperativo o *deathmatch*) y se inicia la espera para que se conecte el terminal esclavo; por otro lado, si se desea unirse a una partida previamente creada, es preciso iniciar un proceso de búsqueda de dispositivos que ofrezcan el servicio de conexión Bluetooth, para localizar al terminal que ha creado la partida e iniciarla; si, por el contrario, se desean realizar otras acciones, desde el menú de opciones se podrá activar / desactivar el sonido de la aplicación, consultar la ayuda, seleccionar otro idioma o mirar las puntuaciones más altas registradas para los dos modos de juego posibles.

Con la partida en curso, hay dos formas de salir de ella: abandonándola de forma manual, antes de que acabe, o terminándose por agotamiento de las vidas de al menos uno de los dos jugadores. En el primer caso, se muestra el aviso de desconexión en el otro extremo, y posteriormente, el menú principal, al igual que para el que forzó la salida de la partida; en el otro caso, se mostrará la puntuación final obtenida y se guardará si se ha alcanzado un nuevo record, para después preguntar si se desea jugar otra vez o abandonar e ir, como antes, al menú principal.

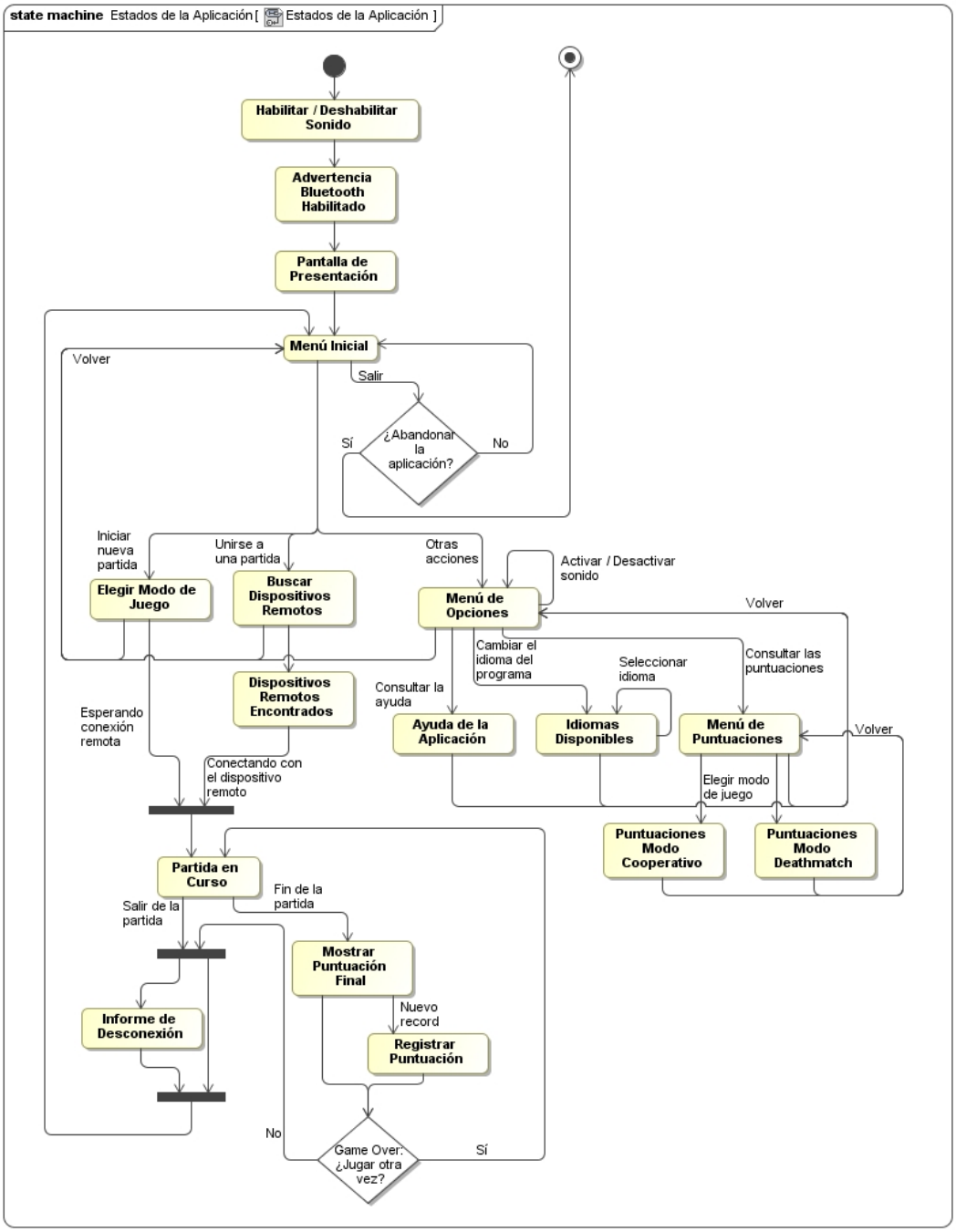


Figura 4.6. Diagrama de estados de la aplicación

4.3. ESTRUCTURA MULTITHREAD O MULTIHILO

Un aspecto a destacar de este proyecto es el empleo de las capacidades *multihilo* de Java. La aplicación hace uso de los *threads* Java para separar diferentes ámbitos de ejecución del videojuego. En la figura 4.7 se puede observar esta estructura.

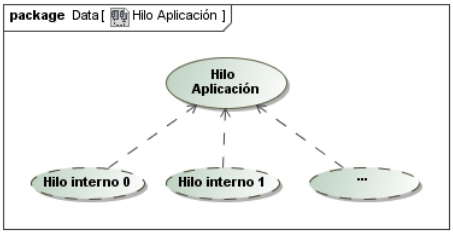


Figura 4.7. Hilos internos al hilo de aplicación

Los programas *multithread* o *multihilo* parten de la idea de multitarea a su nivel más bajo: programas individuales que darán la impresión de llevar a cabo varias labores a la vez. Cada una de estas labores recibe el nombre de *thread* o hilo. Los programas capaces de ejecutar más de un *thread* a la vez reciben el nombre de *multithread* o *multihilo*. Se puede pensar que cada *thread* se está ejecutando en un contexto separado, cada uno con su propia CPU, registros, memoria, etc.

La aplicación constará de varios hilos a lo largo de su tiempo de ejecución. La razón por la que se hace uso de esta capacidad multihilo en el videojuego desarrollado, es, por un lado, para poder ejecutar el bucle principal del juego en un hilo aparte, sin que ello pueda provocar el bloqueo del dispositivo y, por otro, para permitir, durante la partida, la recepción de mensajes desde el terminal remoto sin interrumpir el desarrollo normal del programa. La figura 4.8 muestra el diagrama de secuencia que representa de manera esquemática los diferentes *threads* que componen la ejecución del videojuego.

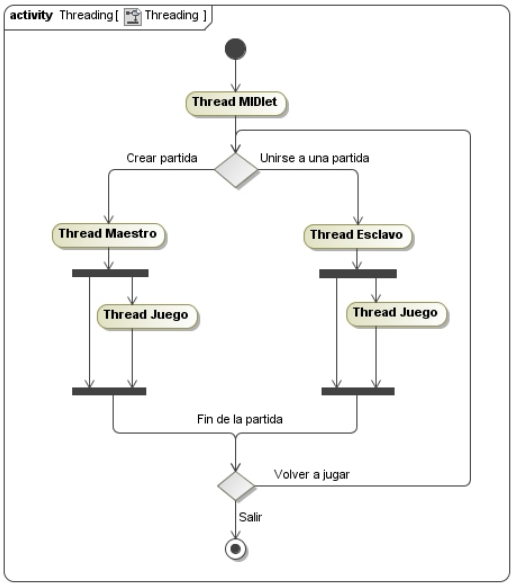


Figura 4.8. Esquema multihilo de la aplicación

El primer hilo que se ejecuta en la aplicación proviene de la clase de arranque en J2ME. Esta clase es una clase que hereda de la superclase abstracta *MIDlet* y que obliga, como tal, a implementar una serie de métodos –el de arranque (*startApp*), pausa (*pauseApp*) y destrucción (*destroyApp*) de la aplicación–. Posteriormente, al iniciar una nueva partida en el terminal, ya sea creándola o uniéndose a una ya creada previamente, se genera otro hilo de ejecución, que será el del terminal maestro o el del esclavo según se haya creado o no la partida, respectivamente, para establecer la comunicación y poder recibir los mensajes del terminal remoto sin que esto suponga el bloqueo de la ejecución normal del programa. Cada uno de estos dos hilos generará, cuando proceda, un nuevo hilo que se encargará de ejecutar el bucle principal de la partida mientras ésta esté en curso. Después de acabar la partida, se podrá crear una nueva o unirse a una partida creada por el otro terminal, por lo que el proceso de crear un nuevo hilo para ejecutar el bucle principal se repetiría; en caso contrario, al salir de la aplicación finalizarían los hilos del terminal maestro y del esclavo, así como el hilo primario.

Para poder disponer de capacidad *multithread*, Java ofrece dos técnicas para crear un *thread*. Por un lado, a través del API *java.lang.Thread* que proporciona, y por otro, mediante la interfaz *java.lang.Runnable*. En el primer caso, la clase que necesite definir un nuevo hilo hereda de *java.lang.Thread* e incluye en *run()* la lógica que se desee ejecutar en dicho hilo. El código que se muestra a continuación ilustra de manera esquemática la estructura que adoptaría esta primera alternativa:

```
public class ThreadEjemplo extends Thread {

        public ThreadEjemplo(){
                super();
                …
        }

        public void run(){
                …
        }

        public void iniciarThread(){
                new ThreadEjemplo().start();
        }
}
```

El método *run()* es invocado cuando se inicia el *thread* (mediante una llamada al método *start()* de la clase *Thread*. El *thread* se inicia con la llamada al método *run()* y

termina cuando termina éste. La otra opción es implementar la interfaz *java.lang.Runnable*. En ese caso, simplemente fuerza solo la implementación del método *run()*. Para ello, es preciso crear una instancia de la clase *Thread* pasándole el objeto que implementa *Runnable* como parámetro. En el siguiente código se expone el esquema que seguiría esta segunda opción:

```
public class ThreadEjemplo implements Runnable {

        public void run(){
                …
        }

        public void iniciarThread(){
                new Thread(new ThreadEjemplo()).start();
        }
}
```

Dado que las clases que necesitan definir un nuevo hilo en el código ya heredan de otra clase (por un lado, las clases *Maestro.java* y *Esclavo.java*, de las que se hablará más en profundidad en la sección de Bluetooth, heredan de *Extremo.java*, y por otro, *Juego.java* es una subclase de *javax.microedition.lcdui.game.GameCanvas*) y que la herencia múltiple no está contemplada en Java, la alternativa más oportuna para este caso es la de implementar la interfaz *java.lang.Runnable*.

4.3.1. CICLO DE VIDA DE UN THREAD

La figura 4.9 resume el ciclo de vida de un *thread*.

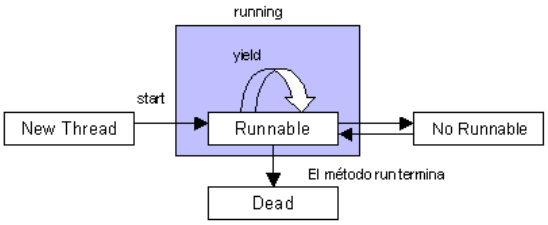


Figura 4.9. Ciclo de vida de un thread

Cuando se instancia la clase *Thread* (o una subclase) se crea un nuevo *Thread* que está en su estado inicial (‘*New Thread*’ en el gráfico). En este estado es simplemente un objeto más. No existe todavía el *thread* en ejecución. El único método que puede invocarse sobre él es el método *start()*. Cuando se invoca el método *start()*

sobre el *thread* el sistema crea los recursos necesarios, lo planifica (le asigna prioridad) y llama al método *run()*. En este momento el *thread* está corriendo (en la imagen, se encuentra en el estado '*Runnable*'). Si el método *run()* invoca internamente el método *sleep()* o *wait()*, o el *thread* tiene que esperar por una operación de entrada/salida, entonces el *thread* pasa al estado '*No Runnable*' (no ejecutable) hasta que la condición de espera finalice. Durante este tiempo el sistema puede ceder el control a otros *threads* activos. Por último, cuando el método *run()* finaliza el *thread* termina y pasa al estado '*Dead*' (Muerto).

4.4. UTILIDADES GENERALES PARA LA APLICACIÓN

Bajo este epígrafe se incluyen un elenco de clases de utilidad para todo el código, como las herramientas que dan formato gráfico y sonoro a la aplicación, así como los mecanismos de lectura / escritura de datos y la definición de algunas constantes de referencia.

4.4.1. INTERFAZ GRÁFICA

La interfaz gráfica de usuario (en inglés, GUI, *Graphics User Interface*) en un programa informático puede definirse como la herramienta que permite una interacción persona-máquina amigable, a través del uso y la representación del lenguaje visual. Para el videojuego que aquí se ha desarrollado, se ha optado por dar un formato único a todas las pantallas de menú del mismo, de modo que todas ellas luzcan de manera uniforme, proporcionando una interfaz gráfica coherente y ordenada. Para ello, se ha implementado la clase *GUICanvas.java*, que fija todos los parámetros relativos a la apariencia y el aspecto gráfico (tipo de letra, tamaño, colores,…) y proporciona métodos y atributos que son de utilidad para las clases que heredan de ella e implementan cada pantalla de la aplicación. En la figura 4.10 puede apreciarse el diagrama de esta clase.

La clase *Canvas* es la superclase de todas las pantallas que usan las APIs de bajo nivel. Permite dibujar cualquier cosa por pantalla (*canvas*, en inglés, significa lienzo) y manejar eventos de bajo nivel. Estas dos características son las que permitirán diseñar una interfaz gráfica personalizada y detectar las pulsaciones de teclas para moverse por las pantallas de la aplicación. Todas las clases que heredan de *GUICanvas.java*, por

ejemplo, *MenuInicialCanvas.java*, *OpcionesCanvas.java*… se encuentran en el paquete *principal*, el mismo en el que se ubica el *MIDlet* de la aplicación.

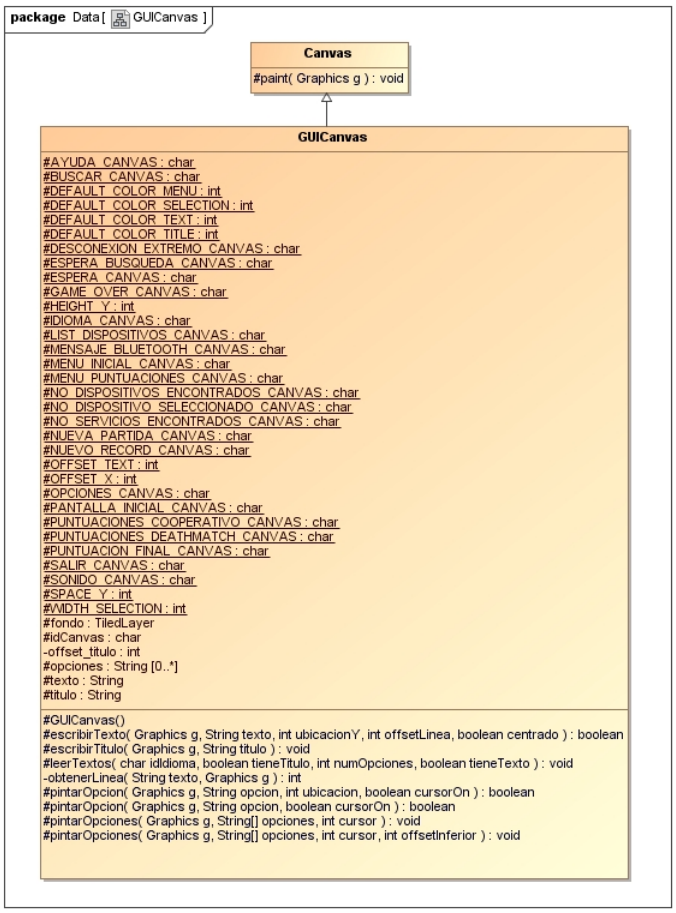


Figura 4.10. Diagrama de la clase GUICanvas.java

La clase define un amplio número de constantes, todas ellas destinadas a dar formato a la interfaz gráfica de cada pantalla y a proporcionar identificadores para cada una de ellas. Su acceso es protegido, esto es, accesibles únicamente desde las clases que hereden de *GUICanvas.java*. Aunque no todas se instancian desde las clases hijas, al diseñar la clase padre se ha interpretado que estas constantes deberían poder ser igualmente accesibles desde ellas. A continuación se analizan de forma agrupada, según su función:

- Identificadores de pantalla: cada identificador reconoce de forma unívoca a cada pantalla de la aplicación (y por extensión a la clase que la implementa) mediante un simple carácter. Éste se asignará en el constructor de la clase que hereda de *GUICanvas.java* para implementar la pantalla correspondiente. Servirá para

poder recuperar posteriormente del sistema de ficheros el texto de las opciones y los mensajes informativos, y el título, si lo hubiera, de la pantalla en cuestión, en el idioma seleccionado en ese momento (esto se tratará con mayor detalle en la sección dedicada a los ficheros de la aplicación). A continuación, se muestra un ejemplo de inicialización de una de las pantallas del programa, en concreto la del menú inicial (*MenuInicialCanvas.java*).

```
//Número de opciones de las que consta este menú
private final static int NUM_OPCIONES = 4;

/** Crea una nueva instancia de MenuInicialCanvas */
public MenuInicialCanvas(){

        this.idCanvas = this.MENU_INICIAL_CANVAS;
        this.opciones = new String[NUM_OPCIONES];
}
```

Puede apreciarse cómo es el proceso de asignación de estos identificadores (a través del atributo **idCanvas**) y también el establecimiento del número de opciones que habrá disponibles en el menú. En la tabla 4.1 se muestra una lista donde se recogen todos los identificadores, relacionándolos con la pantalla a la que hacen referencia.

Identificador	Figura que representa la pantalla
AYUDA_CANVAS	Figura 3.14.
BUSCAR_CANVAS	Figura 3.7.
DESCONEXION_EXTREMO_CANVAS	Figura 3.24.
ESPERA_BUSQUEDA_CANVAS	Figura 3.8.
ESPERA_CANVAS	Figura 3.6.
GAME_OVER_CANVAS	Figura 3.23.
IDIOMA_CANVAS	Figura 3.16.
LIST_DISPOSITIVOS_CANVAS	Figura 3.9.
MENU_INICIAL_CANVAS	Figura 3.4.
MENSAJE_BLUETOOTH_CANVAS	Figura 3.2.
MENU_PUNTUACIONES_CANVAS	Figura 3.17.
NO_DISPOSITIVOS_ENCONTRADOS_CANVAS	Figura 3.10.
NO_DISPOSITIVO_SELECCIONADO_CANVAS	Figura 3.11.
NO_SERVICIOS_ENCONTRADOS_CANVAS	Figura 3.12.
NUEVA_PARTIDA_CANVAS	Figura 3.5.
NUEVO_RECORD_CANVAS	Figura 3.22.
OPCIONES_CANVAS	Figura 3.13. / Figura 3.15
PANTALLA_INICIAL_CANVAS	Figura 3.3.
PUNTUACIONES_COOPERATIVO_CANVAS	Figura 3.18.a
PUNTUACIONES_DEATHMATCH_CANVAS	Figura 3.18.b
PUNTUACION_FINAL_CANVAS	Figura 3.21.
SALIR_CANVAS	Figura 3.19.
SONIDO_CANVAS	Figura 3.1.

Tabla 4.1. Identificadores de las pantallas implementadas

- Identificadores de colores: existe un pequeño grupo de constantes que establecen el juego de colores que se utiliza en todas las pantallas. En total, son cuatro: uno para dar color a cada opción de menú; uno para el borde del recuadro de la opción seleccionada; uno para el color de los textos; y uno para el borde del recuadro que contiene el título de la pantalla. En la tabla 4.2 se representa una lista que recopila los identificadores de estos colores, indicando de qué colores se trata y para qué son utilizados.

Identificador	Función	Color
DEFAULT_COLOR_MENU	Fondo de opciones de menú	Azul oscuro
DEFAULT_COLOR_SELECTION	Borde de opción seleccionada	Gris claro
DEFAULT_COLOR_TEXT	Textos	Azul celeste
DEFAULT_COLOR_TITLE	Borde del título	Rosa

Tabla 4.2. Identificadores de los colores empleados

- Indicadores de espaciado: estas constantes determinan qué espaciado deberán respetar las diferentes opciones y líneas de texto que componen cada pantalla para ser correctamente ubicadas en ellas. La tabla 4.3 muestra una lista que abarca los identificadores de estos indicadores, un breve comentario explicativo acerca de su utilidad en el código y su valor numérico en píxeles.

Identificador	Función	Valor
HEIGHT_Y	Define la altura de los recuadros utilizados	25
OFFSET_TEXT	Offset vertical para ubicar el texto de cada opción	10
OFFSET_X	Determina los márgenes laterales para textos y recuadros	10
SPACE_Y	Espacio vertical entre cada recuadro y cada línea de texto	15
WIDTH_SELECTION	Grosor del borde de títulos y opciones seleccionadas	4

Tabla 4.3. Identificadores de los indicadores de espaciado

Se observa que la altura de cada recuadro es de 25 píxeles. La anchura, sin embargo, no viene determinada; ésta dependerá de la anchura de la pantalla de cada terminal, dejando 10 píxeles de margen a cada lado, según indica *OFFSET_X*. Dos recuadros consecutivos estarán separados entre sí, al menos, *SPACE_Y* píxeles uno del otro en el eje vertical, a no ser que se trate de un recuadro que se ubique aparte del resto, en cuyo caso la distancia será mayor.

Los textos de los recuadros (opciones y títulos, principalmente), van centrados. Por otra parte, los que van fuera de ellos (mensajes de advertencia, el texto de la

ayuda,…), si no van centrados, dejarán, como mínimo, *OFFSET_X* píxeles de margen a cada lado.

Por otro lado, la clase define a su vez una serie de atributos, la mayoría de ellos también de acceso protegido, como las constantes anteriormente analizadas, para permitir el acceso desde las clases que hereden de ella. A continuación, se detallan cuáles son, y se explica su utilidad en el código:

- **fondo**: este atributo guarda la imagen de fondo del mapa correspondiente. Se trata de una imagen compuesta, formada a base de pequeñas imágenes más sencillas, llamadas baldosas, que permiten crear grandes áreas sin el problema de manejar una imagen de grandes dimensiones de memoria. El fondo del mapa se genera desde el constructor, a partir de las dimensiones de pantalla que se pasan por parámetro para determinar cuántas celdas se cogen y cómo se distribuyen para dar lugar a dicha imagen.
  Como todas las pantallas tienen el mismo fondo, éste se genera directamente desde el constructor de la clase padre, o sea, la propia *GUICanvas.java*. Posteriormente, el atributo será invocado desde los métodos *paint(Graphics g)* de las respectivas clases que heredan de ésta para representarlo por pantalla.

- **idCanvas**: identifica de forma unívoca a la pantalla que hereda de *GUICanvas.java*. Su valor, como ya se ha comentado anteriormente, será el de algún identificador de pantalla, de modo que este atributo se inicializa desde los constructores de las clases que implementen cada pantalla de la aplicación.

- **offset_titulo**: indica el *offset* vertical que hay que aplicar en el caso de que la pantalla en cuestión disponga de título. Es un atributo privado. Las pantallas que contengan un título (ubicado siempre en la parte superior) harán que este atributo pase de valer cero a valer *SPACE_Y + HEIGHT_Y*, es decir, el equivalente al alto de un recuadro y la distancia de separación entre recuadros. Esta información será de utilidad a la hora de centrar en pantalla los recuadros que contienen las opciones de menú.

- **opciones**: este atributo es un vector que almacena las opciones de menú de cada pantalla y, eventualmente, pequeños mensajes de advertencia que vayan dentro de recuadros, como el de indicación de que el proceso de búsqueda se ha iniciado (figura 3.7). Se inicializa desde el constructor de cada clase que hereda de *GUICanvas.java* y necesita representar por pantalla alguna opción o mensaje contenido en un recuadro. El tamaño de este *array* dependerá, naturalmente, del número de opciones y de mensajes que se quiera representar. Esta cantidad vendrá dada por la constante *NUM_OPCIONES*, definida para cada clase hija.

- **texto**: almacena el texto que se va a representar por pantalla fuera de cualquier tipo de recuadro (el texto de la ayuda y los mensajes de advertencia que no vayan dentro de recuadros, como el de aviso de que el dispositivo remoto ha abandonado la partida (figura 3.23).

- **titulo**: contiene el texto del título de la pantalla correspondiente. Éste irá siempre contenido en un recuadro con un borde de color rosado, en la parte superior de la pantalla.

Esta clase proporciona también una serie de métodos para poder representar los diferentes elementos (opciones, mensajes, títulos…) que integran cada pantalla de la aplicación. A continuación, se describe cada uno de ellos, ordenándolos según sus características y sus funcionalidades:

- *void paint(Graphics g)*: este método es un método abstracto de la clase *Canvas*, y por tanto, ha de ser implementado por aquellas clases que hereden de ella (en este caso, *GUICanvas.java*). No obstante, la implementación de dicho método en esta clase se deja vacía para que sean las clases hijas quienes la completen según sus necesidades, llamando al resto de métodos que se declaran en la clase padre.

- Método para leer todo tipo de textos pertenecientes a una misma pantalla. Para recuperar de un fichero de texto el título, las opciones, y en general, los textos necesarios para una pantalla de terminada, se utiliza el siguiente método:

void leerTextos(char idIdioma, boolean tieneTitulo, int numOpciones, boolean tieneTexto)

Para efectuar la correspondiente lectura, se deberá especificar el idioma en que se deben presentar los textos, aclarar si la pantalla dispone de título, indicar cuántas opciones (o, en general, líneas de texto enmarcadas en un recuadro) contiene la pantalla, así como si también hay texto fuera de recuadros. Este método será invocado por todas las clases que implementen pantallas, pues todas ellas contienen al menos un título, una opción, un mensaje de aviso o, en general, cualquier tipo de texto.

- Métodos para representar las opciones de menú. En total son cuatro métodos, que se detallan a continuación:

boolean pintarOpcion(Graphics g, String opcion, boolean cursorOn)

boolean pintarOpcion(Graphics g, String opcion, int ubicacion, boolean cursorOn)

El primer método se encarga de pintar una única opción en el centro de la pantalla, mientras que el segundo hace lo propio, pero con la posibilidad de indicar una ubicación en la misma, es decir, la altura a la que se desea posicionar la opción. Además, la opción podrá o no estar señalada, dependiendo del valor del parámetro **cursorOn**. En ambos casos, se devuelve un booleano que indica si la operación se ha realizado con éxito (*true*) o no (*false*). Generalmente, son de utilidad para representar las opciones de menú que se desmarcan del resto de opciones en una pantalla o que, siendo la única opción de una determinada pantalla, se desea ubicarla en una posición distinta de la central (como ocurre con la opción *Volver* en muchas pantallas).

void pintarOpciones(Graphics g, String[] opciones, int cursor)

void pintarOpciones(Graphics g, String[] opciones, int cursor, int offsetInferior)
A diferencia de los dos métodos anteriores, éstos imprimen grupos de opciones. En ambos casos, se trata de representarlas centradas por pantalla. La diferencia es que el segundo método contempla un posible *offset* vertical, motivado por una eventual opción situada en la parte inferior de la pantalla, para centrar correctamente el grupo de opciones. En cualquier caso, ahora la opción señalada

puede ser una entre varias, por lo que, en lugar de utilizar un booleano que indique si está señalada o no, se pasa por el parámetro **cursor** el índice de la opción que lo está (si ninguna lo está, su valor es *-1*).

- Método para escribir un título en pantalla. Para cumplir esta función, se dispone del siguiente método:

  void escribirTitulo(Graphics g, String titulo)
  Este método, naturalmente es instanciado desde las clases que implementen aquellas pantallas que contengan título (por ejemplo, la pantalla para habilitar o no el sonido, mostrada en la figura 3.1 y que aparece nada más arrancar la aplicación).

- Métodos para escribir por pantalla textos fuera de recuadros. Los siguientes métodos permitirán representar por pantalla aquellos textos que no estén contenidos en ningún recuadro (mensajes de aviso y el texto de la ayuda).

  boolean escribirTexto(Graphics g, String texto, int ubicacionY, int offsetLinea, boolean centrado)
  int obtenerLinea(String texto, Graphics g)
  El primero es el método visible por las clases que heredan de *GUICanvas.java*, de modo que para escribir uno de estos textos tienen que invocarlo, indicando el texto completo a representar por pantalla, la altura a partir de la cual se empieza a escribir, a partir de qué línea (si es que el texto pudiera ocupar más de una línea en pantalla) se desea mostrar por pantalla y si la disposición de este texto es centrada o no.
  Dado que es una clase que hereda de *Canvas*, los textos no son entendidos como en un contexto tradicional de procesadores de texto, donde al abarcar todo el ancho de la pantalla se pasa directamente a escribir en la siguiente línea, y así sucesivamente… aquí se trata de 'pintar' sobre algo que simula ser un lienzo (*canvas*), y por tanto, todo control que se quiera hacer sobre el espacio ocupado por lo que se pinta en él, hay que hacerlo manualmente. Es decir, hay que implementar un método que sea capaz de determinar, para cada línea, cuál es el mayor fragmento de texto tomado de un texto más grande que cabe en pantalla.

Eso es lo que hace el segundo método de este punto. A partir del texto que se proporciona como parámetro, devuelve la posición del otro extremo de dicho texto para obtener la línea más grande que cabe en pantalla. Repitiendo la misma operación sucesivas veces para el texto que aún no ha sido fragmentado en líneas, pueden obtenerse todas las líneas que serán representadas por pantalla.

4.4.2. FONDO

La clase responsable de generar el fondo de pantalla de la aplicación se denomina *Fondo.java*. Si bien puede entenderse como una parte más del aspecto gráfico del programa, no se ha incluido dentro de la clase que implementa la interfaz gráfica (*GUICanvas.java*) porque ésta se ocupa de dar formato y estilo común a cada pantalla de menú de la aplicación, mientras que el fondo de pantalla, por su parte, es parte integrante tanto de los menús como del propio videojuego una vez se ha iniciado una partida. En la figura 4.11 se observa el diagrama de esta clase.

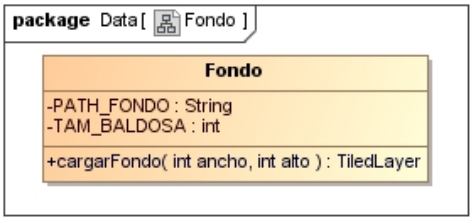


Figura 4.11. Diagrama de la clase Fondo.java

Consiste únicamente en un método que genera el fondo para una pantalla con unas dimensiones determinadas. Asimismo, la imagen original que toma para generar el fondo se encuentra en el *path* indicado por la constante *PATH_FONDO*. El fondo que genera el único método de la clase consiste en un objeto de la clase *TiledLayer*. Esta clase permitirá rellenar grandes áreas de contenido gráfico sin el problema que supondría almacenar una imagen de grandes dimensiones en memoria. Para ello, la imagen original se subdivide en baldosas (en inglés, *tile*) de menor tamaño, concretamente el indicado por la constante *TAM_BALDOSA*, a partir de las cuales se puede generar una imagen de mayores dimensiones. Es decir, tener un objeto *TiledLayer* equivale a tener una serie de piezas de un rompecabezas que se podrán combinar y repetir convenientemente de manera que generen una imagen más grande con el aspecto deseado, lo que resulta especialmente útil para generar imágenes como el

fondo de la aplicación. La figura 4.12 muestra un ejemplo de cómo se genera una imagen mayor a partir de pequeñas baldosas de una imagen original de mayor tamaño.

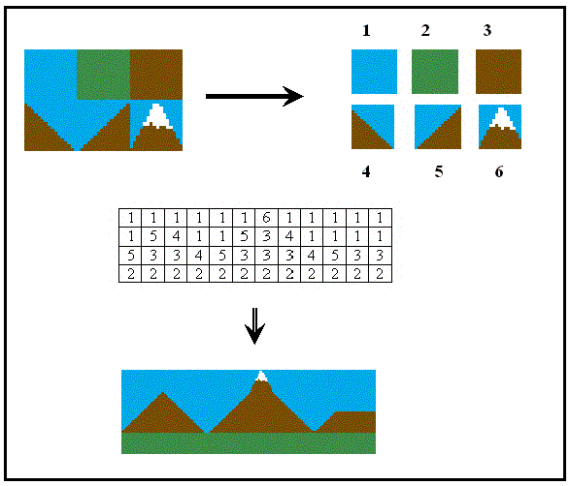


Figura 4.12. Ejemplo de generación de una imagen mediante TiledLayer

La imagen original es la que se encuentra en la parte superior izquierda de la figura. Esta imagen se divide en seis imágenes más pequeñas, todas de igual tamaño, numeradas de izquierda a derecha y de arriba a abajo, como se muestra en la parte superior derecha. Estas imágenes serán las baldosas que darán lugar a la imagen final (parte inferior de la figura). Dicha imagen será el objeto *TiledLayer*, que se compondrá de una serie de celdas, cuya distribución se hace en base a una matriz bidimensional. Las dimensiones de esta matriz se especifican en el constructor de la clase *TiledLayer*, junto con la imagen original y las dimensiones de cada celda, que han de ser iguales a las de las baldosas. El contenido de cada una de las celdas se indicará mediante un entero que hace referencia a una baldosa en concreto. En el ejemplo, se pretende construir una imagen compuesta de cuatro filas de baldosas por doce columnas, por lo que una forma práctica para rellenar las celdas del objeto *TiledLayer* sería leyendo una matriz de esas mismas dimensiones, en la que el contenido de cada elemento se corresponde con un entero que identifica a una baldosa.

En el código implementado para generar el fondo de la aplicación, el tamaño de cada baldosa viene determinado por *TAM_BALDOSA*, como ya se ha comentado anteriormente, mientras que las dimensiones de la imagen final dependerán del alto y del ancho de la pantalla de cada dispositivo, que serán los parámetros que determinarán el número de filas y columnas necesarios que debe componer el objeto *TiledLayer*.

4.4.3. LECTURA / ESCRITURA DE DATOS

Cualquier aplicación informática mínimamente ambiciosa, necesitará poder leer unos datos de entrada, procesarlos y ser capaz de almacenar de forma persistente los datos producidos para su posterior consulta o utilización, ofreciendo así un servicio de calidad al usuario. En este proyecto se ha hecho una distinción entre datos de solo lectura –entre los que se cuentan todas las opciones, textos y mensajes de menú de cada pantalla de la aplicación y la información relativa a cada mapa del conjunto de mapas del videojuego– y datos de lectura/escritura –en este grupo se engloban las puntuaciones más altas en cada modo de juego y el idioma seleccionado para la aplicación–. Los primeros, dado que son datos de solo lectura, serán leídos desde los correspondientes ficheros de texto, incluidos dentro del paquete *recursos*. Los segundos, por su parte, se almacenarán en registros de una base de datos, en una zona de memoria dedicada para este propósito, mediante RMS (*Record Management System*). A continuación se analizarán con mayor detenimiento cómo se ha implementado el tratamiento de cada uno de estos tipos de datos.

4.4.3.1. Ficheros

Los títulos, opciones, mensajes y textos que se muestran por pantalla en el idioma seleccionado, además de la información relativa a cada mapa del conjunto de mapas del videojuego, provienen de ficheros donde se almacena dicha información. Para acceder a ella e interpretar correctamente cada dato, existe una clase que se encarga específicamente de estas funciones, *Fichero.java*, incluida dentro del paquete *util*. El diagrama de esta clase se muestra en la figura 4.13.

La clase dispone de una serie de constantes de uso privado, cuya utilización se describirá a continuación, además de otras tres constantes a las que se necesitará acceder desde otras clases de la aplicación. Pueden resumirse en cuatro grandes grupos, a saber:

- Directorios o *paths*.
- Códigos ASCII de los identificadores de los ladrillos.
- Constantes auxiliares para interpretar los datos de fichero.
- Identificadores para distinguir el tipo de dato a recuperar.

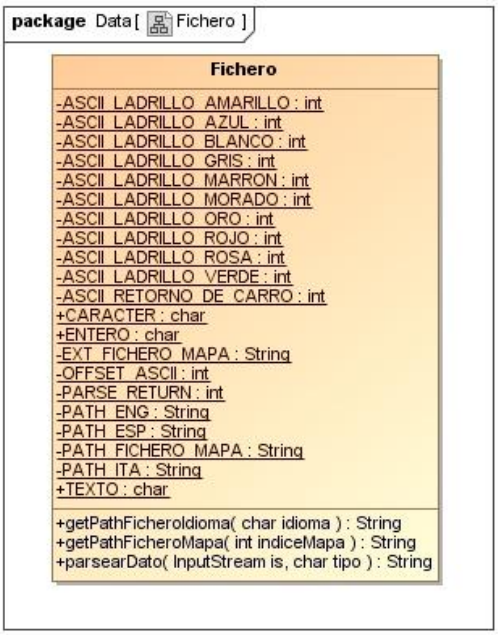


Figura 4.13. Diagrama de la clase Fichero.java

- Directorios o *paths*: en estas constantes se almacenan las rutas de los ficheros donde está contenida la información que se requiere para el programa. Por un lado, están los ficheros que proporcionan la estructura de cada mapa, y por otro, los ficheros encargados de facilitar las opciones, textos, etc. en el idioma correspondiente. Así, para este primer grupo, se definen dos constantes: *PATH_FICHERO_MAPA* y *EXT_FICHERO_MAPA*. La primera proporciona la carpeta donde se encuentran los ficheros de los mapas, mientras que la segunda indica la extensión de éstos. Combinando ambos identificadores, junto con el entero que se pasa por parámetro desde el método *getPathFicheroMapa(int indiceMapa)*, que especifica el mapa, dentro de los diez que componen el videojuego, se obtiene el correspondiente *path* completo del mapa deseado.

  Los textos que aparecen en las pantallas y menús de la aplicación, por su parte, se obtienen del fichero del idioma previamente indicado desde el método *getPathFicheroIdioma(char idioma)*. Según el valor que se pasa por parámetro, se procederá a abrir uno u otro fichero, cuyos *paths* vienen indicados por los identificadores que se recogen en la tabla 4.4. En ella se especifica, además, el valor que hay que pasar por el método indicado para obtenerlos y los ficheros a los que permiten acceder.

Path	Id. que lo habilita	Fichero al que permite acceder
PATH_ENG	Constantes.*LANG_ENG*	Fichero con los textos en inglés
PATH_ESP	Constantes.*LANG_ESP*	Fichero con los textos en español
PATH_ITA	Constantes.*LANG_ITA*	Fichero con los textos en italiano

Tabla 4.4. Identificadores de los paths de los ficheros de texto

- Códigos ASCII de los identificadores de los ladrillos: en los ficheros de los mapas, cada ladrillo viene representado por un carácter que lo distingue del resto de ladrillos posibles. Estos caracteres, a su vez, están asociados a un número, que se corresponde con el código ASCII que identifica a cada carácter. Dado que el proceso de lectura de caracteres que se efectúa desde el método *parsearDato(InputStream is, char tipo)* se hace *byte* a *byte*, para facilitar la identificación de los ladrillos cuando éstos son leídos desde el fichero del mapa correspondiente se han definido una serie de constantes que almacenan el código ASCII que representa a cada uno de los identificadores de los ladrillos utilizados en el videojuego. En la tabla 4.5 se resumen estos identificadores, junto con su valor y el carácter al que representa cada código ASCII.

Identificador	Código ASCII	Carácter que representa
ASCII_LADRILLO_AMARILLO	*65*	*A*
ASCII_LADRILLO_AZUL	*90*	*Z*
ASCII_LADRILLO_BLANCO	*66*	*B*
ASCII_LADRILLO_GRIS	*71*	*G*
ASCII_LADRILLO_MARRON	*77*	*M*
ASCII_LADRILLO_MORADO	*80*	*P*
ASCII_LADRILLO_ORO	*79*	*O*
ASCII_LADRILLO_ROJO	*82*	*R*
ASCII_LADRILLO_ROSA	*83*	*S*
ASCII_LADRILLO_VERDE	*86*	*V*

Tabla 4.5. Identificadores de los códigos ASCII que representan a los ladrillos

- Constantes auxiliares para interpretar los datos de fichero: adicionalmente, la clase cuenta con una serie de constantes de uso privado, dedicadas a facilitar el proceso de interpretación de los datos contenidos en los ficheros. Son las siguientes:

  ASCII_RETORNO_DE_CARRO: cada línea de fichero indica un dato (una opción, un texto, un título, la coordenada de un ladrillo,…), de modo que para distinguir el final de un dato del comienzo de otro, bastará con distinguir la línea

que se está leyendo. La lectura del código ASCII del retorno de carro será la que indique el final de una línea de texto. Esta constante almacena dicho valor.

OFFSET_ASCII: cuando el dato que se quiere recuperar es un entero, el código ASCII del carácter que representa a ese número no coincide numéricamente con dicho entero. Una opción es obtenerlo a partir de su propio código ASCII. Para ello, se sabe que el código ASCII del carácter que representa el '0' es el 48, el que representa el '1' es el 49, y así sucesivamente… de modo que para obtener el entero correspondiente bastará con restar 48 al ASCII obtenido. El valor de esta constante es justamente 48.

PARSE_RETURN: como ya se ha comentado, cada dato va en una línea aparte. Sin embargo, hay veces en las que un mismo dato (una cadena de texto) puede contener varias líneas. Para poder hacer esto posible, en lugar de componer un mismo dato a partir de varias líneas, éstas se almacenan todas en una misma línea de fichero, separándolas unas de otras mediante un carácter especial, reservado para indicar dónde termina una y dónde empieza la siguiente. Esta constante es la encargada de hacer esta separación.

La tabla 4.6 sirve a modo de resumen de estos identificadores.

Identificador	Valor	Función que realiza
ASCII_RETORNO_DE_CARRO	*13*	Código ASCII del retorno de carro, indica el final de línea
OFFSET_ASCII	*48*	Offset para obtener el entero del carácter del código ASCII
PARSE_RETURN	*	Carácter que en una cadena de texto indica un final de línea

Tabla 4.6. Identificadores de las constantes auxiliares

- Identificadores para distinguir el tipo de dato a recuperar: estas constantes son de uso público, puesto que son las que utilizan otras clases para indicar qué tipo de dato se desea recuperar del fichero. Hay de tres tipos, según se muestra en la tabla 4.7.

Identificador	Descripción
CARACTER	Se desea recuperar un carácter (ej: un identificador de ladrillo, de pantalla)
ENTERO	Se desea recuperar un número entero (ej: una coordenada, el número de ladrillos)
TEXTO	Se desea recuperar una cadena de texto (ej: un mensaje, el texto de la ayuda)

Tabla 4.7. Identificadores de los tipos de datos contenidos en los ficheros

4.4.3.1.1. Formato de los ficheros

Como ya se ha comentado en la presente sección, existen ficheros que almacenan las opciones, textos y títulos de las pantallas y menús de la aplicación, a la vez que se dispone de un fichero por cada mapa del videojuego. En ambos casos, se deberá respetar un determinado formato que permita a los métodos de *Fichero.java* decodificar correctamente la información contenida en ellos.

Formato de los ficheros de opciones de menú

Las opciones, textos y títulos de las pantallas y menús se almacenan en tres ficheros, uno por cada idioma disponible en la aplicación. La tabla 4.8 recoge el nombre de estos ficheros.

Fichero	Descripción
Mensajes_ESP.log	Opciones, textos y títulos en español
Messages_ENG.log	Opciones, textos y títulos en inglés
Messaggi_ITA.log	Opciones, textos y títulos en italiano

Tabla 4.8. Identificadores de los ficheros de opciones

En función de qué idioma esté seleccionado en ese momento, se obtendrá la información de un fichero o de otro. En la figura 4.14 se muestran las primeras líneas del fichero de opciones de menú en español.

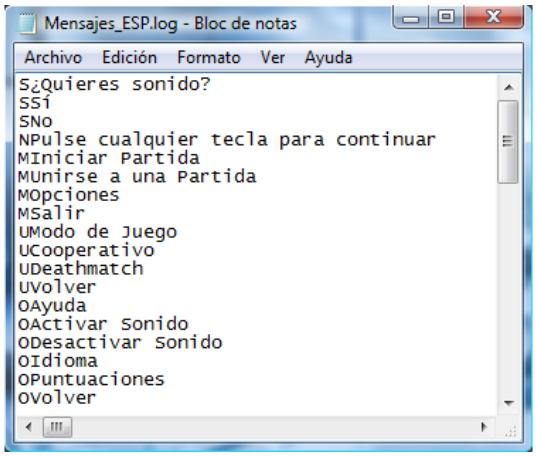


Figura 4.14. Fichero de opciones de menú en español

Cada línea comienza con una letra en mayúsculas. Esta letra es el identificador de pantalla definido en la interfaz gráfica de usuario, como ya se ha comentado en la

sección correspondiente a la interfaz gráfica de la aplicación. Por ejemplo, la '*S*' de las primeras líneas identifica a la pantalla donde se pregunta si se desea activar el sonido al iniciar la aplicación (*SonidoCanvas.java*); la '*N*' hace lo propio para la pantalla de presentación del videojuego (*PantallaInicialCanvas.java*), y así sucesivamente… A continuación del identificador de pantalla está el dato (la opción, mensaje, título o texto que se desee representar por pantalla) propiamente dicho.

Con la información así dispuesta en el fichero, el método *parsearDato* se invocará desde *GUICanvas.java* (concretamente, desde *leerTextos*) para ir leyendo línea por línea hasta encontrar el identificador de la pantalla para la que se quiere obtener los datos. Una vez haya encontrado el primer dato de la pantalla en cuestión, si hubiera más, estarían a continuación, por lo que ya estarían también localizados.

Formato de los ficheros de los mapas

Los mapas vienen definidos en ficheros de texto, denominados como *mapa1.txt*, *mapa2.txt*… de la misma manera para los 10 mapas que componen el videojuego. Todos ellos respetan un mismo formato de almacenamiento de la información. En la figura 4.15 se pueden observar las primeras líneas del fichero del primer mapa (*mapa1.txt*).

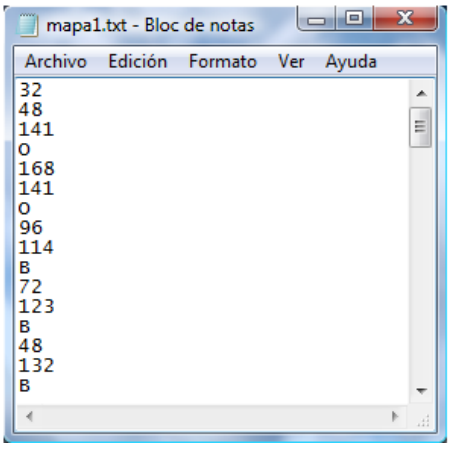


Figura 4.15. Fichero del primer mapa

El primer dato en todos los ficheros de mapas es siempre el número de ladrillos que componen el mismo, en este caso, 32. A continuación, en las sucesivas líneas tienen que describirse las coordenadas en los dos ejes y el tipo de ladrillo que aparecerá en esas

coordenadas, para los 32 ladrillos presentes en el mapa. Así, en las coordenadas *(48,141)* habrá un ladrillo de color oro –un ladrillo irrompible–, en *(168,141)* otro, en *(96,114)* habrá un ladrillo blanco, etc.

De esta manera, cuando el método *parsearDato* se invoque desde la clase *GestorDeMapas.java* (concretamente desde *cargarMapa*), la primera llamada se hará para obtener el número de ladrillos, que servirá para poder generar un mapa indicando cuántos ladrillos lo van a componer, y definir un bucle para ir añadiendo ladrillos (que se leerán según el formato descrito) que se repetirá tantas veces como ladrillos disponga el mapa.

4.4.3.2. RMS

El sistema de gestión de registros o RMS, permite almacenar información entre cada ejecución del *MIDlet* de la aplicación. Como ya se ha comentado al inicio de la presente sección, el RMS está implementado en una base de datos basada en registros. En la figura 4.16 se ilustra la estructura de un *Record Store*.

Record Store	
RecordID	**Datos**
1	byte[] arrayDatos
2	byte[] arrayDatos
…	…

Figura 4.16. Estructura de un Record Store

Cada *Record Store* está compuesto por cero o más registros. Un nombre de *Record Store* es sensible a mayúsculas y minúsculas y está formado por un máximo de 32 caracteres UNICODE.

Cada uno de estos registros está formado por dos unidades:

- Un número identificador de registro (*Record* ID) que es un valor entero que realiza la función de clave primaria en la base de datos.
- Un *array* de bytes que es utilizado para almacenar la información deseada.

Para administrar correctamente la información que se lee y se almacena mediante RMS, se ha implementado la clase *GestorRMS.java* en el paquete *util*, al igual que *Fichero.java*, y cuyo diagrama es el que se muestra en la figura 4.17.

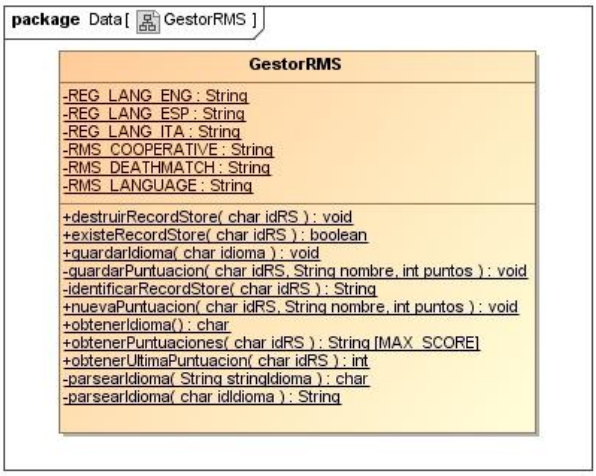


Figura 4.17. Diagrama de la clase GestorRMS.java

Como puede observarse, se define una serie de atributos constantes de uso privado, cuyo significado y utilidad pueden analizarse mejor si se dividen en dos grupos:

- Identificadores de los *Record Store*: son las cadenas de caracteres que dan nombre a los diferentes *Record Store* utilizados en la aplicación. Se resumen en la tabla 4.9.

Identificador	Descripción
RMS_COOPERATIVE	*Record Store* que almacena las puntuaciones del modo cooperativo
RMS_DEATHMATCH	*Record Store* que almacena las puntuaciones del modo *deathmatch*
RMS_LANGUAGE	*Record Store* que almacena el idioma seleccionado

Tabla 4.9. Identificadores de los Record Store

Desde el programa principal, los dos modos de juego vienen definidos en *Juego.java* por un carácter, al igual que el identificador que hace referencia al idioma, definido en *Constantes.java*, como se verá más adelante, por lo que será necesario una conversión en el tipo de datos (o *parse*, en inglés) para adaptar la información de un carácter a una cadena de caracteres. Esto se realiza mediante el método *identificarRecordStore(char idRS)*, el cual proporciona la

correspondiente cadena de caracteres que identifica al *Record Store* que se desea obtener según el carácter de entrada que se pasa por parámetro.

- Identificadores de los idiomas: cada idioma disponible en la aplicación tiene su correspondiente identificador interno en el gestor de registros RMS. Estos identificadores se detallan en la tabla 4.10.

Identificador	Descripción
REG_LANG_ENG	Identificador para el registro del idioma inglés
REG_LANG_ESP	Identificador para el registro del idioma español
REG_LANG_ITA	Identificador para el registro del idioma italiano

Tabla 4.10. Identificadores de los idiomas en RMS

Al igual que antes, los idiomas están definidos en el programa a partir de caracteres (definidos en la clase *Constantes.java*), en lugar de cadenas de caracteres, por lo que nuevamente será necesario un proceso de conversión entre tipos de datos. Además, en este caso, dicha conversión necesitará hacerse en los dos sentidos (es decir, de carácter a cadena de caracteres y viceversa), pues se necesitará tanto recuperar el idioma previamente seleccionado por el usuario cada vez que se inicie una nueva sesión de la aplicación, como guardar el nuevo idioma seleccionado cuando éste haya sido modificado. Para realizar ambos procesos, se utilizarán los métodos *parsearIdioma(String stringIdioma)* y *parsearIdioma(char idIdioma)*, respectivamente, los cuales proporcionarán la información en el tipo de datos adecuado para cada caso (un carácter en el primero y una cadena de caracteres en el segundo).

Al iniciar el *MIDlet* de la aplicación, éste debe recuperar el idioma actual con el que está configurado el programa (en la primera ejecución, se carga el idioma por defecto: el español). Para ello, hará uso del método *obtenerIdioma()*. Naturalmente, éste podrá ser modificado más adelante, desde el correspondiente menú para cambiar el idioma del programa. Esto se hará a través del método *guardarIdioma(char idioma)*, una vez se haya seleccionado el nuevo idioma.

Por otra parte, cuando se trata de guardar una nueva puntuación, se recurre al método *nuevaPuntuacion(char idRS, String nombre, int puntos)*. Este método se invoca

cada vez que se obtiene un nuevo record, pero también en la primera ejecución de la aplicación, para cargar las puntuaciones por defecto cuando aún no hay registrada ninguna puntuación. En el primer parámetro se indica el modo de juego que se trata (cooperativo o *deathmatch*), mientras que los otros dos aportan el nombre de quién, o quiénes han alcanzado la nueva marca y los correspondientes puntos. En el caso de cargar las puntuaciones por defecto, éstas se obtienen desde la clase *Constantes.java*, donde están definidos los nombres y las puntuaciones por defecto. Este método se encargará de introducir la nueva puntuación obtenida de manera que no altere el orden descendente por puntuación en el que estaban ordenadas las que ya había antes, si ya hubiera tres puntuaciones registradas, y descartar, por tanto, el registro con la puntuación más baja, o simplemente agregarla al *Record Store* si no estuvieran registradas todavía tres puntuaciones (por iniciar a cargar las puntuaciones por defecto). En cualquier caso, tras la eventual ordenación de los registros, se procede a su almacenamiento, con ayuda del método privado *guardarPuntuacion(char idRS, String nombre, int puntos)*. Posteriormente, cuando se requiera leer de RMS las puntuaciones obtenidas para mostrarlas por pantalla desde el correspondiente menú de puntuaciones, se invocará el método *obtenerPuntuaciones(char idRS)*, el cual proporcionará un *array* de cadenas de caracteres con todas las puntuaciones registradas. Finalmente, para comprobar, al final de la partida, si la puntuación obtenida debe ser almacenada en RMS como un nuevo record, existe el método *obtenerUltimaPuntuacion(char idRS)*, que proporciona el registro de la puntuación más baja almacenada en el *Record Store* de ese modo de juego (en caso de ser superior, se tratará de un nuevo record).

Por último, hay un par de métodos de utilidad general para todos los *Record Store* de la aplicación: *existeRecordStore(char idRS)*, que comprueba si existe un determinado *Record Store*; y *destruirRecordStore(char idRS)*, que borra un *Record Store* completo.

4.4.4. GESTOR DE SONIDOS

Como ya se comentó en el anterior capítulo, la aplicación dispone de efectos de sonido y pequeñas melodías que sonarán en el terminal si se tiene habilitado el sonido. Para implementarlo, se ha desarrollado una clase específica, dedicada a la gestión y tratamiento de los sonidos que se incluyen en el programa. Esta clase es *GestorDeSonidos.java* y su diagrama se expone en la figura 4.18.

Figura 4.18. Diagrama de la clase GestorDeSonidos.java

En esta clase se detallan todos los sonidos y los directorios donde se ubican los archivos que contienen dichos sonidos, así como los atributos encargados de ejecutarlos, como se analizará a continuación:

- Identificadores de los sonidos: consisten en números enteros que representan el índice del reproductor de esos sonidos dentro del atributo que contiene el *array* de reproductores (o *players*, en inglés) de la clase, **arrayDeSonidos**. En la tabla 4.11 se elabora una lista que contiene los identificadores de los diferentes sonidos.

Identificador	Momento en el que es emitido el sonido
EXTRALIFE	Recogida del *power-up EXTRALIFE*
GAMEOVER	Fin de partida
HOLDBALL	Captura de la pelota por parte de la nave con *HOLDBALL* activado
INICIOPARTIDA	Inicio de una nueva partida en ambos terminales
INICIOVIDA	Inicio de una nueva vida
INTRO	Inicio de la aplicación (melodía de presentación del videojuego)
LADRILLO	Colisión de una pelota con un ladrillo rompible
LADRILLO_IRROMPIBLE	Colisión de una pelota con un ladrillo irrompible
NAVE	Rebote de una pelota con una nave
REBOTE	Colisión entre dos pelotas
SHIPGROW	Recogida del *power-up SHIPGROWUP* / *SHIPGROWDOWN*
SHOT	Lanzamiento de dos proyectiles (*SHOT* activado)
VIDAMENOS	Pérdida de una vida

Tabla 4.11. Identificadores de los sonidos implementados

Para el videojuego desarrollado en el presente videojuego, se han incluido un total de trece sonidos, según viene indicado en la constante *TOTAL_SONIDOS*.

- Identificadores de los directorios de los sonidos: asimismo, cada sonido encuentra su fichero de audio en el *path* proporcionado por el identificador correspondiente. Hay tantos *paths* como sonidos en la aplicación. La lista de identificadores de directorios se recoge en la tabla 4.12.

Path	Descripción
PATH_SONIDO_EXTRALIFE	Archivo de sonido del *power-up* vida extra
PATH_SONIDO_GAMEOVER	Archivo de sonido del final de una partida
PATH_SONIDO_HOLDBALL	Archivo de sonido del *power-up* que atrapa la pelota
PATH_SONIDO_INICIOPARTIDA	Archivo de sonido del inicio de una partida
PATH_SONIDO_INICIOVIDA	Archivo de sonido del inicio de una vida
PATH_SONIDO_INTRO	Archivo de sonido de la melodía de presentación
PATH_SONIDO_LADRILLO	Archivo de sonido de la colisión con un ladrillo rompible
PATH_SONIDO_LADRILLO_IRROMPIBLE	Archivo de sonido de la colisión con un ladrillo irrompible
PATH_SONIDO_NAVE	Archivo de sonido del rebote tras impactar con una nave
PATH_SONIDO_REBOTE	Archivo de sonido de la colisión entre dos pelotas
PATH_SONIDO_SHIPGROW	Archivo de sonido de los *power-ups* que aumentan / disminuyen la anchura de la nave
PATH_SONIDO_SHOT	Archivo de sonido del *power-up* de disparo
PATH_SONIDO_VIDAMENOS	Archivo de sonido de la pérdida de una vida

Tabla 4.12. Identificadores de los directorios de los ficheros de sonido

Una vez presentadas las constantes definidas en la clase, se pasará a analizar los dos atributos que controlarán el procesado y la autorización para que los sonidos puedan ser emitidos:

- **sonidoOn**: este atributo informa sobre si el sonido de la aplicación está habilitado (*true*) o no (*false*). Puede consultarse y modificarse a través de los métodos *getSonido()* y *setSonido(boolean sonidoOn)* respectivamente.

- **arrayDeSonidos**: es un *array* de tantas posiciones como indica *TOTAL_SONIDOS*, en el que se almacenan los reproductores asociados a cada uno de los sonidos del videojuego.

Los reproductores son instancias de la clase *Player*, encargada de controlar la interpretación y ejecución de los archivos de audio. La figura 4.19 muestra el diagrama de estados de esta clase.

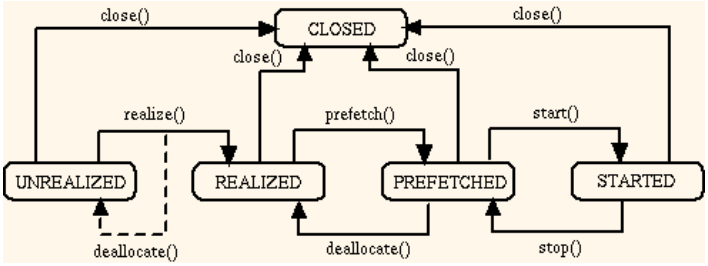


Figura 4.19. Diagrama de estados de la clase Player

Inicialmente, cada objeto *Player* parte del estado *UNREALIZED*. En este estado, el reproductor no dispone de información suficiente para localizar los recursos necesarios para comenzar la reproducción. Una vez haya localizado los datos, pasará al estado *REALIZED*. En el código que se ha implementado, esta transición se produce al crear el reproductor a partir de un flujo de datos de entrada procedente del fichero de audio y del formato de ese fichero de audio, mediante la instrucción *Manager.createPlayer(InputStream is, String formato)*. La tabla 4.13 resume los tipos de ficheros de audio soportados en J2ME, junto con la cadena de caracteres que hay que utilizar como identificador en el parámetro *formato* para referirse a ellos.

Tipo de fichero	Identificador a utilizar
Ficheros “wave audio”	audio/x-wav
Ficheros AU	audio/basic
Ficheros MP3	audio/mpeg
Ficheros MIDI	audio/midi
Secuencias de tonos simples	audio/x-tone-seq

Tabla 4.13. Ficheros de audio soportados en J2ME

El siguiente estado por el que pasaría es el de *PREFETCHED*. Se llega a él cuando el reproductor ha recibido suficiente cantidad de datos para comenzar con la reproducción. La utilidad de definir este estado radica en que llegan a él los reproductores que están listos para empezar con la reproducción de datos. De esta manera se reduce el tiempo de latencia desde que se indica que se inicie la reproducción y el instante en que realmente comienza. En este punto, cuando haya dado comienzo la reproducción se llegaría al estado *STARTED*. Además de estos cuatro estados, existe un

quinto, denominado *CLOSED*, al cual el reproductor podrá llegar desde cualquiera de los anteriores estados cuando se indique que no va a ser utilizado nunca más. En este caso, el reproductor liberará los recursos que tenía reservados.

Cada reproductor se inicializa desde el propio constructor de la clase, mediante una llamada al método *cargarSonido(String path, String formato, int indiceSonido)*. En este momento es cuando se les proporciona la información necesaria para localizar los recursos que les permitan iniciar la reproducción de los sonidos, trasladándose, por tanto, del estado *UNREALIZED* a *PREFETCHED*, pasando por *REALIZED* como estado intermedio entre la creación del reproductor y la llamada al método *prefetch()*. Una vez se hayan inicializado todos los reproductores, éstos oscilarán entre los estados *PREFETCHED* y *STARTED* mientras dure la ejecución del programa. La función de los métodos *pararSonido(int indiceSonido)* y *reproducirSonido(int indiceSonido)* será precisamente la de mover a los reproductores a estos estados, respectivamente, previa verificación de que se trata de un índice de sonido válido y que la opción de sonido está efectivamente habilitada en el terminal. Además, el método *sonidoEnCurso(int indiceSonido)* permite saber si un determinado sonido se encuentra en ejecución (es decir, se encuentra en el estado *STARTED*). Finalmente, cada reproductor se moverá al estado *CLOSED* una vez haya finalizado la ejecución de la aplicación en el terminal.

4.4.5. CONSTANTES

Existe una serie de atributos constantes, de carácter general, de utilidad para el conjunto del código implementado para la aplicación, que se definen en una clase aparte. Esta clase se denomina *Constantes.java*, y su diagrama es el que se muestra a continuación, en la figura 4.20.

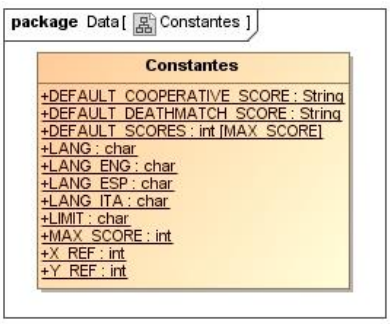


Figura 4.20. Diagrama de la clase Constantes.java

Únicamente consta de atributos, no contiene ningún método. Estos atributos son constantes con modificador de acceso público que, según su significado, pueden agruparse y analizarse de la siguiente manera:

- Identificadores de las puntuaciones por defecto: en esta clase se definen los identificadores que van a contener los nombres y las puntuaciones que aparecerán inicialmente en la aplicación, cuando aun no se ha registrado ningún record alcanzado por los jugadores. La tabla 4.14 resume cuáles son estos identificadores, a los que se añade una breve descripción de cada uno de ellos.

Identificador	Descripción
DEFAULT_COOPERATIVE_SCORE	*String* que contiene el nombre de los usuarios para el modo cooperativo
DEFAULT_DEATHMATCH_SCORE	*String* que contiene el nombre de usuario para el modo *deathmatch*
DEFAULT_SCORES	Puntuaciones por defecto (son iguales para ambos modos de juego)

Tabla 4.14. Identificadores de las puntuaciones por defecto

Los dos primeros identificadores establecen los *friendly names* que aparecerán inicialmente en pantalla desde los respectivos menús de visualización de puntuaciones, y el tercero es un *array* de enteros donde se almacenan las puntuaciones iniciales asociadas a ellos. El número de posiciones de este *array* vendrá determinado por la constante *MAX_SCORE*, también definida en esta clase. Esta constante determinará el número máximo de puntuaciones que se almacenarán en la memoria del terminal, concretamente tres por cada modo de juego.

- Identificadores de idioma: tanto el identificador para referirse al *Record Store* donde se almacena el idioma seleccionado como los identificadores de cada idioma implementado en la aplicación se definen en esta clase. Estos identificadores se detallan en la tabla 4.15.

Identificador	Descripción
LANG	Identificador para el *Record Store* que almacena el idioma seleccionado
LANG_ENG	Identificador del idioma inglés
LANG_ESP	Identificador del idioma español
LANG_ITA	Identificador del idioma italiano

Tabla 4.15. Identificadores de idiomas

- *X_REF*, *Y_REF*: estas constantes indican las dimensiones de la pantalla sobre las que se trabajó para definir las coordenadas de los ladrillos de cada mapa. Coinciden con las dimensiones del emulador de Sun (240 píxeles de ancho por 291 de alto) y sirven como valores de referencia a la hora de calcular cuánto hay que desplazar cada ladrillo para que se representen centrados en las pantallas de los terminales, con independencia de cuáles sean sus dimensiones.

- *LIMIT*: esta constante sirve de separador entre diferentes campos de un mismo mensaje o también de indicador de ausencia de identificador en determinados campos.

4.5. BLUETOOTH

Como ya se comentó en el segundo capítulo, Bluetooth es una tecnología de radio de corto alcance (aproximadamente unos 10 metros en un ambiente libre de obstáculos), que permite conectividad inalámbrica entre dispositivos remotos y que opera en la banda libre de radio ISM (banda industrial científico-médica) a 2.4 GHz. También se señaló que Java permite el desarrollo de aplicaciones que utilicen Bluetooth mediante la especificación JSR 82, que posibilita la abstracción de los detalles de bajo nivel para centrarse únicamente en las capacidades que ofrece (registro de servicios, descubrimiento de dispositivos y servicios, establecimiento de la conexión, etc). En particular, el estándar JSR 82 define un API Java para Bluetooth que depende del paquete CLDC *javax.microedition.io*: es el paquete *javax.bluetooth*.

4.5.1. ELECCIÓN DEL PROTOCOLO DE COMUNICACIÓN

El primer aspecto importante a la hora de diseñar la funcionalidad multijugador es fijar el protocolo de comunicación sobre el que se va a sustentar la estructura. El enfoque seguido será de una estructura de cliente-servidor sobre Bluetooth –limitando el número de clientes a uno solo, es decir, el modo multijugador solo tendría dos jugadores–. Las tres opciones de protocolo de comunicación son: RFCOMM, L2CAP, y OBEX.

RFCOMM (*Radio Frequency Communication*) es también conocido como perfil de puerto serie (*Serial Port Profile*, o SPP). Este protocolo emula múltiples conexiones entre puertos serie RS-232. El protocolo establece las sesiones de comunicación utilizando las direcciones Bluetooth de los dos puntos terminales. Una sesión puede tener más de una conexión –el número de conexiones depende de la implementación–. Las sesiones, por otro lado, están ligadas a cada par único de direcciones Bluetooth de los dispositivos que se conectan. Además, un dispositivo podrá tener más de una sesión, entendiéndose por esto que podrá estar conectado a más de un dispositivo Bluetooth, es decir, que RFCOMM no queda limitado a una única sesión.

L2CAP (*Logical Link Control and Adaptation Protocol*) se trata de un protocolo que posee dos tipos de comunicaciones: una orientada a conexión (bidireccional), y otra no orientada a conexión (unidireccional). El API de JSR 82 no soporta para este protocolo el tipo de comunicación no orientada a conexión. L2CAP requiere de la configuración de una serie de parámetros fundamentales del canal de comunicación que se habrán de negociar entre los dispositivos Bluetooth. Estos parámetros son:

- La Unidad Máxima de Transferencia (MTU): valor del *payload* máximo que el dispositivo que envía la petición puede atender. Por defecto se tiene 672 *bytes*.
- Tiempo de descarte: cantidad de tiempo durante el cual el administrador del canal intenta transmitir satisfactoriamente el paquete antes de descartarlo. Puede establecerse que se retransmita el paquete continuamente hasta que se reciba confirmación de su llegada o el enlace caiga.
- Calidad de Servicio (*QoS*): opción que describe el flujo de tráfico. No está soportada por la API.

OBEX (*OBject EXchange*), por su parte, es también conocido como protocolo de intercambio de objetos. OBEX no se encuentra definido en la API JSR 82 de Bluetooth, sino que posee su propia API, *javax.obex*. OBEX es un protocolo diseñado por *IrDa* (*Infrared Data Association*) para intercambiar objetos entre clientes y servidores mediante el establecimiento de sesiones OBEX. Para J2ME se optó por extender la API de OBEX para dar cobertura a Bluetooth. OBEX implementa la transferencia de objetos estableciendo una sesión, mediante una petición CONNECT. Ésta termina mediante una

petición DISCONNECT. Entre estas dos peticiones, el cliente puede traer objetos del servidor mediante GET, o enviarlos mediante PUT. Los objetos pueden ser archivos, *vCards*, *arrays* de *bytes*, etc.

En un principio los tres protocolos de comunicación podrían utilizarse para realizar la comunicación Bluetooth. No obstante, hay que destacar que en el caso de OBEX su uso supone la creación virtual de un cliente y un servidor OBEX en los dos dispositivos móviles con intención de comunicarse, puesto que la comunicación ha de ser bidireccional y en OBEX es el cliente el que lleva (PUT) o trae (GET) los objetos desde el servidor. Por otro lado, para L2CAP el hecho de que se tengan que configurar parámetros de Unidad Máxima de Transferencia de datos tanto en recepción como en transmisión y tiempos de descarte introduce aspectos de diseño a tener en cuenta a la hora de definir los mensajes a transmitirse por la aplicación y que puede considerarse en cierta forma excesivo para la naturaleza de los mensajes que se espera enviar entre los dos dispositivos.

De los anteriores motivos se desprende que una solución satisfactoria al problema de la comunicación Bluetooth se puede conseguir mediante el empleo del protocolo RFCOMM. Así, de los tres protocolos de comunicación se elige la opción de RFCOMM debido a su mayor versatilidad y sencillez en comparación con las otras dos alternativas.

4.5.2. CONSIDERACIONES ACERCA DEL PROTOCOLO SELECCIONADO

Una aplicación que ofrezca un servicio basado en el perfil de puerto serie (SPP) –protocolo RFCOMM– es un servidor SPP. Una aplicación que inicie una conexión a un servicio SPP es un cliente SPP. Cliente y servidor residen en los extremos de una sesión RFCOMM. El servidor SPP registra su servicio en el SDDB (*Service Discovery DataBase*), y como parte del proceso de registro, se añade un identificador de canal (*channel identifier*) al *ServiceRecord* por la implementación.

Establecer una conexión satisfactoria consta de una serie de pasos: inicialización de la pila Bluetooth, descubrimiento de dispositivos y servicios, manejo del dispositivo y comunicación. Al final del proceso lo que queda es un flujo o *stream* de datos. Para el caso de la aplicación desarrollada en el presente proyecto se tendrán dos flujos –

identificados como las instancias de las clases *DataInputStream* y *DataOutputStream*, como se comentará más adelante–, uno por cada sentido, en el cliente y en el servidor.

Para la inicialización de la aplicación Bluetooth hay que tener en cuenta el BCC (*Bluetooth Control Center*). Los dispositivos Bluetooth que implementen la JSR 82 pueden permitir que múltiples aplicaciones se estén ejecutando concurrentemente. El BCC previene que una aplicación pueda perjudicar a otra. El BCC puede ser una aplicación nativa, una aplicación en un API separado, o sencillamente un grupo de parámetros fijados por el proveedor que no pueden ser cambiados por el usuario.

En la fase de descubrimiento, los dispositivos Bluetooth localizan a otros dispositivos con los que poder realizar la comunicación. Los dispositivos inalámbricos son móviles, motivo por el cual necesitan un mecanismo que permita encontrar, conectar y obtener información sobre las características de otros dispositivos con los que conectarse. Será necesario, pues, localizar dispositivos que ofrezcan un servicio de puerto serie.

4.5.3. IMPLEMENTACIÓN DE LA FUNCIONALIDAD BLUETOOTH

En la figura 4.21 se muestra el diagrama de clases de las clases que implementan el cliente y el servidor de esta aplicación. El extremo que hará de maestro en la comunicación será el que registre el servicio SPP, por lo que a todos los efectos será el extremo servidor. A su vez, el terminal esclavo será el que busque ese servicio, por lo que será el encargado de lanzar la petición de inicio de conexión, convirtiéndose en el cliente de la comunicación. En este sentido, será necesario que el esclavo tenga implementado un *listener* para poder rastrear y localizar los posibles dispositivos remotos que dispongan de un servicio SPP registrado. Dicha implementación consistirá en incluir una clase que desarrolle los métodos de la interfaz *DiscoveryListener*. Estos métodos se encargarán de notificar los dispositivos remotos localizados y los servicios encontrados en ellos, una vez se haya ordenado la correspondiente búsqueda.

Puede apreciarse que ambas clases, maestro y servidor, heredan de una misma clase, *Extremo.java*, ya que ésta contiene los métodos y atributos necesarios para cada extremo de la comunicación, independientemente de si se trata del terminal maestro o del esclavo.

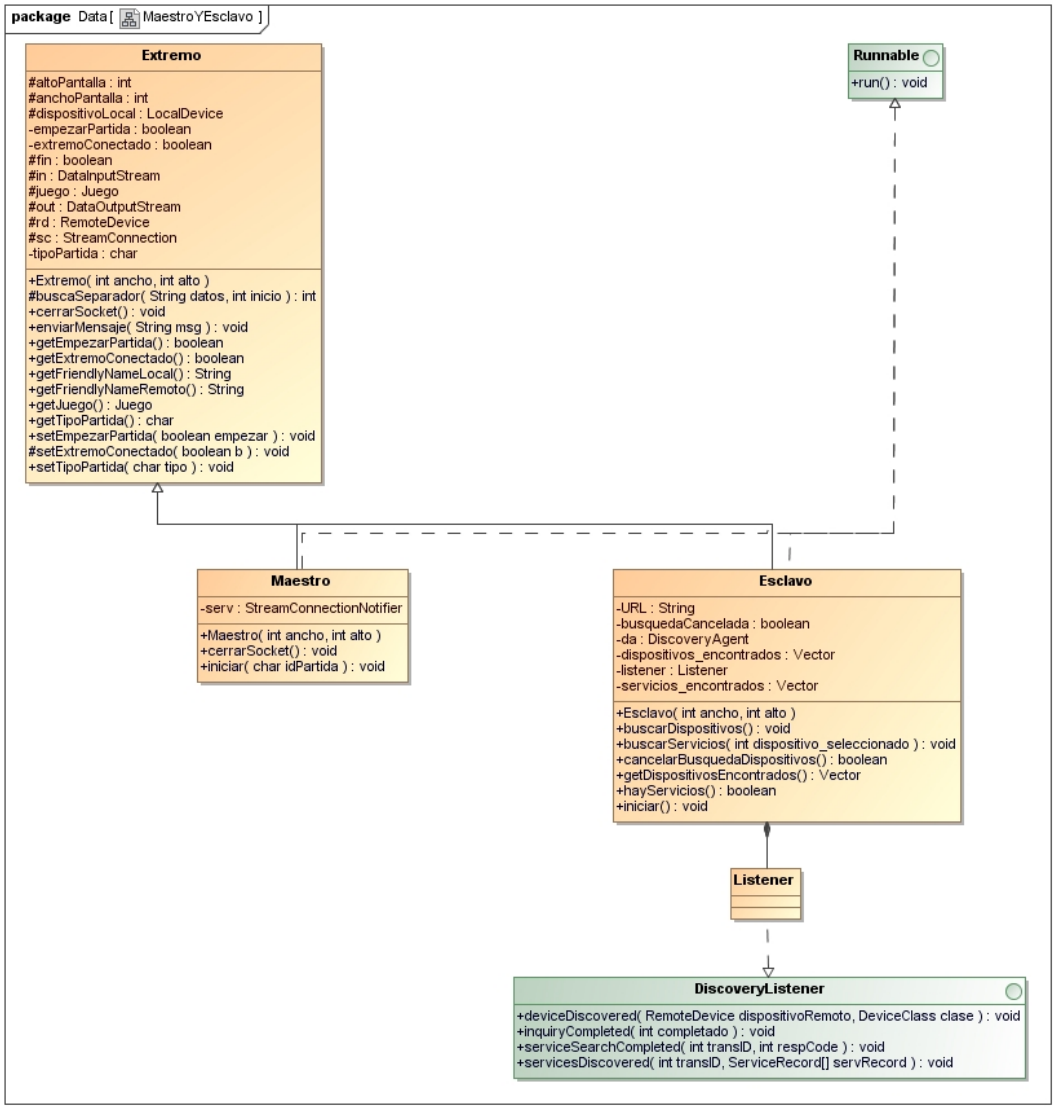


Figura 4.21. Diagrama de clases de Maestro.java y Esclavo.java

Esta clase define los siguientes atributos:

- **anchoPantalla**, **altoPantalla**: definirán las dimensiones que tendrá la pantalla sobre la que se va a desarrollar la partida. En particular, para permitir que el videojuego se visualice correctamente en los dos dispositivos, cada dimensión tomará el valor más pequeño de las pantallas de los dos terminales involucrados en la comunicación. Ambos valores se pasarán por parámetro al constructor de la clase principal del videojuego, *Juego.java*.

- **tipoPartida**: contiene el modo de juego seleccionado por el terminal maestro al crear la partida. Este valor, sin embargo, deberá estar presente en ambos

extremos de la comunicación, pues se requerirá para cuando se precise indicar el modo de juego al inicio de cada partida. Los métodos que permiten acceder a este atributo son *getTipoPartida()* y *setTipoPartida(char tipo)*.

- **juego**: cada extremo de la comunicación debe tener su propia instancia de la clase *Juego.java*, con la que iniciará, desarrollará y terminará una o más partidas. Cuando se precise acceder a ella desde otra clase, se utilizará el método *getJuego()*.

- **in**, **out**: son los flujos de datos sobre los que irán los mensajes enviados y recibidos en cada extremo. Así pues, sobre estos flujos irá la información que deben intercambiar el cliente y el servidor para que el modo multijugador funcione. En particular, el envío de datos se realizará a través de *enviarMensaje(String msg)*, el cual pondrá sobre el flujo de salida (el objeto **out**) la información que se desea enviar al otro terminal, proporcionada en *msg*. En el otro extremo, cuando se recibe el mensaje, se almacena en el flujo de entrada **in**, para su posterior lectura y decodificación, que se hará con ayuda del método *buscaSeparador(String datos, int inicio)*, para localizar los separadores que distinguen los diferentes campos que integran el mensaje.

- Atributos de control de flujo: sirven para controlar el flujo de control de inicio y fin de cada partida. Estos atributos se recogen en la tabla 4.16.

Identificador	Descripción
empezarPartida	Habilita al terminal para iniciar una nueva partida
extremoConectado	Informa acerca de la conexión de un extremo al enlace Bluetooth
fin	Indica el término de la recepción de mensajes

Tabla 4.16. Atributos de control de flujo de cada partida

El atributo **fin** es el booleano que controla la condición del bucle de recepción de mensajes. Se desactivará con el cierre del *socket* del extremo, mediante el método *cerrarSocket()*. Inicialmente, los dispositivos no se encuentran conectados a través del enlace Bluetooth, por lo que **extremoConectado** no está activo. Sin embargo, cuando se recibe el primer mensaje del otro terminal, se entiende que la conexión al enlace es efectiva, por lo que su estado pasa a *true*.

Posteriormente, esta condición servirá tanto a un extremo como a otro para saber si es preciso registrar un servicio de puerto serie o iniciar un proceso de búsqueda de dispositivos, ambos procesos necesarios solo en el establecimiento de la conexión, previo al inicio de la primera partida. El atributo **empezarPartida**, por su parte, establece la disponibilidad del terminal para iniciar una partida en un momento dado. La figura 4.22 representa de manera esquemática el diagrama de estados del videojuego en sus diferentes etapas para el inicio y el final de una partida en ambos extremos de la comunicación y lo relaciona con los valores de **extremoConectado** y **empezarPartida** para cada una de estas etapas.

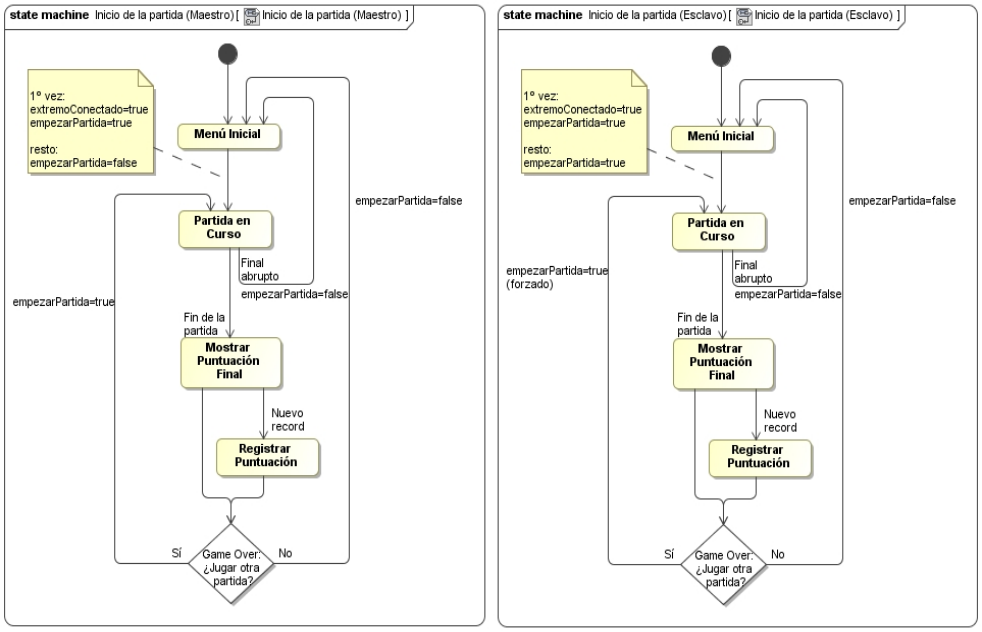


Figura 4.22. Diagramas de estados del inicio y fin de las partidas
(a) terminal maestro, (b) terminal esclavo

En estos diagramas puede observarse cómo se activa **extremoConectado** en ambos terminales la primera vez que se accede a una partida del videojuego. El valor que toma **empezarPartida** depende, sin embargo, de cada terminal: por un lado, el terminal maestro fuerza el inicio de la partida la primera vez y siempre que decida iniciar una nueva partida habiendo previamente terminado la anterior con normalidad. En el resto de casos, después de indicar el tipo de partida esperará a que el terminal esclavo decida unirse a ella (de otra manera, iniciaría la partida sin haberle dado opción al otro terminal de unirse a él); el terminal

esclavo, por su parte, siempre que inicia una nueva partida lo hace con **empezarPartida** activo. Cuando acaba una partida de forma abrupta o se termina con normalidad, pero no se desea iniciar una nueva en ese momento, se deja a *false*, esperando a que el terminal maestro cree una nueva partida, indicando el correspondiente modo de juego. Cuando sepa qué tipo de partida es (recibiendo el mensaje de inicio de partida), activará **empezarPartida**, indicando que cuando quiera unirse a ella estará en condiciones de hacerlo.

Puede accederse al contenido de cada uno de estos dos atributos, a través de los correspondientes métodos *get* y *set*, en particular: *getEmpezarPartida()*, *setEmpezarPartida(boolean empezar)*, *getExtremoConectado()* y *setExtremoConectado(boolean b)*.

- Otros atributos generales: finalmente, existe un pequeño grupo de atributos comunes a ambos extremos, que son necesarios para establecer la comunicación entre ellos. Se resumen en la tabla 4.17.

Identificador	Descripción
dispositivoLocal	Objeto de la clase *LocalDevice* que permite reconocer al dispositivo local
rd	Objeto de la clase *RemoteDevice* que permite reconocer al dispositivo remoto
sc	Objeto de la clase *StreamConnection* que permite establecer la conexión Bluetooth

Tabla 4.17. Otros atributos definidos en Extremo.java

Los métodos *getFriendlyNameLocal()* y *getFriendlyNameRemoto()* permitirán recuperar los *friendly names* de los dispositivos, a partir de **dispositivoLocal** y **rd**, respectivamente. Serán de utilidad a la hora de representar y, si procede, almacenar las puntuaciones finales obtenidas por los jugadores.

Posteriormente, cada extremo añade a estos métodos y atributos los que son necesarios para ejercer su correspondiente rol en la comunicación. Así, el terminal maestro necesitará del objeto *serv* de la clase *StreamConnectionNotifier* para recibir la petición de conexión del cliente al servicio que ofrece y reescribir el método *cerrarSocket()*, heredado de *Extremo.java*, para incluir ahí el cierre de este objeto adicional. Además, incluirá desde su método *iniciar(char idPartida)* el modo de juego seleccionado. El esclavo, por su parte, necesitará poner en marcha el mecanismo de búsqueda de dispositivos remotos y de servicios encontrados para un dispositivo

remoto. Esto lo realizará a través de su atributo de la clase *DiscoveryAgent* **da**. Mediante el método *buscarDispositivos()* se inicia el rastreo de dispositivos Bluetooth en el radio de cobertura del terminal, lanzando el *listener* implementado. Dicha búsqueda puede interrumpirse manualmente, en cuyo caso se invocaría al método *cancelarBusquedaDispositivos()*. Cada dispositivo encontrado lo notifica a través del método *deviceDiscovered(RemoteDevice dispositivoRemoto, DeviceClass clase)*. Cuando haya concluido con la búsqueda de dispositivos, el *listener* ejecutará el método *inquiryCompleted(int completado)*. Posteriormente, una vez encontrados los dispositivos, se pueden consultar desde el método *getDispositivosEncontrados()* e iniciar el proceso de búsqueda de servicios de puerto serie en alguno de ellos, con ayuda del método *buscarServicios(int dispositivo_seleccionado)*, indicando el dispositivo de la lista de dispositivos encontrados previamente. Para ello, el *listener* notificará con *servicesDiscovered(int transID, ServiceRecord[] servRecord)* el descubrimiento de un nuevo servicio en ese dispositivo remoto y con *serviceSearchCompleted(int transID, int repsCode)* la finalización de la búsqueda de servicios en dicho dispositivo. Una vez se haya completado la búsqueda, el método *hayServicios()* confirmará si se han encontrado servicios de puerto serie sobre el dispositivo seleccionado. En tal caso, la partida podrá iniciarse finalmente, invocando al método *iniciar()*, que para no acaparar el hilo principal de la ejecución, ejecutará en un hilo aparte la recepción de mensajes, almacenando en **URL** la dirección del dispositivo encontrado al que se desea conectar.

Como ya se explicará con mayor detalle cuando se analicen los mensajes de establecimiento y fin de la conexión, la primera vez que el terminal maestro recibe un mensaje de inicio de partida (mensaje de tipo *START*), debe decodificar los parámetros referentes al ancho y alto de la pantalla del terminal remoto que vienen adjuntos en dicho mensaje. Posteriormente, realiza un breve proceso de elección del tamaño de la pantalla donde va a tener lugar la partida. Para garantizar la correcta visualización en ambos terminales, se tomará el valor mínimo para cada dimensión. Además, crea el objeto de la clase principal con las dimensiones finalmente seleccionadas. Tras esto, lanza la partida, según las dimensiones de la pantalla y el tipo de partida especificados, de la misma manera que la lanzaría si no fuera la primera vez que recibe este tipo de mensaje. Por otra parte, si recibe un mensaje de fin de la partida (mensaje de tipo *STOP*), la partida ha sido interrumpida por abandono del jugador que controla el terminal esclavo, por lo que deberá detener la misma e indicar por pantalla el

correspondiente mensaje de desconexión del otro extremo de la comunicación. Finalmente, si recibe un mensaje de datos (mensaje identificado como *DATA*) solamente deberá delegar en la función *actualizacionRemota*, perteneciente a *Juego.java*, para que ésta se encargue de decodificar y administrar correctamente la información contenida en dicho mensaje.

En el otro extremo de la comunicación, es decir, en el terminal esclavo, la recepción de datos es similar. Solo hay que añadir que, al recibir el mensaje de inicio de partida, necesitará decodificar el tipo de partida que se ha escogido desde el terminal maestro. Además, cabe la posibilidad de que, al iniciar una partida tras haber interrumpido otra con anterioridad o decidido no volver a jugar otra más, el terminal maestro haya creado otra nueva y el jugador que controla el terminal esclavo aún no ha decidido unirse a ella, por lo que, en lugar de iniciarla al instante de recibir el mensaje de tipo *START*, pospone el comienzo de la misma, indicando en **empezarPartida** la disposición de lanzarla cuando decida unirse a ella.

4.5.4. INTERCAMBIO DE MENSAJES BLUETOOTH. SINCRONIZACIÓN.

La necesidad de una arquitectura de cliente-servidor proviene de la sincronía temporal que debe existir en el juego. En lugar de técnicas como el envío masivo de mensajes de sincronización cada cierto intervalo de tiempo, desde el terminal maestro al esclavo, indicando la posición y sentido de movimiento de cada pelota y las coordenadas de las naves, o de una comunicación de igual a igual, en la que ninguno de los dos terminales realiza tareas que no correspondan al otro, se ha optado, con la intención de dotar de mayor jugabilidad al videojuego, por una solución intermedia que se irá comentando a lo largo de la presente sección.

En primer lugar, un modelo de sincronización en el que el terminal maestro abarca todo el peso de simultanear la ejecución de la partida en los dos terminales, ralentizaría el desarrollo de la misma en ambos dispositivos: en el maestro por el exceso de mensajes enviados a una frecuencia relativamente alta para no perder el sincronismo con el otro extremo; y en el esclavo, por una dependencia desmesurada de los mensajes que se reciben desde el terminal maestro, impidiendo el desarrollo normal de la partida en su propio terminal, de acuerdo con sus propios parámetros de movimiento para cada elemento movible. Además, la solución así planteada deja entrever que se producen

muchos envíos de mensajes que son innecesarios, pues una vez se ha determinado para cada pelota un sentido y una cantidad de movimiento en cada eje, los dos dispositivos por separado son perfectamente autosuficientes para continuar con la partida sin perder la coherencia en la ejecución hasta el siguiente evento que se produjera (colisión entre pelotas, con un ladrillo, recogida de un *power-up*…). A esto hay que añadir los mensajes que necesariamente debe enviar el terminal esclavo al maestro cada vez que el jugador que maneja ese dispositivo decida mover la nave hacia un lado u otro, o pulse el botón de disparo para lanzar la pelota o disparar proyectiles si tuviera activado el *power-up* de disparo. Estos movimientos podrían contradecir los mensajes de actualización que le llegan desde el terminal maestro, poniendo en entredicho la jugabilidad del videojuego.

Por otro lado, está la opción contraria, es decir, no dar más inteligencia a un terminal con respecto al otro e implementar una aplicación de igual a igual. Esta otra solución, llevada al extremo también plantea ciertos inconvenientes: en algún momento alguno de los dos terminales debe asumir la capacidad para crear la partida o para indicar al otro dispositivo qué *power-ups* se van a disponer y en qué ladrillos.

Finalmente, se ha optado por una solución intermedia que implemente un modelo ligeramente asimétrico, en el que un terminal asume la responsabilidad de la creación de la partida y la decisión de volver a jugar, seleccionando el modo de juego para cada una de ellas, o la asignación de los *power-ups* sobre los ladrillos en cada mapa y la posterior comunicación al otro extremo. La sincronización durante la partida se realiza en base a eventos que cambian el estado de los elementos de la misma, como una colisión con otra pelota, con un ladrillo, el lanzamiento de una pelota desde una nave o el disparo de proyectiles, etc., asumiendo que una sincronización total entre los dos dispositivos, que permita un ritmo de ejecución simultáneo en ambos en cada instante durante toda la partida, es inviable. En algunos de estos eventos se añade información sobre el resto de elementos de la partida, aun cuando no están involucrados en dicho evento (por ejemplo, se envía información del resto de pelotas activas cuando se produce una colisión de una pelota con un ladrillo) para no perder la coherencia en la ejecución en los dos dispositivos, impidiendo así situaciones de incertidumbre en las que puedan darse unas determinadas circunstancias en un dispositivo y el otro no tenga

noticia de ello, o la partida haya evolucionado de una forma distinta en uno y otro dispositivo.

Como ya se ha adelantado, ambos terminales deberán intercambiar diversos mensajes durante la partida para mantener la sincronización, indicar al otro extremo un movimiento de la nave, determinar el inicio y el final de la partida, etc. Todos los tipos y variantes posibles de los mensajes se definen en una clase ubicada en el paquete *bluetooth*: *Mensaje.java*. En la figura 4.23 se muestra el contenido de esta clase.

Figura 4.23. Diagrama de la clase Mensaje.java

Se puede observar que la clase se compone únicamente de atributos estáticos. Estos atributos determinarán los tipos de mensajes definidos en la aplicación, y que se analizarán con mayor detalle a continuación. El modificador de acceso de todos ellos es *public*, salvo el de *SCREEN*, que es utilizado únicamente por las clases *Maestro.java* y *Esclavo.java*, que pertenecen al mismo paquete. De ahí que su modificador de acceso sea *package*.

Hay dos tipos de mensajes principalmente: los de establecimiento y fin de la conexión Bluetooth, por un lado, y los de datos, por otro.

4.5.4.1. Mensajes de establecimiento y fin de la conexión

Sirven para indicar al otro extremo el inicio y el término de la comunicación Bluetooth entre los dos terminales. Se distinguen dos tipos de mensajes:

- Mensaje *START*: cada vez que se decide iniciar una nueva partida, el terminal maestro envía un mensaje de este tipo para sugerir al esclavo que inicie la partida en su terminal; el esclavo, por su parte, lo envía al maestro cuando se conecta por primera vez al servicio que ofrece éste, y cuando desea unirse a cualquier otra partida creada por el terminal maestro desde la pantalla de menú inicial (si la partida ha terminado de forma natural y se desea jugar otra vez desde la pantalla de *game over*, la partida se inicia automáticamente en los dos terminales sin necesidad de esperar a la confirmación por parte del jugador del terminal maestro). En cualquier caso, este tipo de mensaje servirá para dar a entender al otro extremo la intención de iniciar una nueva partida.

  Además, existe una variante dentro de los mensajes de tipo *START*, que consiste en enviar la información del ancho y el alto de la pantalla del propio terminal, para poder determinar posteriormente el ancho y alto definitivo para la ejecución de la partida. Esto se realiza mediante un identificador complementario para este tipo de mensajes, el identificador *SCREEN*. Esta variante de los mensajes de tipo *START* se emplea al iniciar la primera partida. Cada terminal envía al otro este mensaje para que cada uno pueda establecer, siguiendo el mismo criterio, unas dimensiones de pantalla comunes para ambos. La figura 4.24 representa los dos formatos comentados que puede presentar de este tipo de mensaje.

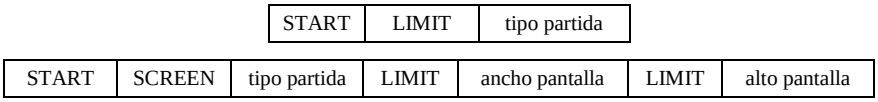


Figura 4.24. Formato de los mensajes START

En el diagrama de secuencia que se muestra en la figura 4.25, se puede observar el esquema de cuáles son los pasos que se han de seguir para establecer una comunicación Bluetooth entre los dos terminales e iniciar la primera partida. Este esquema no es representativo del caso en que se decide

iniciar una partida después de haber terminado otra previamente o de haber abandonado una partida empezada; en ese caso, solo habría mensajes *START* que se envían entre los dos dispositivos y únicamente un mensaje *START* del terminal maestro al esclavo en el caso de que la nueva partida se creara desde la pantalla de *game over*, tras haber terminado con normalidad otra partida previamente.

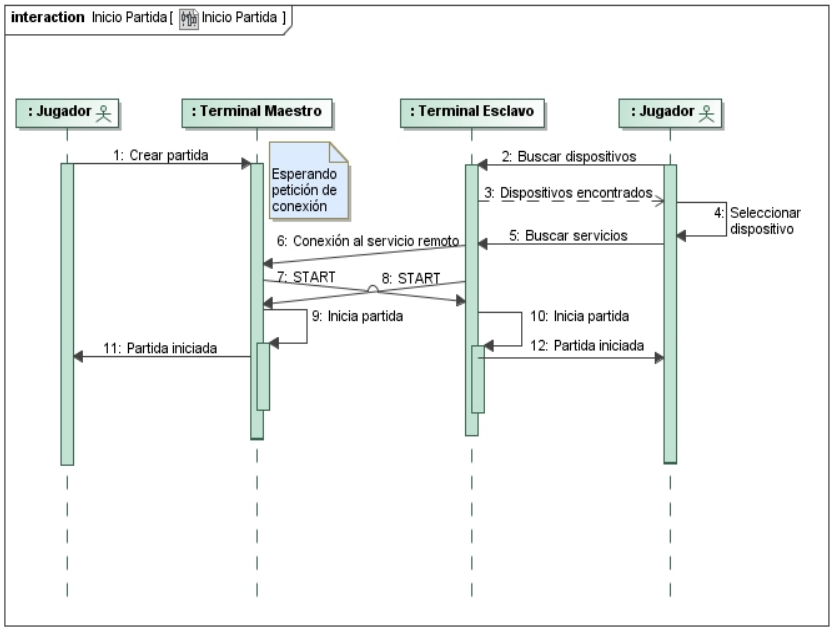


Figura 4.25. Diagrama de secuencia del inicio de una nueva partida

Como es natural, el primer paso es crear la partida. El jugador que decide crearla, hace de su dispositivo el terminal maestro de la comunicación. Tras esto, su terminal se quedará en espera de que un dispositivo remoto se conecte al servicio recién creado, desbloqueando la función *acceptAndOpen()* mencionada con anterioridad.

El otro jugador, por tanto, deberá realizar una búsqueda de dispositivos que ofrezcan un servicio de conexión Bluetooth, por lo que se convierte en el terminal esclavo de la comunicación. Si la búsqueda ha resultado satisfactoria, se mostrará en la pantalla del terminal esclavo la lista de dispositivos encontrados. Tras seleccionar el dispositivo del jugador que ha creado la partida, se busca el servicio registrado en su terminal para conectarse a él (mediante la instrucción *Connector.open*) y desbloquearlo.

Tras esto, ambos estarán ya en disposición de intercambiarse los mensajes *START*, indicando las dimensiones de cada pantalla. Después de la recepción del correspondiente mensaje en cada extremo, se procede al inicio de la partida en uno y otro dispositivo, hecho que el jugador percibirá al visualizar en la pantalla de su terminal el primer nivel, junto con las naves y las pelotas preparadas para iniciar la partida.

- Mensaje *STOP*: maestro y esclavo envían un mensaje de este tipo para salir de una partida ya iniciada, advirtiendo así al extremo remoto de su desconexión de la partida. Además, el terminal maestro también lo envía cuando, no habiéndose desconectado de la partida durante la misma, decide no iniciar una nueva. En cualquier caso, la recepción de este mensaje, genera en ese extremo la aparición del mensaje de desconexión de la partida.

  Este tipo de mensajes se envía sin ningún otro identificador adicional que lo complemente o que extienda su funcionalidad, no hace falta indicar nada más cuando se trata de dar por finalizada una partida. En la figura 4.26 se puede observar el diagrama de secuencia que se corresponde con la finalización de la partida, provocada por el abandono de uno de los jugadores.

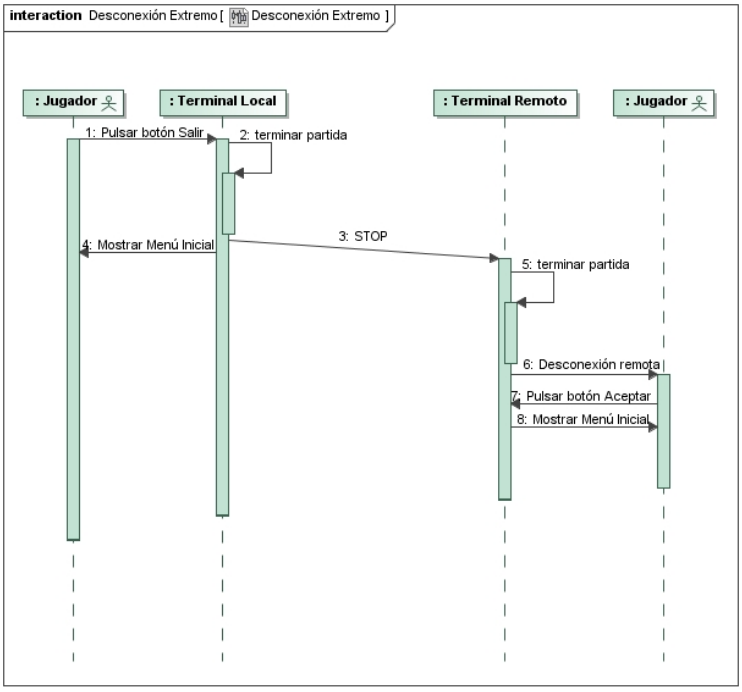


Figura 4.26. Diagrama de secuencia de la desconexión del enlace Bluetooth

El terminal del jugador que decide salir de la partida, la termina y envía el correspondiente mensaje de tipo *STOP* al otro extremo. Tras esto, muestra en la pantalla del dispositivo el menú inicial de la aplicación. En el otro extremo, después de recibir el mensaje, se procede, a su vez, a terminar la partida y a mostrar por pantalla el mensaje de advertencia de desconexión del otro extremo, explicando el motivo por el que la partida se ha cerrado. Tras haberlo leído y pulsado el botón *Aceptar*, se muestra también en él el menú inicial.

En el caso en que la partida termine de forma natural, por agotamiento de las vidas de al menos uno de los dos jugadores, y no se deseara iniciar una nueva, el diagrama sería similar al aquí mostrado. La única diferencia es que, para cuando llegara el mensaje de tipo *STOP* al otro extremo, la partida ya habría terminado. Además, se pulsaría el botón *No* (de no jugar una nueva partida) desde la pantalla de *game over*, en lugar del botón *Salir* de la partida.

4.5.4.2. Mensajes de datos

Se utilizan para comunicar al otro extremo alguna información referente a la partida o a cualquiera de los elementos que la componen (naves, pelotas, ladrillos, *power-ups*…). Se reconocen al comenzar por el identificador *DATA*. A continuación, le sigue un identificador del tipo de mensaje de datos que se trata. Los diferentes identificadores que especifican qué tipo de mensaje de datos es, se analizan a continuación:

- Mensaje *BALLS*: sirve para transmitir al otro extremo información sobre la posición de todas las pelotas presentes en ese momento (sus coordenadas y cuánto y hacia dónde se mueven en cada eje). Dado que el número de pelotas activas en un momento concreto de la partida es variable, la longitud de este mensaje también lo será. En la figura 4.27 se observa el esquema de la estructura de este tipo de mensajes.

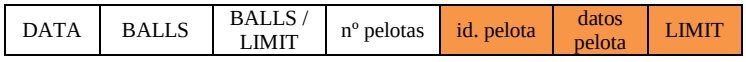


Figura 4.27. Formato de los mensajes BALLS

En color anaranjado se representan aquellos campos que se pueden repetir una o más veces (según el número total de pelotas que haya). El identificador de cada pelota se indica en función del identificador asignado en el terminal receptor (las pelotas que controla el jugador en su propio terminal pasan a ser identificadas como las pelotas que controla el jugador del terminal remoto en el otro terminal). Se envía cada vez que hay una colisión entre dos pelotas y cuando una pelota colisiona contra una de las paredes laterales. Para distinguir un caso del otro, se incluye, en el tercer campo, el identificador *BALLS* para indicar que se trata de una colisión entre dos pelotas o *LIMIT* si es una colisión contra una pared lateral. Así, el terminal remoto sabrá si debe reproducir el sonido de rebote tras la colisión entre pelotas o no.

- Mensaje *BALLSHIP*: se envía siempre que una pelota colisiona con la nave que maneja el jugador, y también cuando el modo de juego es *deathmatch* y la pelota del otro jugador no colisiona sobre la nave (y por tanto, rebota al llegar al fondo de la pantalla). Sirve para comunicar al otro extremo la posición de todas las pelotas (además del sentido de movimiento de cada una de ellas), de nuevo identificadas desde el punto de vista del receptor, y de la nave propia en el momento de la colisión. Se envía toda esta información para no dar lugar a incoherencias en la ejecución en uno y otro terminal. Cuando la pelota del otro jugador colisiona en la nave propia en el modo *deathmatch*, además de rebotar, el jugador que maneja la nave pierde una vida, como ya se comentó en el capítulo 3 al explicar en detalle los dos modos de juego posibles. Para distinguir este caso del resto de posibles rebotes sobre la nave, se añade en el mensaje el identificador *BOUNCEKILLING*, aludiendo al hecho de que el rebote de la pelota del otro jugador provoca la pérdida de una vida. Adicionalmente, a este identificador le puede seguir el identificador *POWERUP*, indicando así que el *power-up* que atrapa la pelota está activo (esta información podría pensarse que es redundante, en tanto en cuanto ambos terminales saben ya de antemano si el *power-up HOLDBALL* está activo o no para ese jugador; sin embargo, esta notificación es nuevamente para evitar posibles incoherencias, al tratarse de un evento de colisión entre nave y pelota). En cualquier otro caso, le sigue en

su lugar el identificador *LIMIT*. En la figura 4.28 se puede observar la estructura de este tipo de mensajes.

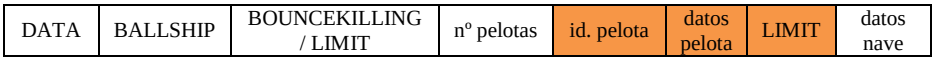


Figura 4.28. Formato de los mensajes BALLSHIP

- Mensaje *BRICK*: este tipo de mensajes se envía cuando hay una colisión con un ladrillo. La colisión, como ya se comentó en el capítulo anterior, puede ser entre una pelota y un ladrillo, o entre un proyectil y un ladrillo. El terminal remoto necesita distinguir entre un caso y el otro para saber si necesita hacer rebotar una pelota o eliminar un proyectil de la pantalla. Es por ello que en este mensaje se incluye también el identificador *SHOT*, si se trata de un proyectil, o el identificador *LOCAL* o *REMOTE* en su lugar, según si se trata de la pelota propia o de la del otro jugador, respectivamente. En el primer caso, se añade la información necesaria para distinguir qué proyectil es el que impacta contra el ladrillo. Además, estos mensajes adjuntan, aparte del identificador del correspondiente ladrillo golpeado, información sobre la posición de todas las pelotas que hay en juego –a modo de sincronización, de nuevo para evitar incoherencias en la ejecución en los dos terminales– y el sentido de movimiento de una eventual cápsula que pueda desprenderse del ladrillo recién impactado. La figura 4.29 muestra el aspecto que presentan los dos posibles tipos de mensajes *BRICK*. En tono grisáceo se presentan los campos que podrían estar presentes o no.

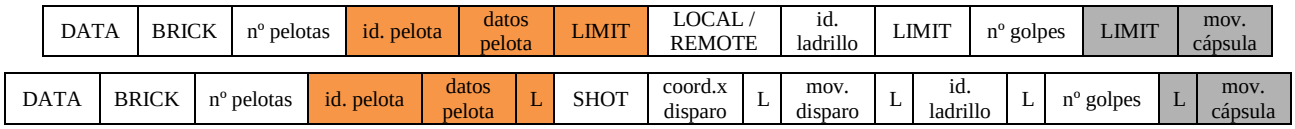


Figura 4.29. Formato de los mensajes BRICK

- Mensaje *GAMEOVER*: este mensaje se transmite al extremo remoto una vez el jugador ha agotado todas sus vidas. No le acompaña ningún otro identificador, puesto que no necesita precisar ninguna otra información adicional, simplemente fuerza en el otro terminal la finalización del juego

por la consumición de todas sus vidas. La figura 4.30 representa la sencilla estructura de este tipo de mensaje.

Figura 4.30. Formato de los mensajes GAMEOVER

- Mensaje *LISTPOWERUP*: el terminal maestro envía al esclavo este mensaje al inicio de cada nivel para indicarle la lista de ladrillos a los que se les va a asociar un determinado *power-up*, junto con el identificador del *power-up* correspondiente. En la figura 4.31 se observa el esquema de este tipo de mensajes.

Figura 4.31 Formato de los mensajes LISTPOWERUP

La longitud del mensaje dependerá del número de *power-ups* que vaya a disponer el nivel. Dicha cantidad se especifica a continuación de la cabecera que identifica a estos mensajes.

- Mensaje *OUTBALL*: se envía cuando se pierde una pelota. La notificación puede hacer referencia a una pelota del propio jugador o a una del otro. Para diferenciarla, se añade en el mensaje el identificador de dicha pelota, siempre desde el punto de vista del terminal que recibe el mensaje.
  Cuando un jugador dispone de más de una pelota al mismo tiempo (por activación del *power-up* que triplica el número de pelotas) y pierde la pelota con la que inició la vida (en el caso del jugador local, es la pelota 0, mientras que para el otro jugador, se trataría de la pelota 3), se realiza un proceso de intercambio de pelotas para que el identificador de la misma siga manteniendo en pantalla a una pelota activa (por ejemplo, se pierde la pelota 0, pero la 1 sigue activa, de modo que la pelota identificada por 1 pasa a identificarse por la 0, y viceversa). En estos casos, se añade el identificador *CHANGEBALL* a continuación del identificador *OUTPUT*, y las dos pelotas

involucradas en el intercambio. Podría pensarse que este proceso resulta innecesario a la hora de controlar y mantener identificadas durante la partida las pelotas de un mismo jugador. De hecho, bastaría con comprobar si hay pelotas activas o no, para saber si el jugador ha perdido una vida, con independencia de si está activa o no la pelota con la que inició esa vida. Sin embargo, mantener identificada a una pelota como la principal, distinguiéndola del resto que eventualmente pueden añadirse a ésta durante el desarrollo de la partida, facilitará la tarea de hacer comprobaciones durante el código (comprobar si la pelota aún no ha sido lanzada, utilizarla como pelota de referencia para añadir las otras dos al activar el *power-up* que triplica el número de pelotas, etc). De otra forma, habría que implementar un algoritmo que permitiera saber cuál es la pelota que cumple ese rol. La figura 4.32 muestra los dos formatos que admite de este tipo de mensajes.

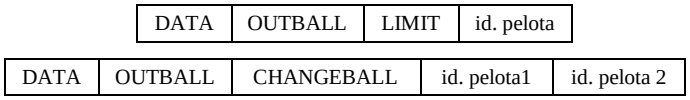


Figura 4.32. Formato de los mensajes OUTBALL

- Mensaje *POWERUP*: sirve para indicar al otro extremo que un determinado *power-up* se ha activado tras recoger con la nave su correspondiente cápsula. Este mensaje se envía cuando la nave controlada por el propio usuario es la que recoge la cápsula. En él se especifica el *power-up* que se ha activado, junto con las coordenadas y sentido de movimiento de su cápsula para borrarla de la pantalla en el terminal remoto, toda vez que su recogida ya ha sido efectuada por la nave. La figura 4.33 muestra el aspecto que presenta este tipo de mensajes.

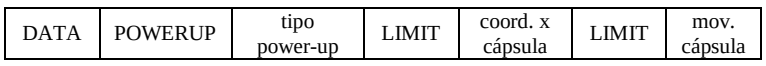


Figura 4.33. Formato de los mensajes POWERUP

- Mensaje *RESTART*: este mensaje se envía al otro extremo cuando hay que reubicar en pantalla la pelota propia –por tanto, la pelota del otro jugador para el terminal remoto–, concretamente sobre la nave, para continuar con el

juego una vez que la pelota se ha perdido y aun le quedan vidas al jugador (si no le quedaran, se enviaría el mensaje *GAMEOVER*). En la figura 4.34 se observa la estructura de este tipo de mensaje.

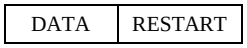


Figura 4.34. Formato de los mensajes RESTART

- Mensaje *SHIP*: este mensaje se transmite al otro extremo cada vez que el jugador desplaza su nave hacia la izquierda o hacia la derecha. Al enviarlo se adjunta la nueva posición que ocupa la nave en pantalla. La figura 4.35 muestra el formato de estos mensajes.

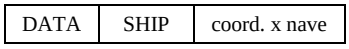


Figura 4.35. Formato de los mensajes SHIP

- Mensaje *SHOT*: sirve para indicar al otro extremo que el jugador ha disparado un nuevo par de proyectiles (al enviar este mensaje se asume que el jugador tiene activado el *power-up* de disparo). En este mensaje se adjuntan las coordenadas iniciales de la nueva pareja de proyectiles que tendrán que representarse por pantalla. En la figura 4.36 puede apreciarse el aspecto que presentan estos mensajes.

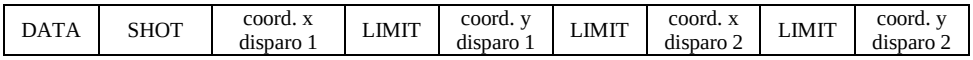


Figura 4.36. Formato de los mensajes SHOT

El sentido hacia el que se mueven los proyectiles no es necesario especificarlo, ya que el extremo que genera este mensaje es siempre el del jugador que ha disparado, por lo que los proyectiles saldrán hacia arriba, y por tanto hacia abajo al ser representados en el otro terminal. Por otra parte, el movimiento es sólo vertical, así que no hay que indicar ningún desplazamiento sobre el eje X.

- Mensaje *SPEED*: este mensaje se transmite al otro extremo cuando la velocidad de la pelota, o pelotas, controladas por el jugador incrementan su velocidad como consecuencia de mantenerse en pantalla durante un tiempo prolongado. Este tiempo, viene determinado por la constante *TIMER_SPEED*, definida en la clase *Juego.java*, que se comentará más adelante. Conviene señalar que hay que distinguir este caso del caso en que la velocidad viene aumentada por activación del correspondiente *power-up*, en cuyo caso, el incremento de dicha velocidad se tratará en el terminal remoto mediante el mensaje de tipo *POWERUP*, como ya se indicó anteriormente, y no a través de un mensaje de este tipo. En la figura 4.37 puede observarse su estructura.

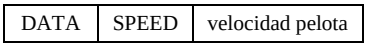


Figura 4.37. Formato de los mensajes SPEED

- Mensaje *THROW*: sirve para informar al extremo remoto de que el jugador ha lanzado su pelota, bien accionando manualmente dicho lanzamiento con el botón de disparo, bien automáticamente, después de transcurrir unos segundos. La figura 4.38 describe el formato de este tipo de mensaje.

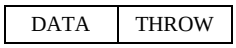


Figura 4.38. Formato de los mensajes THROW

Como ya se ha comentado en párrafos anteriores, estos mensajes de datos, al llegar al otro extremo son decodificados e interpretados desde una función contenida en *Juego.java*, en concreto, la función *actualizacionRemota(String datos)*.

4.6. ELEMENTOS DEL VIDEOJUEGO

Los diferentes elementos que integran una partida de la versión multijugador de "Arkanoid" que se desarrolla en el presente proyecto pueden observarse en el diagrama

de la figura 4.39. Puede observarse que una partida se compone de seis pelotas, dos naves, un gestor de mapas que administra un total de diez mapas (todos los mapas disponibles en el videojuego), compuestos cada uno de ellos por uno o más ladrillos y un conjunto de posibles *power-ups* que, de existir, deberán estar asociados cada uno a un solo ladrillo. Cuando se eliminan estos ladrillos, generan la correspondiente cápsula con los *power-ups* que estaban asociados a ellos y la añaden a la partida.

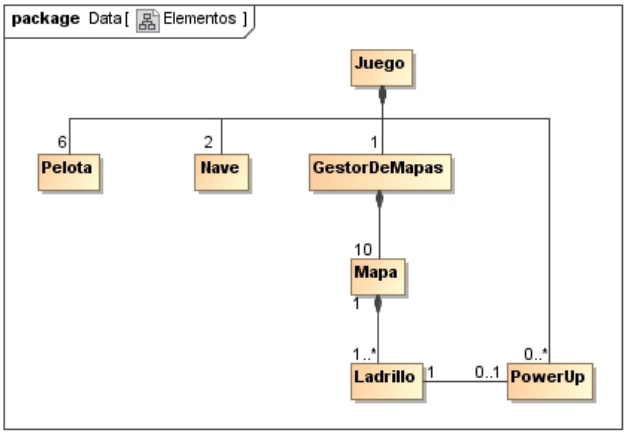


Figura 4.39. Diagrama de clases de los elementos de una partida

4.6.1. PELOTAS

La información relativa a las pelotas que aparecen en el videojuego, sus propiedades y características se recogen en la clase *Pelota.java*, del paquete *juego*. En la figura 4.40 se muestra el diagrama de esta clase, donde se indican sus atributos y los métodos que ofrece. Puede observarse que el modificador de acceso de todos los métodos, incluido el propio constructor, y de algunos atributos –concretamente algunas constantes– es *package*. Esto es así porque todas las llamadas a estos métodos y atributos se realizan desde clases que pertenecen al mismo paquete donde se ubica *Pelota.java*, es decir, el paquete *juego*. Por tanto, no hay necesidad de darles mayor alcance.

En esta clase se definen cuatro constantes, que se corresponden con los primeros cuatro atributos del diagrama. El significado de cada uno de ellos se detalla a continuación:

Figura 4.40. Diagrama de la clase Pelota.java

- *DEFAULT_MOVX* y *DEFAULT_MOVY*: establecen el movimiento por defecto sobre los dos ejes cuando la pelota se lanza inicialmente desde la nave.

- *DIAMETRO_PELOTA*: indica cuál es el diámetro, o lo que es lo mismo, el lado del cuadrado del *sprite* que representa la pelota en el videojuego. Su valor es de 9 píxeles. Es de utilidad cuando se realiza el proceso de inversión de coordenadas para representar la misma pelota en el terminal remoto, ya que hay que considerar el espacio que ocupa la imagen (ancho y alto) a la hora de fijar la posición. Por el mismo motivo, también se necesita para estudiar las posibles colisiones y rebotes.

- *MAX_VELOCIDAD*: determina la máxima velocidad a la que se puede mover una pelota. El valor de este parámetro no trasciende a otras clases, solo se utiliza a nivel interno para regular la velocidad que adopta la pelota durante la partida, por lo que su modificador de acceso es *private*.

Además de las constantes, la clase también define una serie de atributos privados que se analizarán con mayor detenimiento a continuación:

- **contacto**: indica si la pelota está adherida a la nave (*true*) o no (*false*). Esta información resulta útil por varios motivos. Es necesaria para saber si al mover la nave hay que mover con ella también a la pelota; por otro lado, también determina si se debe lanzar la pelota toda vez que hayan transcurrido unos segundos sin que el jugador haya efectuado el lanzamiento de forma manual; y por último, sirve para saber si es necesario reubicar la pelota cuando le llega la correspondiente notificación desde el extremo remoto, o por el contrario ya se ha hecho desde el propio terminal.

  Los métodos que permiten acceder a este atributo para obtener su valor o modificarlo son los correspondientes métodos *get* y *set*: *getContacto()* y *setContacto(boolean contacto)*.

- **imagenPelota**: este atributo almacena la imagen que representa a la pelota. La imagen se carga cuando se crea el objeto, indicando en el parámetro del constructor la ruta donde se encuentra la imagen. Posteriormente, se puede consultar su valor a través del método *getImagen()*.

- **movPelotaX**, **movPelotaY**: contienen el valor instantáneo del movimiento de la pelota en cada eje de referencia. La consulta de estos valores se hace desde los métodos *getMovPelotaX()* y *getMovPelotaY()*, mientras que dichos atributos pueden actualizarse mediante los métodos *setMovPelotaX(int movPelotaX)*, *setMovPelotaY(int movPelotaY)* y *setMovimiento(int movPelotaX, int movPelotaY)*, que llama a los dos métodos anteriores para completar la modificación de los parámetros.

- **pelotaX**, **pelotaY**: corresponden a las coordenadas donde se ubica la pelota. Concretamente, se refieren a la esquina superior izquierda de la imagen que la representa. Dichas posiciones se pueden obtener a partir de los métodos *getCoordenadaX()* y *getCoordenadaY()*, y modificar desde los métodos *setCoordenadaX(int pelotaX)* y *setCoordenadaY(int pelotaY)*.

Adicionalmente, se emplea por comodidad el método *setActualizacion(int pelotaX, int pelotaY, int movPelotaX, int movPelotaY)* para modificar con una sola llamada a un método, tanto las coordenadas como el movimiento de la pelota.

- **velocidad**: indica el exceso de velocidad aplicado a la pelota. Inicialmente es de 2, pero ese valor podrá verse modificado durante la partida, bien por la recepción de los correspondientes *power-ups* que aumenten o disminuyan la velocidad de la misma, bien porque la pelota lleve mucho tiempo sin perderse e incremente paulatinamente su velocidad. Con *getVelocidad()* se puede consultar su valor instantáneo, mientras que con los métodos *decrementarVelocidad()* e *incrementarVelocidad()* se reduce y aumenta la misma, respectivamente.

Estos cinco últimos parámetros entran también en juego en el método *moverPelota()* para desplazarla en cada ciclo de ejecución del bucle principal del juego. Al valor de *pelotaX* y *pelotaY* se les suma, siempre que la pelota no esté adherida a la nave, el valor de *movPelotaX* y *movPelotaY*, respectivamente, y además, el exceso de velocidad, en valor absoluto, asociado a ella.

Después de analizar los atributos de la clase y relacionarlos con los correspondientes métodos de consulta y modificación de los mismos, quedan aun un par de métodos por comentar y justificar su uso. Son *distanciaEjeX(Pelota pelota)* y *distanciaEjeY(Pelota pelota)*. Se utilizan para calcular la distancia a la que se encuentran dos pelotas entre sí –la que llama al método y la que se pasa por parámetro–, para poder determinar si hay colisión entre ellas. Si esa distancia es menor que el diámetro de la pelota, puede haber solapamiento entre las dos pelotas al representarlas, por lo que habría que resolver una posible colisión entre ambas.

4.6.2. NAVES

En la clase *Nave.java* se recogen todos los atributos y métodos necesarios para caracterizar cada una de las dos naves del videojuego. En la figura 4.41 se muestra su diagrama.

De nuevo, el modificador de acceso del constructor y de todos los métodos vuelve a ser *package*, por la misma razón de antes, el uso que se hace de ellos no trasciende a clases de otros paquetes. Igualmente ocurre con los atributos que no se han definido como privados –las constantes cuyo valor se precisa conocer desde la clase principal del código–.

En esta clase se definen once constantes, los primeros once atributos que aparecen listados en el diagrama. Pueden agruparse según el rol que desempeñan dentro de la clase:

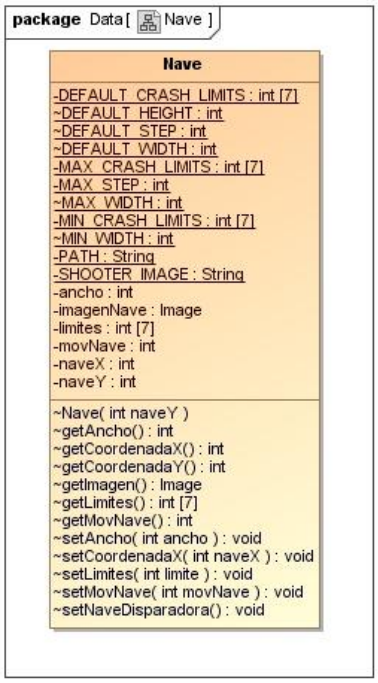


Figura 4.41. Diagrama de la clase Nave.java

- *DEFAULT_HEIGHT*, *DEFAULT_WIDTH*: indican la altura y la anchura por defecto del *sprite* que representa la nave, respectivamente. La altura se mantendrá constante durante toda la partida, por lo que no será necesario crear un atributo que almacene un valor variable del mismo a lo largo del juego. La anchura, por su parte, a diferencia de la altura, si podrá oscilar durante la partida, ya que hay *power-ups* que alteran su valor aumentándolo o disminuyéndolo, por lo que habrá un atributo que se encargue de guardar los posibles valores que pueda adoptar este parámetro durante la misma. Sus valores son 9 píxeles para la altura de la nave y 41 para su anchura.

- *MIN_WIDTH*, *MAX_WIDTH*: fijan la anchura mínima y máxima, respectivamente, que puede tomar el *sprite* que representa la nave.

- Constantes que delimitan las regiones de rebote de la nave: Se trata de tres constantes que acotan las zonas en las que se divide la nave en cada uno de sus tres tamaños posibles, para determinar qué ángulo de rebote adopta la pelota al impactar contra ésta. Cada constante es un *array* de siete posiciones en las que se almacenan los valores de los siete límites que encuadran las ocho zonas resultantes en las que se fracciona la nave para un tamaño de nave determinado. En la tabla 4.18 se presenta una lista que recoge los identificadores de estos tres *arrays*, indicando el ancho de la nave para el que se definen y el contenido de estos vectores.

Identificador	Tam. Nave	Valor
DEFAULT_CRASH_LIMITS	DEFAULT_WIDTH	{-5,0,7,16,25,32,37}
MAX_CRASH_LIMITS	MAX_WIDTH	{-2,6,16,28,40,50,58}
MIN_CRASH_LIMITS	MIN_WIDTH	{-6,-2,4,11,18,24,28}

Tabla 4.18. Constantes que delimitan las regiones de rebote

Es de notar que las posiciones de los límites van referidas en función de la posición de la nave. Por ejemplo, para el caso en que el ancho de la nave es el ancho por defecto, la primera frontera se establecería 5 píxeles a la izquierda de la coordenada X de la nave (o sea, el contenido del atributo **naveX** menos 5); la segunda coincide con la posición de la nave, la tercera 7 píxeles a la derecha, y así sucesivamente… el hecho de que haya valores negativos en estos *arrays* se debe a que hay que considerar el grosor (el diámetro) de la pelota a la hora de estudiar las posibles colisiones. La figura 4.42 servirá para aclarar este aspecto.

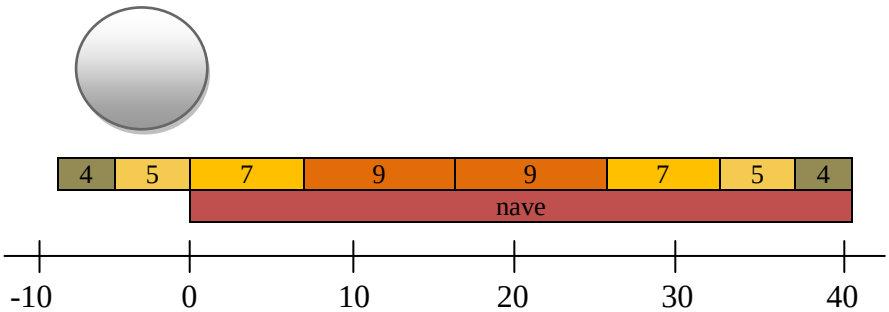


Figura 4.42. Distribución de las regiones de rebote

En ella se muestran las ocho regiones de rebote que, para el caso por defecto, surgen como consecuencia del establecimiento de las siete fronteras o límites definidos en *DEFAULT_CRASH_LIMITS*. En cada región se indica el número de píxeles de ancho que tiene cada una. Si la coordenada X del *sprite* que representa la pelota (es decir, la coordenada X de la esquina superior izquierda de dicha imagen) está más a la izquierda que la primera frontera, pero no más de 4 píxeles, entonces se interpreta que la región sobre la que rebota es la primera. Nótese que el contacto existe, pues el diámetro de la pelota es de 9 píxeles, así que estaría impactando por su extremo inferior derecho. Igualmente, si la coordenada X de la pelota se encuentra entre 5 píxeles a la izquierda de la nave y la propia coordenada X de la nave, el impacto tendría lugar sobre la segunda región de rebote, adoptando el correspondiente ángulo de rebote para esta zona, y así sucesivamente… En el caso que se ilustra en la figura, la coordenada X de la pelota está más a la izquierda que la primera frontera, por lo que desde la clase principal, al resolver la colisión de la pelota con la nave, se le asignará a la primera el ángulo correspondiente al de la primera región de rebote.

- *DEFAULT_STEP*, *MAX_STEP*: ambos hacen referencia a la cantidad de píxeles que se desplaza la nave a izquierda y derecha cada vez que el jugador pulsa las flechas izquierda y derecha, respectivamente. La primera constante indica el valor por defecto de este parámetro, mientras que la segunda establece el máximo valor que puede llegar a tomar, dado que es una cantidad que puede verse alterada por efecto de los *power-ups* que la aumentan o disminuyen, y que por tanto, precisará un atributo que registre el valor instantáneo de este parámetro durante la partida.

- *PATH*: directorio donde se ubica la imagen que representa a la nave.

- *SHOOTER_IMAGE*: identificador del archivo que contiene la imagen que representa a la nave cuando se activa el *power-up* de disparo.

Junto con las constantes, también se definen en la clase una serie de atributos privados que almacenan los valores de los parámetros de interés. Estos atributos son los siguientes:

- **ancho**: contiene el valor del ancho de la nave. Dicho valor ya se ha aclarado que podrá oscilar entre los valores dados por *MIN_WIDTH* y *MAX_WIDTH*. Al igual que *DIAMETRO_PELOTA*, conocer este parámetro sirve para el proceso de inversión de coordenadas al representar la nave en el otro terminal, al igual que para determinar la posibilidad de colisiones y rebotes entre una pelota y una nave. Además, de su valor dependerá la capacidad de movimiento de la nave (cuanto más ancha sea, ocupará más espacio, y por tanto menos se necesitará mover para llegar a un extremo o a otro). Dicho valor podrá ser alterado cuando el usuario recoja un *power-up* cuyo efecto consista en aumentar o reducir el ancho de la nave. Para modificarlo, se invoca el método *setAncho(int ancho)*, mientras que para consultar su valor se utiliza el método *getAncho()*.

- **imagenNave**: este atributo almacena la imagen que representa a la nave. Inicialmente, cuando se crea el objeto se carga la imagen por defecto, tomando como referencia el ancho y el alto por defecto. Posteriormente, el aspecto de la nave podrá cambiar si se invoca el método *setAncho(int ancho)*, comentado en el punto anterior, o si se adquiere el *power-up* de disparo, en cuyo caso se llamará al método *setNaveDisparadora()*. El contenido de este atributo se puede consultar a través del método *getImagen()*.

- **limites**: es un *array* de siete posiciones que almacena las coordenadas de las fronteras que dan lugar a las ocho regiones en las que se divide la nave para determinar el ángulo de rebote de la pelota al impactar contra ésta. Como ya se ha dicho, el ancho de la nave puede variar a lo largo de la partida, y con él, han de modificarse también los límites de estas regiones. Así pues, este atributo contiene las fronteras de las regiones válidas para la nave en cada momento. Su valor puede ser consultado mediante el método *getLimites()*, el cual devolverá el *array* íntegro, y modificado a través del método *setLimites(int limite)*, donde en el parámetro *limite* se indica el conjunto de límites que debe adoptar la nave (los

correspondientes al valor máximo del ancho de la nave, mediante *MAX_WIDTH*; los asociados al valor mínimo del ancho de la nave, a través de *MIN_WIDTH*; y los que se aplicarían para el valor por defecto del ancho de la nave, con cualquier otro valor de *limite*, distinto de los dos anteriores).

- **movNave**: registra la cantidad de píxeles que se moverá la nave hacia un lado u otro cada vez que el jugador pulse la flecha izquierda o derecha. Esta cantidad, como ya se anunció anteriormente, podrá oscilar en función de los eventuales *power-ups* activados cuyo efecto sea el de modificar su valor. Para ello, se invocará al método *setMovNave(int movNave)*. Para consultar su valor, sin embargo, se usará el método *getMovNave()*.

- **naveX**, **naveY**: indican la posición, en los dos ejes de referencia, de la esquina superior izquierda del *sprite* que representa la nave. La coordenada Y, una vez fijada al crear el objeto –es el parámetro que se pasa en el constructor– no necesita ser modificada durante el desarrollo de la partida (la nave propia siempre estará en la parte inferior, a la misma altura, e igualmente la nave del otro jugador en el extremo superior), por lo que sólo podrá ser consultada, no modificada, a través del método *getCoordenadaY()*. La coordenada X, en cambio, si cambiará cada vez que el jugador la desplace a izquierda y derecha, por lo que podrá ser consultada y modificada mediante el uso de los métodos *getCoordenadaX()* y *setCoordenadaX(int naveX)*, respectivamente.

4.6.3. POWER-UPS

Los identificadores de los tipos de *power-ups* implementados en el videojuego, así como las características de las cápsulas que contienen a cada uno de ellos, se explicitan en la clase *PowerUp.java*. En la figura 4.43 puede verse su diagrama.

Al igual que en las clases anteriores, los métodos y los atributos que necesiten ser invocados desde fuera de la clase, serán llamados siempre desde dentro del mismo paquete, por lo que su modificador de acceso será *package*. El resto, naturalmente, serán privados.

En esta clase hay definidas muchas constantes. No obstante, éstas pueden ser agrupadas según su propia funcionalidad dentro del código. Así, se podrá distinguir entre constantes que designan identificadores de *power-ups*, directorios de almacenamiento de sus respectivas imágenes y posibles efectos visuales y otras constantes de utilidad, como se detallará a continuación:

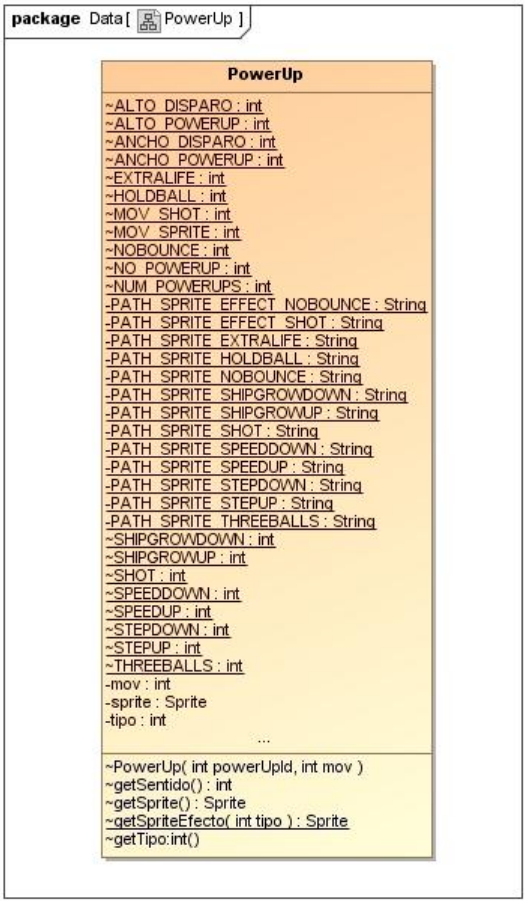


Figura 4.43. Diagrama de la clase PowerUp.java

- Identificadores de *power-ups*: son las constantes que permiten reconocer los diferentes *power-ups* implementados en el videojuego. En la tabla 4.19 se enumeran por orden alfabético y haciendo una referencia al título en castellano utilizado en el capítulo 3 para identificarlos.

  En este punto, también es de mencionar la existencia del identificador *NO_POWERUP* para indicar la ausencia de *power-up*. La mayoría de ladrillos no desprenderán una cápsula, incluso algunos de ellos ni siquiera se podrán destruir. Este parámetro será el que se utilice para aquellos casos.

PowerUp	Significado en castellano
EXTRALIFE	Vida extra
HOLDBALL	Atrapar la pelota
NOBOUNCE	No rebote
SHIPGROWDOWN / SHIPGROWUP	Disminuir / Aumentar la velocidad de la nave
SHOT	Disparo
SPEEDDOWN / SPEEDUP	Disminuir / Aumentar el ancho de la nave
STEPDOWN / STEPUP	Disminuir / Aumentar la velocidad de la pelota
THREEBALLS	Triplica el número de pelotas

Tabla 4.19. Identificadores de los power-ups implementados

Además, se dispone asimismo de la constante *NUM_POWERUPS*, que indica el total de *power-ups* implementados en el código. Esta cifra resulta de utilidad para crear desde la clase principal un *array* de esas dimensiones que pueda albergar la información de qué *power-ups* se encuentran activos en cada instante.

- Directorios o *paths*: estas constantes pueden almacenar el *path* completo de las imágenes que representan a las cápsulas asociadas a cada *power-up* o bien las imágenes relativas al efecto que producen determinados *power-ups*. A continuación, en la tabla 4.20 se enumeran, por orden alfabético cada uno de estos *paths*, indicando el tipo de imagen al que permiten acceder.

Path	Imagen a la que permite acceder
PATH_SPRITE_EFFECT_NOBOUNCE	Estela de color verde para el *power-up NOBOUNCE*
PATH_SPRITE_EFFECT_SHOT	Proyectil para el *power-up SHOT*
PATH_SPRITE_EXTRALIFE	Cápsula asociada al *power-up EXTRALIFE*
PATH_SPRITE_HOLDBALL	Cápsula asociada al *power-up HOLDBALL*
PATH_SPRITE_NOBOUNCE	Cápsula asociada al *power-up NOBOUNCE*
PATH_SPRITE_SHIPGROWDOWN	Cápsula asociada al *power-up SHIPGROWDOWN*
PATH_SPRITE_SHIPGROWUP	Cápsula asociada al *power-up SHIPGROWUP*
PATH_SPRITE_SHOT	Cápsula asociada al *power-up SHOT*
PATH_SPRITE_SPEEDDOWN	Cápsula asociada al *power-up SPEEDDOWN*
PATH_SPRITE_SPEEDUP	Cápsula asociada al *power-up SPEEDUP*
PATH_SPRITE_STEPDOWN	Cápsula asociada al *power-up STEPDOWN*
PATH_SPRITE_STEPUP	Cápsula asociada al *power-up STEPUP*
PATH_SPRITE_THREEBALLS	Cápsula asociada al *power-up THREEBALLS*

Tabla 4.20 Identificadores de los directorios relativos a los power-ups

- *ALTO_DISPARO, ANCHO_DISPARO*: representan las dimensiones de los proyectiles que se lanzan cuando está activado el *power-up* de tipo *SHOT*. El conocimiento de estos valores es necesario en el proceso de inversión de

coordenadas para poder representar correctamente el mismo proyectil en el otro terminal. También sirven para estudiar posibles colisiones contra los ladrillos.

- *ALTO_POWERUP*, *ANCHO_POWERUP*: determinan la altura y la anchura, respectivamente, de cualquiera de las cápsulas que dan lugar a los posteriores *power-ups*. Estos valores coinciden con las dimensiones de los ladrillos. La altura de la cápsula, a su vez, sirve para comprobar si la cápsula en cuestión ha salido de la pantalla (sólo se mueve hacia arriba o hacia abajo, no hacia los lados, por lo que el único parámetro de interés para comprobar si la cápsula se sale sin ser recogida por la correspondiente nave es la altura).

- *MOV_SHOT*, *MOV_SPRITE*: establecen la velocidad, medida en número de píxeles que se avanza por cada ciclo de ejecución del bucle principal, a la que se mueven los proyectiles y las cápsulas, respectivamente. En el caso de los proyectiles, el sentido de movimiento lo determinará el jugador que los lance (es decir, hacia arriba si los lanza el jugador que controla el terminal y hacia abajo si corresponden a una actualización de un mensaje remoto en el que se indica que el otro jugador ha disparado proyectiles), mientras que en las cápsulas, el sentido de movimiento lo determinará el jugador al que pertenezca la pelota que ha impactado contra el ladrillo del que se ha desprendido (hacia abajo si ha sido la pelota del jugador que controla el terminal y hacia arriba si se trata de la pelota del otro jugador).

En la clase también se definen unos pocos atributos, cuyo significado y utilidad son bastante intuitivos:

- **mov**: determina el sentido hacia el que se mueve la cápsula asociada al *power-up* en cuestión. Como ya se ha dicho en el punto anterior, la cápsula puede desplazarse hacia arriba o hacia abajo, según qué pelota haya golpeado el correspondiente ladrillo del que procede. Dicho parámetro se fija desde el constructor, una vez se crea el objeto, y posteriormente puede consultarse desde el método *getSentido()*.

- **sprite**: hace referencia al *sprite* que representa a la cápsula que identifica un determinado *power-up*. En la figura 4.44 se muestran –por orden alfabético según el *power-up* al que identifican– los *sprites* que generan las diferentes cápsulas en movimiento, una vez se desprenden del ladrillo recientemente eliminado.

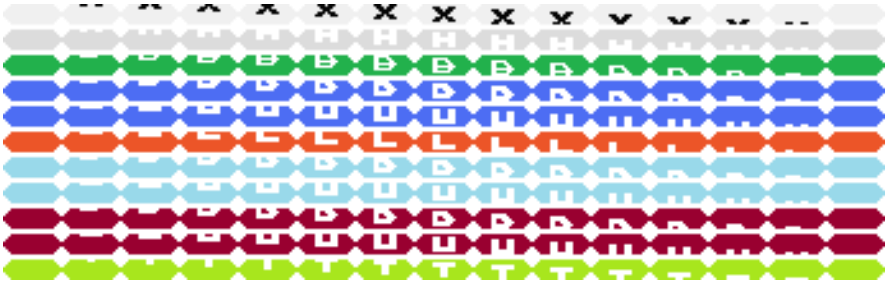


Figura 4.44. Sprites de las cápsulas

Se puede observar que cada *sprite* es una secuencia de imágenes, donde cada imagen representa a la misma cápsula en diferentes posiciones. Cuando la secuencia se pone en marcha y avanza de imagen en imagen a medida que la cápsula desciende (o asciende) genera el efecto óptico de rotación de la cápsula.
El *sprite* se determina cuando se crea el objeto, a partir de la información que se le pasa por parámetro al constructor (es decir, según qué tipo de *power-up* se trate, habrá que escoger una secuencia de imágenes u otra, y según en qué sentido se desplace la cápsula, ésta rotará en un sentido o en otro, lo que repercutirá en el sentido en el que se recorrerá la secuencia). Posteriormente, dicho atributo podrá ser consultado desde el método *getSprite()*.

- **tipo**: determina el tipo de *power-up* asociado a la cápsula que se ha desprendido del ladrillo. Conocer su valor servirá, evidentemente, para saber qué *power-up* es el que hay que activar si finalmente la nave ha recogido la cápsula. Al igual que los otros dos atributos, se fija cuando se crea el objeto, al pasarse su valor por parámetro en el constructor, y se consulta desde el método *getTipo()*.

Por último, queda por mencionar el método estático *getSpriteEfecto(int tipo)*. Simplemente recupera el *sprite* del efecto del *power-up* activado, si procede. Los

efectos que puede devolver son dos que ya se mencionaron anteriormente: la estela de color verde que deja la pelota cuando está activado el *power-up NOBOUNCE* y el proyectil que lanza la nave cuando dispone del *power-up SHOT*.

4.6.4. LADRILLOS

La información que define y caracteriza a cada ladrillo, se encuentra en la clase *Ladrillo.java*. En la figura 4.45 se muestra su diagrama.

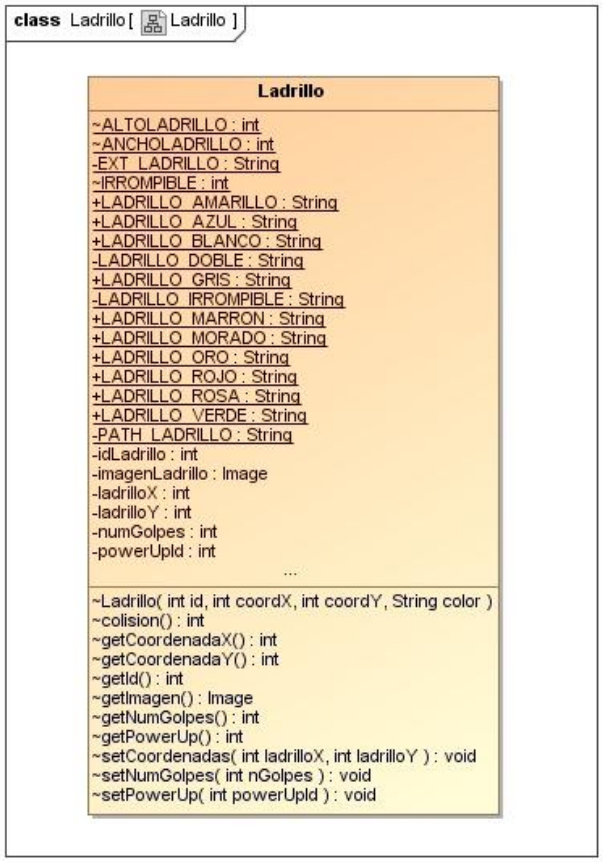


Figura 4.45. Diagrama de la clase Ladrillo.java

De forma análoga a las clases anteriormente expuestas, los métodos y las constantes cuyo uso se limite al de clases que pertenezcan al mismo paquete, tendrán el modificador de acceso *package*. Sin embargo, hay un grupo de constantes que serán necesarias desde la clase *Fichero.java*, ubicada en el paquete *util*, por lo que serán declaradas como públicas.

Nuevamente, el análisis de las constantes definidas en esta clase resulta más cómodo si se hace agrupándolas según su función:

- Tipos de ladrillo: la mayoría de estas constantes son para indicar un tipo de ladrillo, según el color, de entre los diferentes tipos definidos para el videojuego desarrollado. Servirán para poder etiquetar correctamente los ladrillos, una vez se ha leído e interpretado la información contenida en los ficheros de mapas, que ya se comentaron en la sección de ficheros. La tabla 4.21 resume en una sencilla lista cuáles son estos identificadores y los ordena en función del número de golpes necesarios para eliminarlos.

Nº de golpes	Identificador de ladrillo
1	*LADRILLO_AMARILLO* *LADRILLO_AZUL* *LADRILLO_BLANCO* *LADRILLO_MARRON* *LADRILLO_MORADO* *LADRILLO_ROJO* *LADRILLO_ROSA* *LADRILLO_VERDE*
2	*LADRILLO_GRIS*
irrompible	*LADRILLO_ORO*

Tabla 4.21. Identificadores de los ladrillos implementados

Puede deducirse de la tabla que, en realidad, los identificadores *LADRILLO_DOBLE* y *LADRILLO_IRROMPIBLE* hacen referencia a los mismos identificadores que *LADRILLO_GRIS* y *LADRILLO_ORO*, respectivamente.

- *PATH_LADRILLO*, *EXT_LADRILLO*: indican la ruta donde se alojan todas las imágenes que representan a los ladrillos y la extensión de las mismas, respectivamente. Ambos atributos, junto con los identificadores de ladrillo, darán lugar al *path* completo donde se ubica cada imagen.

- *ALTOLADRILLO* y *ANCHOLADRILLO*: establecen las dimensiones de un ladrillo, que serán fijas e independientes del tipo de ladrillo. Estos valores son de utilidad cuando hay que comunicar al otro extremo la posición de un determinado ladrillo (al cargar los ladrillos en pantalla, el esclavo debe invertir

sus coordenadas) y a la hora de resolver una colisión con una pelota o con un proyectil.

- *IRROMPIBLE*: esta constante proporciona un valor de referencia para el número de golpes cuando se trate de un ladrillo irrompible. Es decir, si el ladrillo es irrompible, su parámetro de número de golpes tomará este valor y se mantendrá inamovible durante la partida.

Además de las constantes, la clase también define una serie de atributos privados que serán analizados a continuación, junto con los correspondientes métodos de acceso y modificación:

- **idLadrillo**: es un entero que identifica unívocamente a cada ladrillo dentro de un mismo mapa. Este identificador se asigna cuando se crea el objeto, a partir del correspondiente parámetro que se pasa por el constructor de la clase. Puede ser consultado mediante el método *getId()*. Se utiliza cuando hay que identificar el ladrillo contra el que ha colisionado una pelota o un proyectil y cuando hay que informar al otro extremo sobre qué ladrillo ha sido el que se ha impactado.

- **imagenLadrillo**: contiene la imagen que representa al ladrillo. Ésta se genera también desde el constructor, a partir del color que se pasa por parámetro. Esta imagen podrá ser consultada a través del método *getImagen()*. Sirve únicamente para la fase de inicialización de cada mapa, donde hay que cargar los ladrillos para representarlos por pantalla. En la figura 4.46 puede verse una recopilación de las imágenes que generarán los *sprites* para representar los ladrillos.

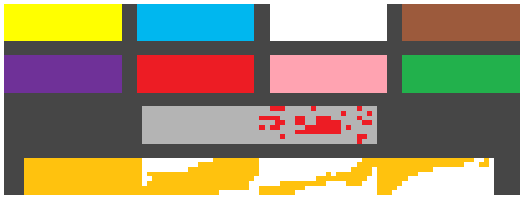


Figura 4.46. Imágenes de los ladrillos

Los primeros ocho ladrillos son los ladrillos simples, eliminables tras un único impacto. Debajo está la imagen que se usa para el *sprite* del ladrillo gris, que necesita dos impactos para destruirlo. La imagen es una unión de los dos aspectos que presenta este tipo de ladrillo: cuando todavía no ha sido impactado y cuando ya ha recibido un golpe. Finalmente, en la parte inferior de la figura se muestra la imagen para el *sprite* del ladrillo irrompible. Ésta es una concatenación de los sucesivos estados por los que pasa el ladrillo cuando ha sido golpeado. Esta sucesión de imágenes es la que provoca el efecto de 'brillo' durante la partida.

- **ladrilloX**, **ladrilloY**: son las coordenadas del ladrillo, nuevamente referidas a la esquina superior izquierda del *sprite* que lo representa. Sirven para ubicar en pantalla a cada ladrillo y para localizar al ladrillo en cuestión cuando ha sido impactado. Sus valores se establecen desde el constructor, cuando se crea el objeto, aunque también es posible modificarlos usando el método *setCoordenadas(int ladrilloX, int ladrilloY)*. –al invertir en el esclavo las coordenadas de los ladrillos con respecto al maestro–. Los métodos *getCoordenadaX()*, *getCoordenadaY()*, por su parte, sirven para recuperar sus respectivos valores.

- **numGolpes**: almacena el número de golpes que son necesarios para eliminar un ladrillo de la pantalla. En el videojuego desarrollado, puede ser un golpe, dos o ninguno porque el ladrillo sea irrompible. Este valor se fija desde el constructor, una vez se ha identificado qué tipo de ladrillo es. El conocimiento de este valor permitirá determinar qué acción realizar sobre el ladrillo recién impactado (eliminarlo si era rompible tras un golpe, cambiar su aspecto si eran necesarios dos golpes o hacerlo brillar brevemente si era un ladrillo irrompible). Se puede consultar su valor utilizando el método *getNumGolpes()* y modificarlo con el correspondiente *setNumGolpes(int nGolpes)*.

- **powerUpId**: indica el tipo de *power-up* asociado a la eventual cápsula que se desprendería una vez fuese destruido el ladrillo. Por defecto, no se define ningún

power-up para cada ladrillo (la mayoría de ladrillos rompibles no arrojará ninguna cápsula). Sin embargo, aleatoriamente se designarán ciertos *power-ups* a determinados ladrillos que sean rompibles. La asignación correrá a cargo del método *setPowerUp(int powerUpId)*. Por otro lado, su valor puede ser recuperado mediante el método *getPowerUp()*.

Posiblemente, el método más relevante de esta clase sea *colision()*, cuya función principal, como su propio nombre indica, es el de resolver, desde el punto de vista del ladrillo, la colisión de éste con una pelota o con un proyectil. En esencia, lo que hace es reducir en una unidad el número de golpes que son necesarios para eliminar el ladrillo, si éste no es irrompible, y devolver el identificador del posible *power-up* asociado a él cuando la cuenta del número de golpes llegue a cero.

4.6.5. MAPAS

Un mapa es el marco sobre el que discurre la acción de la partida, es decir, el escenario en el que se representan las pelotas y las naves que los jugadores deberán manejar para eliminar los ladrillos del mismo. Cada mapa deberá disponer de una imagen de fondo y una colección de ladrillos. La clase encargada de dar formato y caracterizar a cada uno de ellos es *Mapa.java*. La figura 4.47 muestra el aspecto que presenta su diagrama.

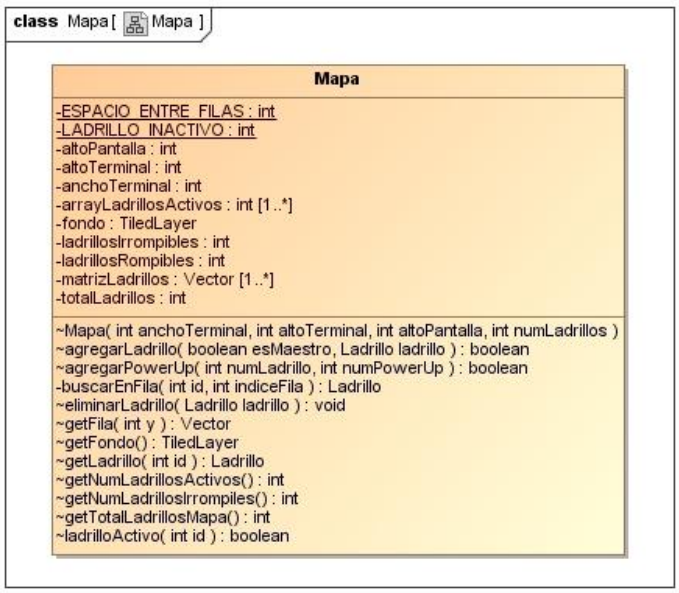


Figura 4.47. Diagrama de la clase Mapa.java

Se observa que dispone de una serie de atributos privados, cuyo significado y utilidad se pasará a analizar a continuación, y una lista de métodos con modificador de acceso *package*, que permitirán modificar y consultar estos atributos:

- *ESPACIO_ENTRE_FILAS*: esta constante determina la distancia vertical, es decir, la diferencia medida en el eje Y, que debe haber entre dos ladrillos que estén colocados uno a continuación del otro en el eje vertical. Es un valor clave para la organización interna de los ladrillos dentro del mapa, como se explicará a continuación.

- *LADRILLO_INACTIVO*: es una constante, de valor simbólico, utilizada para representar en la estructura interna del mapa que un ladrillo ha sido eliminado del mismo.

- **altoPantalla**, **altoTerminal**, **anchoTerminal**: los dos últimos representan las dimensiones de la pantalla del terminal, mientras que el primero determina la altura efectiva sobre la que se va a representar el mapa (podrá ser menor o igual que la del terminal, pero nunca mayor). Los tres valores se establecen desde el constructor, donde son pasados por parámetro. **altoPantalla** servirá para mapear en memoria, junto con *ESPACIO_ENTRE_FILAS*, los ladrillos que componen el mapa. Los atributos que indican las dimensiones de la pantalla del terminal, por su parte, serán utilizados para calcular la posición de un ladrillo en el terminal esclavo, invertida con respecto a la del maestro, y la altura, a su vez, servirá también para determinar dónde agregar o buscar un ladrillo dentro de la estructura interna del mapa.

- **arrayLadrillosActivos**: es una colección de valores enteros que determinan qué ladrillos permanecen todavía en pantalla y cuáles han sido ya eliminados. Cada posición representa el estado de un ladrillo, el cual es indexado en el *array* por su propio identificador. Si está en pantalla –es decir, está activo–, el contenido del *array* en la posición correspondiente indica en qué fila de la matriz de ladrillos (parámetro que se comentará más adelante) se encuentra. En caso contrario, tomará el valor de la constante *LADRILLO_INACTIVO*. El tamaño de

este *array* viene dado por el número total de ladrillos que hay inicialmente en el mapa –parámetro que se proporciona en el constructor de la clase–.

- **fondo**: se define de la misma manera que se hizo cuando se explicó la clase *GUICanvas.java*. El motivo por el que aquí se vuelve a definir es porque la clase *Mapa.java* no hereda de ella (no es una pantalla de menú) y se necesita como imagen que constituya, en la clase principal, el fondo del videojuego para un mapa determinado. Al igual que en aquella ocasión, se puede acceder al contenido de este atributo a través del método *getFondo()*.

- **ladrillosIrrompibles**, **ladrillosRompibles**, **totalLadrillos**: los dos primeros indican la cantidad de ladrillos irrompibles y rompibles, respectivamente, que hay en el mapa, mientras que el tercero proporciona el número total de ladrillos que componen el mapa original, con independencia de si son rompibles, irrompibles, presentes o no en pantalla. A medida que se agregan los ladrillos cuando se cargan inicialmente en el mapa, estas cantidades se van incrementando según corresponda. El número de ladrillos irrompibles y el total de ladrillos presentes en el mapa no cambian durante el desarrollo de la partida sobre un mismo mapa. Sin embargo, el número de ladrillos rompibles disminuye en número a medida que se van eliminando, de manera que cada vez que se elimina un ladrillo por impacto de una pelota o de un proyectil, este atributo disminuye su valor en una unidad. La cantidad de ladrillos irrompibles puede consultarse usando el método *getNumLadrillosIrrompibles()*, mientras que la cantidad total de ladrillos (rompibles e irrompibles) disponibles en el mapa en cada momento vendrá dada por el método *getNumLadrillos()*. Así, la condición para pasar al siguiente mapa, será que ambos métodos devuelvan el mismo valor (no queden ladrillos rompibles en el mapa). El número total de ladrillos puede consultarse a través del método *getTotalLadrillosMapa()*. Conocer su valor será útil a la hora de cargar los ladrillos por pantalla al inicio de cada nivel.

- **matrizLadrillos**: es un vector bidimensional, de tantas filas como indique el cociente entre el alto de la pantalla efectiva sobre la que se va representar el mapa y la constante *ESPACIO_ENTRE_FILAS*, y tantas columnas como

ladrillos haya en cada una de esas filas, según su distribución en el mapa. La figura 4.48 ilustra gráficamente la estructura de esta matriz bidimensional.

Nº fila				
0	ladrillo 0			
1				
2	ladrillo 1	ladrillo 2		
3	ladrillo 3	ladrillo 4	ladrillo 5	ladrillo 6
4	ladrillo 7	ladrillo 8	ladrillo 9	
…				

Figura 4.48. Estructura de la matriz de ladrillos

Cada ladrillo se introduce en la matriz según su coordenada vertical en el mapa. El cociente entre ésta y la constante *ESPACIO_ENTRE_FILAS* determinará la fila de la matriz en la que se agrega dicho ladrillo. La estructura de una matriz bidimensional agiliza el proceso de búsqueda de un ladrillo dentro del mapa, pues ahora no hay que buscar entre todos los ladrillos presentes en el mismo (como ocurriría en el caso peor si no estuvieran organizados según su posición), sino solo entre aquellos que compartan posición en el eje vertical, que, por lo general, serán muchos menos. En realidad, un ladrillo situado en una posición *(x,y)* podría admitirse en la misma fila que otro cuyas coordenadas fueran *(x,y+1)*. Sin embargo, si la coordenada vertical fuera diferente, todos los ladrillos que componen los mapas preparados para este videojuego distan entre sí, en ese eje, una cantidad igual o superior que la dada por *ESPACIO_ENTRE_FILAS*, por lo que dos ladrillos con diferente coordenada vertical irán siempre a filas diferentes.

Al cargar el mapa, los ladrillos se van agregando uno a uno, mediante el método *agregarLadrillo(boolean esMaestro, Ladrillo ladrillo)*. El hecho de que sea el terminal maestro o no, determinará si las coordenadas del ladrillo se deben invertir o no, respectivamente. Una vez se introduce en la matriz de ladrillos, se indica en **arrayLadrillosActivos** en qué fila se puede encontrar. Finalmente, se incrementa el número de ladrillos del tipo al que pertenezca (rompibles o irrompibles). No obstante, como se comentará en la siguiente sección, cuando se carguen en pantalla los ladrillos, puede ocurrir que, por cuestiones de espacio, algunos de ellos no puedan ser representados en las pantallas de los dispositivos, de manera que al final se muestran por pantalla menos ladrillos de los que componían inicialmente el mapa. Aquellos que

no puedan ser representados, serán eliminados de la matriz de ladrillos y señalados como inactivos en el *array* de ladrillos activos.

Posteriormente, cada ladrillo puede ser recuperado invocando a *getLadrillo(int id)*. Para ello, se realiza una búsqueda con el método privado *buscarEnFila(int id, int indiceFila)* en la fila indicada por **arrayLadrillosActivos**. Obtener un ladrillo servirá para extraer la información necesaria para representarlo por pantalla con el *sprite* que corresponda al tipo de ladrillo, para consultar cualquier información relativa a él o para añadirle un *power-up* cuando sea preciso hacerlo. También, para facilitar el estudio de las posibles colisiones, se pueden obtener todos los ladrillos de una misma fila, sin más que indicando en *getFila(int y)* la coordenada Y ligada a la fila de interés.

Por otra parte, para agregar correctamente un *power-up* a un ladrillo determinado, la clase *Mapa.java* proporciona el método *agregarPowerUp(int numLadrillo, int numPowerUp)*. El motivo por el que se recurre a este método en lugar de agregar directamente el *power-up* sobre el ladrillo –mediante la invocación de *setPowerUp(int powerUpId)* desde la instancia del propio ladrillo– es porque así se puede comprobar antes si el ladrillo al que se quiere asociar el *power-up* está activo dentro de la estructura del mapa (si no lo estuviera, ya no estaría presente en él, por lo que no sería posible completar este proceso). Aun en el inicio de la partida, es posible que haya ladrillos que no quepan en la pantalla para jugar la partida, por lo que son eliminados de la estructura del mapa. Para saber si un determinado ladrillo sigue activo, es decir, está representado en pantalla, se puede recurrir al método *ladrilloActivo(int id)*.

Los ladrillos podrán ser eliminados del mapa a través del método *eliminarLadrillo(Ladrillo ladrillo)*, siempre que se trate de un ladrillo rompible y haya sido impactado el número de veces necesario para eliminarlo. Una vez se ha eliminado el ladrillo en cuestión, se suprime de la matriz de ladrillos, se señala como inactivo en el *array* de ladrillos activos y se disminuye en una unidad el número de ladrillos rompibles presentes en el mapa.

4.6.6. GESTOR DE MAPAS

Para administrar correctamente la ejecución ordenada de cada mapa del videojuego, la carga en pantalla de éstos y las transiciones de un mapa a otro cuando

termina uno y se empieza el siguiente, es preciso diseñar una clase que realice estas funciones. Dicha clase se llama *GestorDeMapas.java*, cuyo diagrama es el que se muestra a continuación, en la figura 4.49.

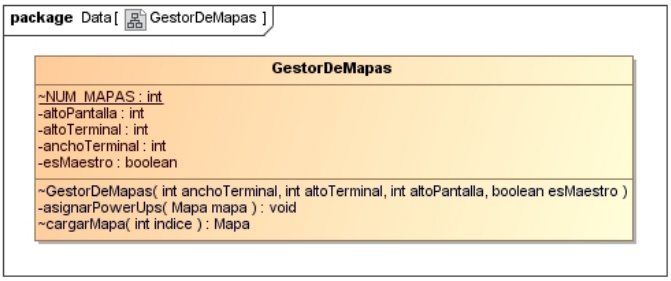


Figura 4.49. Diagrama de la clase GestorDeMapas.java

Se observa que es una clase que define un pequeño grupo de atributos y dos métodos que se encargarán de realizar la funcionalidad requerida para esta clase:

- NUM_MAPAS: El número de mapas que componen el videojuego, en concreto diez, viene determinado por esta constante, que servirá a su vez para controlar cuándo hay que empezar de nuevo por el primer mapa en caso de que la partida continúe tras superar el último nivel.

- **altoPantalla**, **altoTerminal** y **anchoTerminal**: la utilidad de estos parámetros es la misma que la comentada anteriormente en la clase *Mapa.java*. Los dos últimos, además, servirán para ubicar correctamente en pantalla los ladrillos de cada mapa cuando éstos son cargados desde el método *cargarMapa(int índice)*, en caso de que sea necesario corregir sus posiciones para que aparezcan centrados en la pantalla del terminal.

- **esMaestro**: servirá para distinguir si se trata del terminal maestro o no, en cuyo caso tendrá la responsabilidad de designar, mediante el método *asignarPowerUps(Mapa mapa)*, qué ladrillos arrojarán una cápsula con un *power-up* cuando éstos sean eliminados. Para favorecer la jugabilidad del videojuego y para impedir que se memoricen en qué ladrillos se encuentran ciertos tipos de *power-ups*, esta asignación se realiza mediante un procedimiento aleatorio, tanto para elegir los *power-ups* como los ladrillos que los albergan.

4.7. LÓGICA DEL VIDEOJUEGO

La lógica que determina el funcionamiento del videojuego viene definida en la clase *Juego.java*, cuyo diagrama es el que se muestra en la figura 4.50.

4.7.1. HERENCIA E INTERFAZ IMPLEMENTADA

Como ya se adelantó en la sección de las estructuras multihilo, la clase *Juego.java* es una subclase de *GameCanvas*, por lo que para poder disponer de capacidad *multithread* y así poder ejecutar el bucle principal del juego en un hilo aparte, es preciso implementar la interfaz *Runnable*. Así, al igual que en *Maestro.java* y *Esclavo.java*, en el código aparecerá el método *run()* en el que se incluirán las instrucciones que deben ser ejecutadas en un hilo aparte.

GameCanvas, por su parte, es una subclase de *Canvas* que permite examinar el estado de las teclas del dispositivo cuando el jugador ejecuta una acción, además de crear animaciones rápidas y libres de parpadeo cuando haya *sprites* en movimiento. Estas características, naturalmente serán de utilidad para la clase que implementa la lógica del videojuego. Los mecanismos que emplea para que los procesos de pintado y gestión de eventos de teclado tengan un nivel de transparencia mayor para el programador, son:

- La posibilidad de acceder directamente al objeto *Graphics* y de disponer de un *buffer* específico en el que se podrán representar y tratar las imágenes antes de mostrarlas en la pantalla del dispositivo.
- La técnica de *polling* (encuesta). Permite acceder directamente al estado de las teclas del dispositivo.

4.7.2. ATRIBUTOS DE LA CLASE PRINCIPAL

La clase principal define un importante número de atributos, de los cuales un grupo de ellos son constantes. En función de su utilidad para el código pueden agruparse en pequeños conjuntos de constantes que serán analizados a continuación:

Figura 4.50. Diagrama de la clase Juego.java

- Directorios o *paths*: estas constantes almacenan los directorios de algunas imágenes que se muestran por pantalla durante la partida, concretamente las relacionadas con las paredes laterales, las naves que representan las vidas que les quedan a cada jugador y las pelotas controladas por cada uno de ellos, identificadas con colores diferentes. La tabla 4.22 recopila estos identificadores.

Path	Imagen a la que permite acceder
PATH_NAVEVIDA	Nave que representa una vida
PATH_PARED1	Tramo de pared más grande
PATH_PARED2	Tramo de pared más pequeño
PATH_PELOTA1	Pelota controlada por el jugador (blanca)
PATH_PELOTA2	Pelota controlada por el otro jugador (amarilla)

Tabla 4.22. Identificadores de los directorios de imágenes

- Dimensiones de imágenes: establecen el tamaño de las imágenes obtenidas a partir de los directorios anteriormente descritos, salvo las pelotas cuyo diámetro es igual para todas y se define en la propia clase *Pelota.java*. Los identificadores que especifican estos valores se detallan en la tabla 4.23.

Identificador	Descripción
ALTOPARED	Altura del tramo de pared más grande
ALTOVIDA	Altura de la nave que representa una vida
ANCHOPARED	Anchura del tramo de pared más grande
ANCHOVIDA	Anchura de la nave que representa una vida

Tabla 4.23. Identificadores de dimensiones de imágenes

Las dimensiones de las paredes servirán para determinar en qué punto deben rebotar las pelotas cuando colisionan contra ellas y hasta dónde puede moverse la nave controlada por el jugador sin llegar a salirse del espacio ocupado por la pantalla de juego. En cada extremo lateral de la pantalla, la pared se compone de dos fragmentos de diferente tamaño, de manera que la condición de rebote se formaliza en base a la condición más restrictiva (en este caso, la determina la anchura más grande).

Por otra parte, las dimensiones de las naves de menor tamaño, que simbolizan el número de vidas disponibles que le queda a cada jugador, se deberán tener en cuenta a la hora de ubicar a cada una de ellas, una a continuación de la otra, y también para situar correctamente las naves de los jugadores en ambos terminales.

- Temporizadores o *timers*: durante el transcurso de la partida, hay ciertas acciones que se necesitan ejecutar periódicamente. Éstas requerirán un valor de referencia que indique cada cuántos ciclos de ejecución del bucle principal del juego se deben repetir o se permite que sean repetidas, según sea el caso. La tabla 4.24 recoge los identificadores de estos *timers* y añade una breve descripción.

Identificador	Descripción
TIMER_SHOT	Número mínimo de ciclos del bucle entre disparos consecutivos
TIMER_SPEED	Ciclos para incrementar automáticamente la velocidad de la pelota
TIMER_THROW	Ciclos para lanzar la pelota de forma automática

Tabla 4.24. Temporizadores implementados

- *COOPERATIVE_MODE*, *DEATHMATCH_MODE*: los identificadores que sirven para distinguir el modo de juego de la partida están definidos en esta clase. Consisten en un carácter que identificará al modo cooperativo y al modo *deathmatch*, respectivamente. Necesitan ser públicos, para poder indicar correctamente el modo de juego seleccionado desde los menús previos al inicio de la partida y para referirse al modo de juego del que se desea consultar las puntuaciones registradas.

- Otras constantes: finalmente, se definen una serie de constantes de diversa naturaleza que serán de utilidad para la presente clase. La tabla 4.25 resume los identificadores de estas constantes y adjunta una breve descripción explicativa de las mismas.

Identificador	Descripción
MAX_PELOTAS	Número máximo de pelotas permitidas simultáneamente
MAX_VIDAS	Número máximo de vidas permitidas a cada jugador
OFFSET_PELOTA	Desplazamiento de la pelota con respecto a la nave cuando está adherida
PUNTOS	Puntuación que se añade al jugador por cada ladrillo rompible eliminado
RETARDO	Retraso introducido en cada ejecución del bucle principal

Tabla 4.25. Otras constantes definidas

El número máximo de pelotas que puede haber activas al mismo tiempo durante la partida, viene determinado por *MAX_PELOTAS*. Su valor es igual a 6,

considerando la posibilidad de que ambos jugadores puedan tener activado a la vez el *power-up* que triplica el número de pelotas.

Por otro lado, el número máximo de vidas que puede llegar a acumular un mismo jugador se fija a 5 y viene establecido por la constante *MAX_VIDAS*.

OFFSET_PELOTA, por su parte, indica la posición relativa, con respecto a la nave, donde se debe ubicar la pelota cuando está adherida a ésta. Este *offset* se define sobre el eje horizontal.

Como ya se comentó en el capítulo 3, al eliminar un ladrillo rompible tras un impacto, se suman 100 puntos para el jugador que lo ha eliminado. Esta cifra se recoge en la constante *PUNTOS*. Si el ladrillo era rompible tras dos impactos, la cantidad a sumar se duplica.

Por último, es de mencionar el retardo de 20mseg introducido en cada ejecución del bucle principal, para dar tiempo a refrescar la imagen en pantalla y a recoger los eventos de teclado. Esta cantidad se indica en la constante *RETARDO*.

Además de todas estas constantes, la clase define a su vez una variada lista de atributos que serán de utilidad a la hora de implementar la funcionalidad requerida. Estos atributos podrán agruparse en conjuntos de atributos para agilizar su explicación y justificación dentro de la clase:

- Contadores: estos atributos se encargarán de almacenar la cuenta del número de veces que se ejecuta el bucle principal del videojuego en cada mapa o establecerán el ciclo de ejecución en concreto en el que se debe aplicar una determinada función programada. Estos últimos estarán íntimamente ligados a los *timers* definidos con anterioridad. La tabla 4.26 recoge los identificadores de estos contadores, a los que les acompaña una breve descripción.

Identificador	Descripción
contDisparo	Ciclo a partir del cual se pueden volver a disparar proyectiles
contIncVelocidad	Ciclo en el que se incrementará automáticamente la velocidad de la pelota
contLanzamiento	Ciclo en el que se lanzará la pelota de forma automática
contador	Número de veces que se ha ejecutado el bucle principal en un mismo mapa

Tabla 4.26. Contadores implementados

Según se observa en la tabla, además del contador que indica cuántos ciclos del bucle principal se completan, se definen otros tres contadores adicionales:

contDisparo almacena el valor de **contador** en el momento del disparo de los proyectiles (se asume que el jugador tiene habilitado el *power-up* de disparo) más el número de ciclos indicado por *TIMER_SHOT*, de manera que cuando se efectúa un disparo, el jugador no podrá volver a disparar antes de que se ejecuten *TIMER_SHOT* ciclos del bucle principal; por otra parte, **contIncVelocidad** indica el ciclo en el que se incrementará la velocidad de la pelota de forma automática, por la permanencia de ésta en la pantalla sin dejar de estar activa en la partida. Se inicializa al número de ciclos indicado por *TIMER_SPEED*, de manera que si **contador** alcanza ese valor antes de que la pelota se haya perdido, ésta incrementa su velocidad (si hubiera más pelotas en ese momento, por activación del *power-up THREEBALLS*, todas incrementarían su velocidad). Posteriormente, su contenido se actualizará al indicado por **contador**, al que se le añade la cantidad almacenada en *TIMER_SPEED* para programar el nuevo instante en el que se deberá incrementar la velocidad de la pelota. Esta actualización se produce también cuando se activa un *power-up* que incide sobre la velocidad de la pelota, o pelotas que tenga a disposición el jugador en ese momento y, naturalmente, cuando inicia una nueva vida; finalmente, **contLanzamiento** almacena el ciclo de ejecución en el que se debería lanzar la pelota del jugador al inicio de cada vida, en caso de que éste no la haya lanzado manualmente con anterioridad.

- Identificadores de componentes del juego: definen los atributos relacionados con cualquier elemento que forma parte del videojuego. La tabla 4.27 recopila estos identificadores y añade un breve comentario describiendo su significado.

Identificador	Descripción
gestorDeMapas	Administra y carga cada mapa del videojuego
mapaActual	Mapa extraído de *gestorDeMapas* sobre el que se desarrolla la partida
numMapa	Índice del mapa sobre el que se desarrolla la partida (*mapaActual*)
naveLocal	Nave controlada por el jugador
naveRemota	Nave controlada por el otro jugador
numPelotasLocales	Número de pelotas activas controladas por el jugador en cada instante
numPelotasRemotas	Número de pelotas activas controladas por el otro jugador en cada instante
numVidasLocal	Número de vidas disponibles para el jugador
numVidasRemoto	Número de vidas disponibles para el otro jugador
pelotas	*Array* que contiene todas las pelotas, activas o no, de los dos jugadores
puntuacionLocal	Puntos acumulados por el jugador
puntuacionRemota	Puntos acumulados por el otro jugador

Tabla 4.27. Identificadores de componentes del juego

- Identificadores de elementos gráficos y sonoros: son los atributos encargados de constituir el apartado visual y sonoro del videojuego. Almacenan los *sprites* que representan las imágenes de cada elemento de la partida y permiten gestionar y administrar la ejecución de cada uno de los sonidos que se precisan durante la misma. La tabla 4.28 muestra cada uno de estos identificadores junto con una breve descripción explicativa.

Identificador	Descripción
gestorCapas	Almacena los *sprites* propios de cada mapa
gestorSonidos	Administra y controla los efectos de sonido del videojuego
num_items	Número fijo de elementos gráficos que se muestran por pantalla
spriteNaveLocal	*Sprite* que representa la nave controlada por el jugador
spriteNaveRemota	*Sprite* que representa la nave controlada por el otro jugador
spritePelotas	*Array* de *sprites* que representan las pelotas del videojuego

Tabla 4.28. Elementos gráficos y sonoros

El atributo **gestorCapas** es un objeto de la clase *LayerManager*. A él se añaden los *sprites* que representan los objetos propios de cada mapa, es decir, las imágenes de las paredes, pelotas, naves controladas por los jugadores, naves que representan las vidas disponibles, ladrillos del mapa, el fondo de la aplicación y las eventuales cápsulas que se desprendan de algunos ladrillos tras ser eliminados.

Los atributos **spriteNaveLocal**, **spriteNaveRemota** y **spritePelotas** son también elementos que se agregan a este gestor de capas, pero se definen como atributos aparte para poder manipularlos con mayor facilidad, ya que van a ser los elementos en movimiento que van a controlar ambos jugadores.

Por otro lado, el valor de **num_items** indicará el número de elementos fijos que se representan por pantalla, con independencia del mapa que se trate: pelotas, naves controladas por los jugadores, naves que representan las vidas disponibles y los tramos de pared que conforman las dos paredes laterales. Conocer este valor será de utilidad a la hora de manejar las imágenes de los ladrillos insertados en **gestorCapas**, pues irán a continuación de estos elementos y se insertarán en orden según su identificador. Así, para localizar a la imagen de cada ladrillo dentro del gestor, habrá que sumar la cantidad indicada por **num_items** al identificador de dicho ladrillo y el resultado será el índice donde se encuentra.

Por último, **gestorSonidos** no es más que una copia del gestor de sonidos definido desde el *MIDlet* de la aplicación, que para no tener que invocarlo cada vez que se necesita ejecutar o comprobar si se está ejecutando un sonido, se pasa por parámetro en el constructor para copiarlo en este atributo.

- **offsetX**, **offsetY**: indican el desplazamiento necesario para representar correctamente cada elemento en la pantalla de cada dispositivo.

- Atributos relacionados con los *power-ups*: definen una serie de booleanos y vectores vinculados al tratamiento de los *power-ups* declarados en el videojuego. La tabla 4.29 recopila estos identificadores y añade una breve descripción a cada uno de ellos.

Identificador	Descripción
activarPowerUpRemoto	*Power-up* que se debe activar en el otro jugador
arrayPowerUpsLocal	*Power-ups* que tiene activados el jugador
arrayPowerUpsRemoto	*Power-ups* que tiene activados el otro jugador
droppinPowerUps	Vector que almacena las cápsulas con *power-ups* desprendidas
leerListaPowerUps	Booleano que indica la lectura de la lista de *power-ups* del mapa
listPowerUps	Lista de *power-ups* asociados a ladrillos para cada mapa

Tabla 4.29. Atributos relacionados con los power-ups

Cuando se recibe del otro extremo de la comunicación un mensaje para activar un *power-up* para el otro jugador, el atributo **activarPowerUpRemoto** recoge dicho *power-up* para su posterior activación. Para indicar, si lo hubiera, qué *power-up* se encuentra activado en cada momento en uno u otro jugador, se utilizan los *arrays* **arrayPowerUpsLocal** y **arrayPowerUpsRemoto**. El vector **droppinPowerUps**, por su parte, almacena las cápsulas desprendidas de los ladrillos eliminados a los que estaban asociados. Éstas, como ya se ha comentado, son insertadas también en **gestorCapas**, pero después de todas las imágenes y el fondo de la aplicación, por lo que será necesario recuperar estos *sprites* para indicar expresamente que se representen por pantalla. Esto es así porque en un objeto *LayerManager*, la última posición corresponderá a la capa más profunda a la hora de ser representadas por pantalla. Finalmente, si se trata del terminal esclavo, el booleano **leerListaPowerUps** indicará, cuando se active, que se ha recibido del terminal maestro la lista con los ladrillos a los que van

asociados ciertos *power-ups*, por lo que deberá actualizar esta información a partir de **listPowerUps**, que es la cadena de caracteres que se recibe en el mensaje.

- Vectores de posiciones de efectos de *power-ups*: indican las coordenadas de las imágenes que son producto de algunos *power-ups* disponibles en el videojuego. La tabla 4.30 resume sus identificadores y añade una pequeña descripción.

Identificador	Descripción
posicionDisparos	Almacena las coordenadas y movimiento de cada proyectil
posicionesNoBounceLocal	Almacena las coordenadas de la estela verde del *power-up NOBOUNCE*
posicionesNoBounceRemoto	Almacena las coordenadas de la estela verde del *power-up NOBOUNCE*

Tabla 4.30. Vectores de posiciones de efectos de power-ups

Cabe destacar que las imágenes generadas por los correspondientes *power-ups* (el de disparo para **posicionDisparos** y el de no rebote para **posicionesNoBounceLocal** y **posicionesNoBounceRemoto**) no se almacenan en **gestorCapas**, por lo que a la hora de representarlas por pantalla habrá que consultar, además del contenido de **gestorCapas** y **droppinPowerUps**, si estos vectores contienen elementos o por el contrario están vacíos. Se ha preferido no incluirlos en dicho gestor para no complicar el mecanismo para acceder tanto a los ladrillos como a las cápsulas de *power-ups* añadidas al final del mismo.

- Vectores asociados a ladrillos: se distinguen dos vectores de este tipo: **ladrillosIrrompiblesGolpeados** y **ladrillosRemotos**. El primero, como su propio nombre indica, contiene aquellos ladrillos irrompibles presentes en el mapa que han sido recientemente golpeados. Esta información es relevante en tanto en cuanto se necesita para saber si hay que hacerlos 'brillar' o no, como consecuencia de un eventual impacto por parte de una pelota o de un proyectil. El segundo, por su parte, contiene la información relativa a un ladrillo que ha sido impactado en el terminal remoto antes que en el terminal que maneja el propio jugador. Esta información no es otra que la que se deduce de los mensajes de tipo *BRICK* recibidos.

- Atributos relativos al tipo de partida y al rol de cada extremo: la clase principal del videojuego necesita saber qué modo de juego se ha seleccionado y qué rol asume el terminal sobre el que se está ejecutando el código. Para ello, se definen una serie de atributos, que se resumen en la tabla 4.31.

Identificador	Descripción
esMaestro	Booleano que indica si es el terminal maestro o no
esclavo	Instancia de la clase *Esclavo.java*
maestro	Instancia de la clase *Maestro.java*
tipoPartida	Indica el modo de juego seleccionado

Tabla 4.31. Atributos relativos al tipo de partida y al rol de cada extremo

El modo de juego indicado por **tipoPartida**, determinará algunas acciones que se han de tomar durante el transcurso de la partida, como el hecho de saber si el rebote de la pelota del otro jugador sobre la nave propia resta una vida al jugador que maneja la nave o no, o si la puntuación final es la suma de las puntuaciones obtenidas por cada jugador o cuenta solo la alcanzada por uno de ellos. Por otra parte, el rol que se adjudica a cada terminal se realiza desde el constructor de la clase. Si el objeto que se pasa por parámetro es una instancia de *Maestro.java*, el atributo que se utilizará en ese terminal será **maestro**, poniendo a *true* **esMaestro**; en otro caso, se utilizaría **esclavo**, poniendo consecuentemente a *false* **esMaestro**.

- Otros atributos: finalmente, queda un pequeño grupo de atributos booleanos, resumidos en la tabla 4.32.

Identificador	Descripción
lanzarRemota	Habilita el lanzamiento de la pelota del otro jugador desde su nave
partidaActiva	Indica si la partida está en curso
quitarVida	Señala cuándo hay que restar una vida al otro jugador (modo *deathmatch*)
reubicarRemota	Permite volver a situar en su nave la pelota del otro jugador

Tabla 4.32. Otros atributos definidos

El atributo **lanzarRemota** indica cuándo hay que lanzar la pelota desde la nave controlada por el otro jugador, independientemente de si ha sido lanzada manualmente o de forma automática. Al iniciar la partida, **partidaActiva** se pone a *true* para indicar que está en curso, y seguirá así hasta que se indique que ésta termina, bien por agotamiento de las vidas de alguno de los jugadores o por

abandono de uno de ellos. En una partida en modo *deathmatch*, el atributo **quitarVida** indicará que hay que restarle una vida al otro jugador porque una pelota que no es suya ha rebotado sobre su nave, toda vez que se haya recibido el correspondiente mensaje de tipo *BOUNCEKILLING*. Finalmente, **reubicarRemota** se activará cuando se reciba un mensaje de tipo *RESTART*, como consecuencia de haber perdido la última pelota del otro jugador y servirá para volver a situarla adherida en la nave de éste.

4.7.3. FUNCIONAMIENTO DE LA CLASE PRINCIPAL: MÉTODOS

Como ya se indicó en la sección de Bluetooth, cuando se inicia la primera partida, los dos extremos de la comunicación envían un mensaje de tipo *START*, incluyendo la información correspondiente a las dimensiones de la pantalla de cada terminal. Al recibir ese mensaje, es cuando cada extremo crea el objeto de la clase *Juego.java* para poder iniciar la partida posteriormente. Al crear dicho objeto, además de inicializar algunos atributos de la clase, se determina el rol del terminal en la comunicación, estudiando el tipo de instancia del parámetro *Object o* incluido en el constructor, se especifica el valor de los atributos **offsetX** y **offsetY** en función de las dimensiones de la pantalla sobre la que se va a desarrollar la partida y se copia en **gestorSonidos** el gestor de sonidos previamente definido desde el *MIDlet* de la aplicación. Además, se invoca a los métodos que se muestran en la figura 4.51.

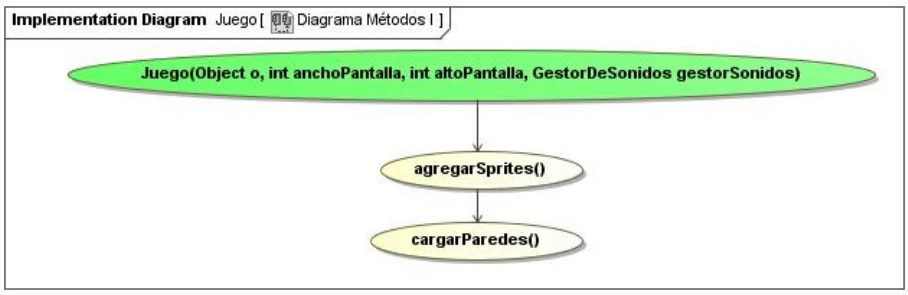


Figura 4.51. Diagrama de métodos invocados desde el constructor de la clase

El método *agregarSprites()* cargará en el gestor de capas los elementos comunes a todos los mapas que integran el videojuego: las paredes, las pelotas, las naves y las vidas de cada jugador. Para añadir las paredes llamará al método *cargarParedes()* que se encargará de realizar esta operación, determinando los fragmentos de pared que son necesarios en función de las dimensiones de la pantalla.

Es de notar el color verde con que se ha remarcado el constructor de la clase. En general, el color verde será indicativo del carácter público del modificador de acceso del método así coloreado. En este caso, se justifica que el constructor sea público por ser invocado éste desde las clases *Maestro.java* y *Esclavo.java*, pertenecientes ambas al paquete *bluetooth*.

Una vez el objeto de la clase *Juego.java* ha sido creado en cada terminal, se puede dar comienzo a la partida. Esto se hace a través del método *jugar(char idPartida)*. Este método determinará, a partir del carácter que se le pasa por parámetro, el modo de juego que adoptará la partida, además de inicializar el número de vidas de cada jugador al correspondiente a cada inicio de partida (es decir, tres vidas), reiniciar las puntuaciones de cada uno de ellos y establecer en **numMapa** el índice del primer mapa para ser cargado a continuación. Asimismo, llamará a los métodos indicados en la figura 4.52 para completar satisfactoriamente el inicio de una nueva partida, no sin antes indicarlo habilitando el atributo **partidaActiva**.

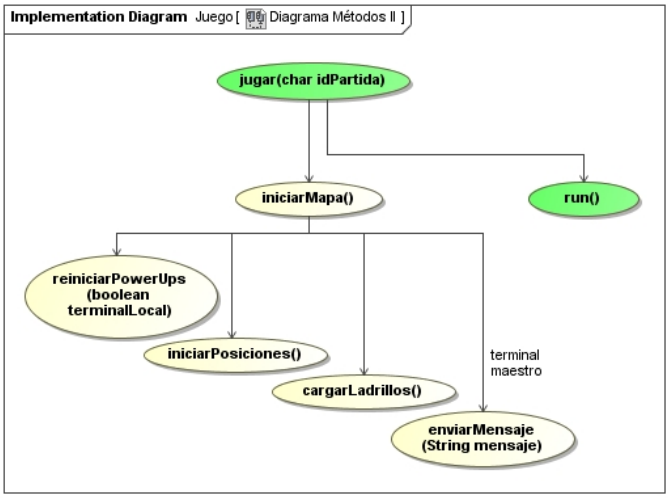


Figura 4.52. Diagrama de métodos invocados desde jugar

Puede observarse que el método *jugar(char idPartida)* es público, ya que éste es invocado nuevamente desde las clases *Maestro.java* y *Esclavo.java*. El método *run()*, por su parte es público por definición.

Mediante la llamada a *iniciarMapa()*, además de inicializar algunos atributos más, se reinician los *power-ups* que pudieran estar activados en cada jugador (no debe haber activado ninguno al inicio de cada mapa) llamando al método *reiniciarPowerUps(boolean terminalLocal)*; a través de *iniciarPosiciones()*, se inicializan las posiciones de las naves y las pelotas, estas últimas adheridas sobre sus respectivas naves; se carga en memoria el correspondiente mapa y mediante *cargarLadrillos()*, se añaden al gestor de capas los *sprites* que representan a los ladrillos de dicho mapa; por último, el terminal maestro deberá determinar qué ladrillos contendrán *power-ups*, y qué *power-ups* serán estos, y enviárselos mediante *enviarMensaje(String mensaje)* al terminal esclavo para que ambos compartan la misma información. Cabe destacar que, para garantizar el funcionamiento en todos los terminales, se cargarán solo los ladrillos que estén dentro de los límites indicados por las dimensiones de la pantalla sobre la que se va a desarrollar la partida.

Finalmente, se crea un nuevo hilo para ejecutar el bucle principal de la partida. Los métodos invocados desde el método *run()* se muestran en la figura 4.53.

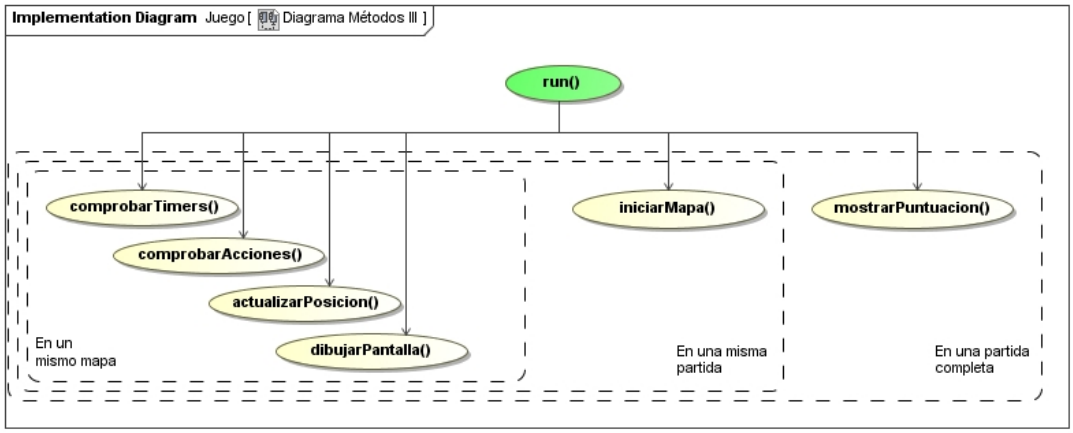


Figura 4.53. Diagrama de métodos invocados desde run

En él se puede observar una serie de funciones que deben realizarse en cada ciclo de ejecución del bucle principal mientras la partida discurre sobre el mismo mapa:

- Desde *comprobarTimers()* se controla si se deben ejecutar una serie de actividades programadas para ese momento o si necesitan resolver ciertos eventos registrados tras recibir un mensaje del terminal remoto.

- A partir de *comprobarAcciones()*, se gestionan los eventos de teclado, actualizando la posición de la nave si el jugador ha efectuado un desplazamiento hacia la izquierda o hacia la derecha, o ha pulsado el botón de disparo para lanzar la pelota desde la nave o para lanzar proyectiles si tiene activado el correspondiente *power-up*.
- Mediante *actualizarPosicion()*, se renuevan las coordenadas de las pelotas, de las posibles cápsulas con *power-ups* desprendidas de sus correspondientes ladrillos a los que estaban asociados y de los proyectiles que eventualmente se hayan lanzado.
- Por último, *dibujarPantalla()* será el encargado de representar por pantalla todos los elementos presentes en el mapa.

Estos métodos son el corazón de la lógica del videojuego y serán analizados con mayor profundidad en las siguientes páginas. Una vez se han completado, se introduce el retardo de 20mseg mencionado al considerar las constantes definidas en la clase, para refrescar la imagen en pantalla y recoger los próximos eventos de teclado, y se incrementa el contador del sistema.

Cuando se completa un mapa, mediante la verificación de que, si quedan ladrillos, todos son irrompibles, se procederá a la carga del siguiente mapa del videojuego. Para ello, se incrementa **numMapa** y se carga el mapa correspondiente volviendo a llamar al ya mencionado *iniciarMapa()*, que esta vez, además, eliminará del gestor de capas todos aquellos elementos residuales que hayan quedado del mapa recién completado (ladrillos irrompibles y eventuales cápsulas de *power-ups*). En el caso de que se hubiera superado el último mapa del videojuego, se reiniciaría dicho atributo para volver a cargar el primer mapa.

Finalmente, al concluirse la partida por agotamiento de las vidas de al menos uno de los dos jugadores, se ejecuta *mostrarPuntuacion()* para determinar, en función del modo de juego y de la puntuación obtenida por ambos jugadores, la pantalla de menú a mostrar en cada terminal.

Como ya se ha comentado, el eje principal del código gira en torno a las funciones que se ejecutan en cada iteración del bucle mientras la partida transcurre en el mismo mapa. La primera de ellas era *comprobarTimers()*, uno de los métodos más

importantes dentro del código, cuyo diagrama de métodos a los que invoca es representado de manera esquemática en la figura 4.54. En él se pueden observar las diferentes tareas que ha de realizar, las cuales serán analizadas a continuación:

Este método se encargará del lanzamiento automático de la pelota controlada por el jugador en el caso de que **contador** haya alcanzado el valor indicado por **contLanzamiento** y la pelota no haya sido lanzada de forma manual. Dicha operación la realizará llamando al método *lanzarPelotaLocal()*, que a su vez enviará el correspondiente mensaje al extremo remoto avisándole de dicho lanzamiento.

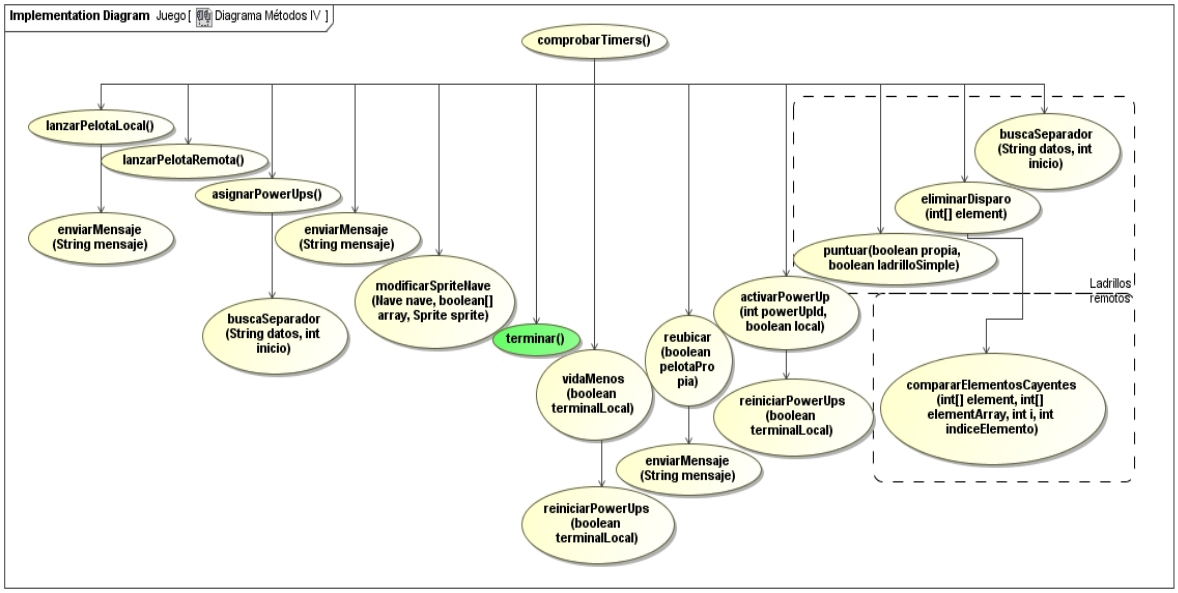


Figura 4.54. Diagrama de métodos invocados desde comprobarTimers

Asimismo, la pelota del otro jugador será lanzada una vez se haya recibido el mensaje de tipo *THROW*, en cuyo caso se habilitaría la ejecución del método *lanzarPelotaRemota()*.

Además, si es el terminal esclavo, tendrá que actualizar la información sobre qué ladrillos tienen asociado un *power-up* y cuáles son estos *power-ups*, una vez le haya llegado el correspondiente mensaje de tipo *LISTPOWERUP*, enviado por el terminal maestro, y almacenado la información adjunta en **listPowerUps**. De esta tarea se encarga el método *asignarPowerUps()*. Para decodificar correctamente la información almacenada en **listPowerUps**, necesitará la ayuda de *buscaSeparador(String datos, int*

inicio). La utilización de este último método servirá, en lo sucesivo, para decodificar correctamente los campos contenidos en una cadena de caracteres en condiciones similares.

Por otra parte, será necesario enviar un mensaje al otro terminal cuando se detecte que **contador** ha alcanzado el valor indicado por **contIncVelocidad**, para avisarle de que la pelota controlada por él ha aumentado de velocidad. Esto lo hará invocando al método *enviarMensaje(String mensaje)*. También habrá que enviar un mensaje cuando el jugador se quede sin vidas, en cuyo caso se ejecutará también el método *terminar()* –definido como público porque, como ya se señaló en la sección de Bluetooth, necesita ser invocado también desde *Maestro.java* y *Esclavo.java* para finalizar la partida–, como señal de que la partida ha terminado. Este método se ejecutará igualmente si el que se ha quedado sin vidas ha sido el otro jugador. Al hilo de este argumento, desde *comprobarTimers()* se restan vidas sólo al otro jugador, tras haber recibido el correspondiente mensaje de tipo *BOUNCEKILLING*. En tal caso, se ejecuta el método *vidaMenos(boolean terminalLocal)*, que a su vez reiniciará mediante *reiniciarPowerUps()* los *power-ups* que pudiera tener activos dicho jugador. Y también, si es necesario reubicar en pantalla una pelota, es decir, volver a situarla sobre la nave para iniciar una nueva vida, se realiza desde *comprobarTimers()* mediante una llamada al método *reubicar(boolean pelotaPropia)*. Si la pelota reubicada ha sido la del propio jugador, se le enviará un mensaje de tipo *RESTART* al otro extremo para que tenga conocimiento de ello.

Por lo que a la activación de *power-ups* respecta, desde *comprobarTimers()* se activarán en el otro jugador los *power-ups* que se reciban de los mensajes de tipo *POWERUP*. Esto lo realizará ejecutando el método *activarPowerUp(int powerUpId, boolean local)*. Además, previo a la activación de todo *power-up*, este método invocará a *reiniciarPowerUps(boolean terminalLocal)* para desactivar cualquier otro que pudiera disponer el jugador (aunque manteniendo en juego las pelotas que controlaba antes de ser activado el nuevo *power-up*, si se da el caso de que previamente había activado el que triplicaba el número de pelotas, y manteniendo el mismo número de vidas disponibles, si había recogido un *power-up* de vida extra). Además, si alguna nave está en proceso de modificación de su anchura por haber activado recientemente un *power-up*, mediante *modificarSpriteNave(Nave nave, boolean[] array, Sprite sprite)* se

sustituirá progresivamente el *sprite* que representa a dicha nave por el *sprite* inmediatamente superior o inferior hasta que alcance el nuevo tamaño que le corresponde, para provocar el efecto visual de ensanchamiento o encogimiento, respectivamente.

Por último, si algún ladrillo irrompible está 'brillando' como consecuencia de haber sido impactado recientemente, desde *comprobarTimers()* se realiza la sucesión de imágenes, a través de los *frames* del *sprite*, que dan lugar al efecto de 'brillo', hasta que deje de brillar (vuelve de nuevo al primer *frame*). Esta operación no requiere invocar a ningún método. Sin embargo, para resolver los posibles impactos sobre ladrillos que se hayan producido en el terminal remoto antes que en el propio, sí será necesario recurrir a algunos de ellos. En particular, si el impacto ha sido provocado por un proyectil, hay que eliminarlo de la pantalla. Para ello, tras haber decodificado correctamente la información relativa a dicho proyectil, se procede a su eliminación mediante *eliminarDisparo(int[] element)*. Éste, a su vez, necesitará de la ayuda de *compararElementosCayentes(int[] element, int[] elementArray, int i, int indiceElemento)* para identificar cuál es el disparo que finalmente impacta sobre el ladrillo. En cualquier caso, si el ladrillo ha sido destruido, se comprueba si contenía algún *power-up* y se puntúa debidamente, con la ayuda del método *puntuar(boolean propia, boolean ladrilloSimple)*; si era el primer impacto para un ladrillo rompible tras dos impactos, se cambia la imagen del ladrillo por la del ladrillo golpeado; y si era un ladrillo irrompible se inicia la sucesión de *frames* que simula el efecto de brillo. Para obtener correctamente la información sobre el ladrillo impactado, se usa de nuevo *buscaSeparador(String datos, int inicio)*.

Otra de las funciones requeridas en cada iteración del bucle principal del programa, es la que se encarga de gestionar las pulsaciones de teclado. Tarea que se realiza a través de *comprobarAcciones()*, según se ha comentado. En la figura 4.55 se representa el diagrama con los métodos que son invocados desde él.

Básicamente, necesitará enviar un mensaje al otro terminal cada vez que el jugador desplace la nave hacia un lado o hacia el otro, para indicarle la nueva posición que adopta la misma, y para informarle, cuando proceda, de que la pelota la ha lanzado

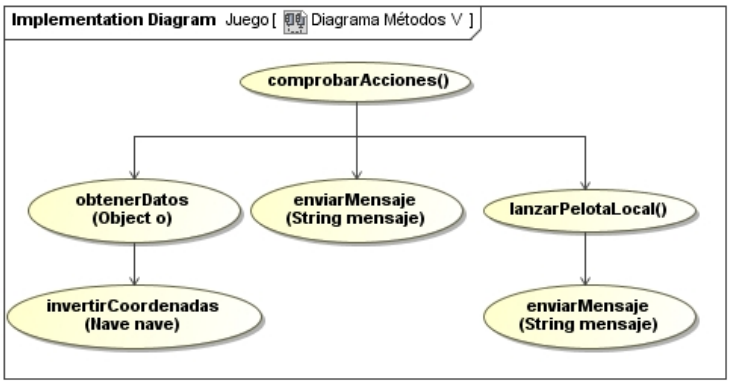


Figura 4.55. Diagrama de métodos invocados desde comprobarAcciones

manualmente tras pulsar el botón de disparo, o que ha lanzado dos proyectiles, si tiene activado el *power-up* de disparo.

Cuando mueva la nave, tendrá que adjuntarle la información relativa a la posición de ésta. Para ello, utilizará el método *obtenerDatos(Object o)* que, en función del tipo que sea la instancia del parámetro *o*, pasará a obtener las coordenadas invertidas de una pelota o de una nave. En este caso, naturalmente, invertirá las coordenadas de la nave en cuestión, mediante el método *invertirCoordenadas(Nave nave)*. Además, deberá comprobar si la pelota está adherida a la nave, en cuyo caso deberá "seguir" el movimiento aplicado a ésta.

Igualmente, si con la pelota ligada a la nave, el jugador pulsa el botón de disparo, se invoca al método *lanzarPelotaLocal()* para resolver el evento de teclado y se envía, como ya se ha dicho, un mensaje al otro terminal para notificar el lanzamiento.

Finalmente, si se pulsa el botón de disparo y se tiene activado el *power-up* de disparo, se generará el correspondiente par de proyectiles, se insertarán en **posicionDisparos** y se notificará al otro extremo con un mensaje de tipo *SHOT* que dos nuevos proyectiles han sido lanzados con las coordenadas especificadas.

La tercera de las funciones implementadas que se ejecutan en el bucle principal del videojuego se encarga de refrescar en pantalla la posición de los objetos en movimiento, de acuerdo con sus coordenadas. La figura 4.56 representa el esquema de los métodos a los que llama para completar su función.

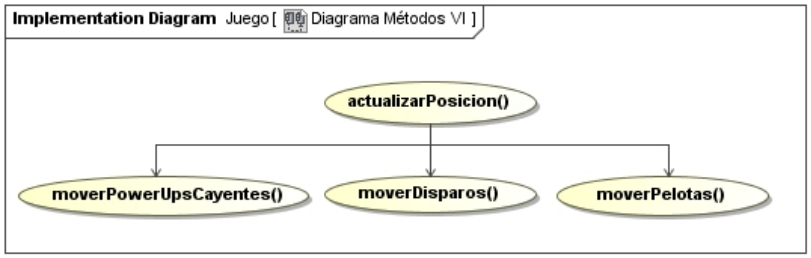


Figura 4.56. Diagrama de métodos invocados desde actualizarPosicion

Cada uno de estos métodos se encargará de la actualización de la posición de un tipo de elemento diferente. En particular, *moverPowerUpsCayentes()* se encargará de desplazar, si las hubiera, aquellas cápsulas desprendidas de sus ladrillos una vez han sido eliminados. Igualmente, si hubiese proyectiles lanzados, éstos se desplazarán según las indicaciones dadas desde *moverDisparos()*. Las pelotas activas de la partida, por su parte, se desplazarán mediante el método *moverPelotas()*. El contenido y funcionamiento de estos tres métodos se desglosará y se analizará a partir de los siguientes diagramas, que darán una visión más detallada de la implementación de cada uno de ellos.

En primer lugar está el método *moverPowerUpsCayentes()*, cuyo diagrama de métodos a los que invoca se expone en la figura 4.57.

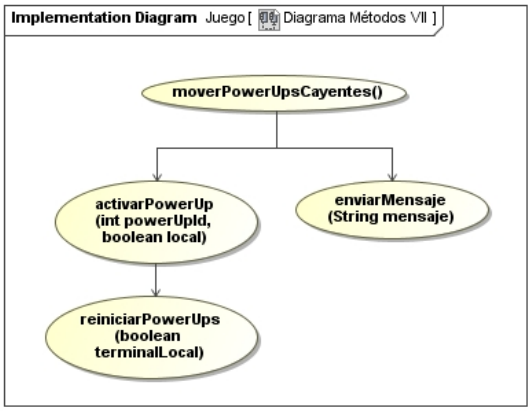


Figura 4.57. Diagrama de métodos invocados desde moverPowerUpsCayentes

Este método comprueba si hay algún elemento dentro del vector **droppinPowerUps**. De ser así, estos elementos son, como ya se indicó cuando se

describieron los atributos de la clase, *sprites* que representan las cápsulas de los *power-ups*. Para cada uno de estos *sprites*, fija el siguiente *frame* dentro de la secuencia de *frames* que lo compone para generar el efecto óptico de rotación de la cápsula, comentado cuando se explicó la implementación de los *power-ups* y desplaza la cápsula según su sentido de movimiento sobre el eje vertical. Posteriormente, se comprueba si ha excedido los límites de la pantalla, en cuyo caso se elimina del vector. Si, por el contrario, la cápsula ha sido recogida por la nave del extremo hacia el que se dirigía, se activa el correspondiente *power-up*, nuevamente mediante *activarPowerUp(int powerUpId, boolean local)*, y desactivando con *reiniciarPowerUps(boolean terminalLocal)*, de la misma forma que ya se comentó anteriormente, los posibles *power-ups* que pudieran estar activados. Además, se envía el mensaje de tipo *POWERUP* al otro extremo para informarle de la activación de dicho *power-up*.

El siguiente método que es invocado desde *actualizarPosicion()* es *moverDisparos()*. La figura 4.58 muestra el esquema de los métodos a los que se invoca desde éste.

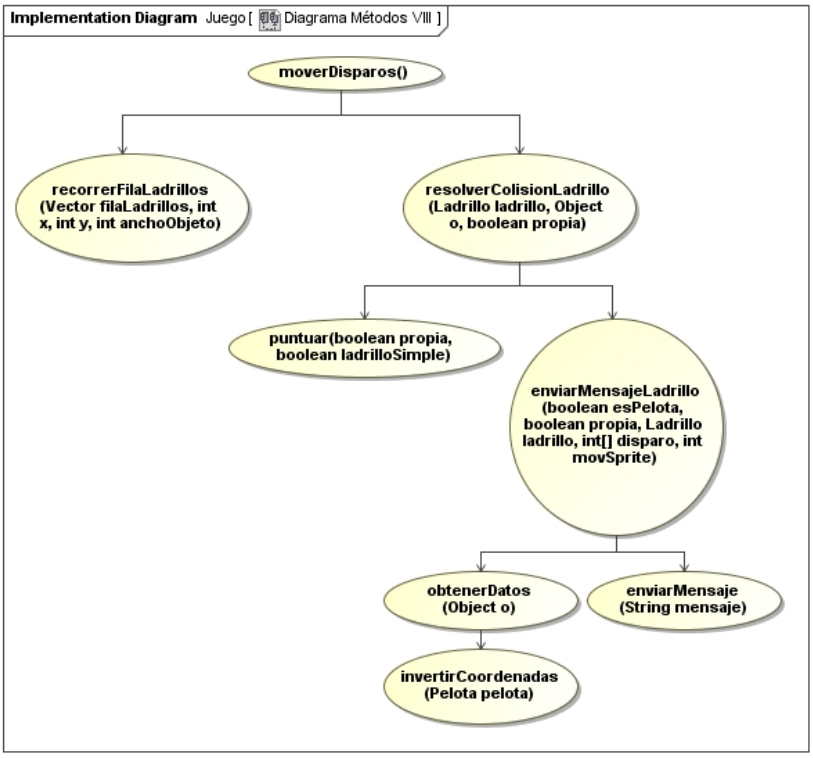


Figura 4.58. Diagrama de métodos invocados desde moverDisparos

De la misma manera que antes, este método comprueba si hay algún elemento dentro del vector **posicionDisparos**. De ser así, a cada disparo almacenado se le modificará su posición en función del sentido hacia el que se desplaza sobre el eje vertical y se comprobará si colisiona con algún ladrillo. Para averiguar si efectivamente hay alguna colisión, se llama al método *recorrerFilaLadrillos(Vector filaLadrillos, int x, int y, int anchoObjeto)* que, a partir de una fila de ladrillos situada a una determinada altura, comprobará si el objeto en cuestión (en este caso, un proyectil) colisiona con algún ladrillo de esa fila, en función de las coordenadas sobre la que se sitúe dicho objeto. En caso de existir tal colisión, el método devolverá el ladrillo implicado, que podrá ser pasado por parámetro en *resolverColisionLadrillo(Ladrillo ladrillo, Object o, boolean propia)* para solventarla. Este método, pues, será el encargado de invocar al método *puntuar(boolean propia, boolean ladrilloSimple)*, en caso de que tras la colisión se haya eliminado un ladrillo rompible del mapa y de generar y añadir al gestor de capas la correspondiente cápsula si dicho ladrillo llevaba asociado un *power-up*. Si fuera un ladrillo irrompible, o un ladrillo rompible tras dos impactos al cual se ha impactado por primera vez, simplemente se añadiría, en el primer caso, al vector **ladrillosIrrompiblesGolpeados** para iniciar la sucesión de *frames* que dan lugar al efecto de 'brillo' y que posteriormente se controla desde *comprobarTimers()*, como ya se ha comentado. En el segundo caso, bastaría con cambiar la imagen del ladrillo, por la del ladrillo 'roto' tras el primer impacto de los dos necesarios. Por último, independientemente del tipo de ladrillo que fuera, se informa al otro extremo de tal colisión, mediante el método *enviarMensajeLadrillo(boolean esPelota, boolean propia, Ladrillo ladrillo, int[] disparo, int movSprite)*, el cual envía un mensaje de tipo *BRICK*, recabando la información que necesite a partir de los parámetros que se le pasa y llamando a *obtenerDatos(Object o)*. Una vez preparada la información del mensaje, se hace efectivo su envío mediante *enviarMensaje(String mensaje)*.

Finalmente, el tercer método al que se llamaba desde *actualizarPosicion()* es *moverPelotas()*. En la figura 4.59 se observa el diagrama que resume los métodos a los que se invoca desde él.

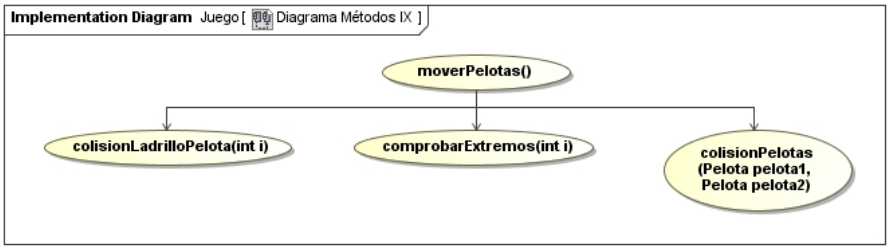


Figura 4.59. Diagrama de métodos invocados desde moverPelotas

Este método es el más complejo de los tres, pues además de realizar el propio desplazamiento de las mismas, tendrá que comprobar las posibles colisiones con ladrillos y el consiguiente rebote, si no tiene activado el *power-up* de no rebote, además de las que pueda haber con otras pelotas y comprobar si exceden los límites de la pantalla por alguno de sus extremos o rebotan sobre una nave. Cada una de estas actividades están implementadas por los métodos que son invocados desde *moverPelotas()* y que serán analizados cada uno a continuación.

El primero de ellos es *colisionLadrilloPelota(int i),* cuyo diagrama de métodos es el que se muestra en la figura 4.60.

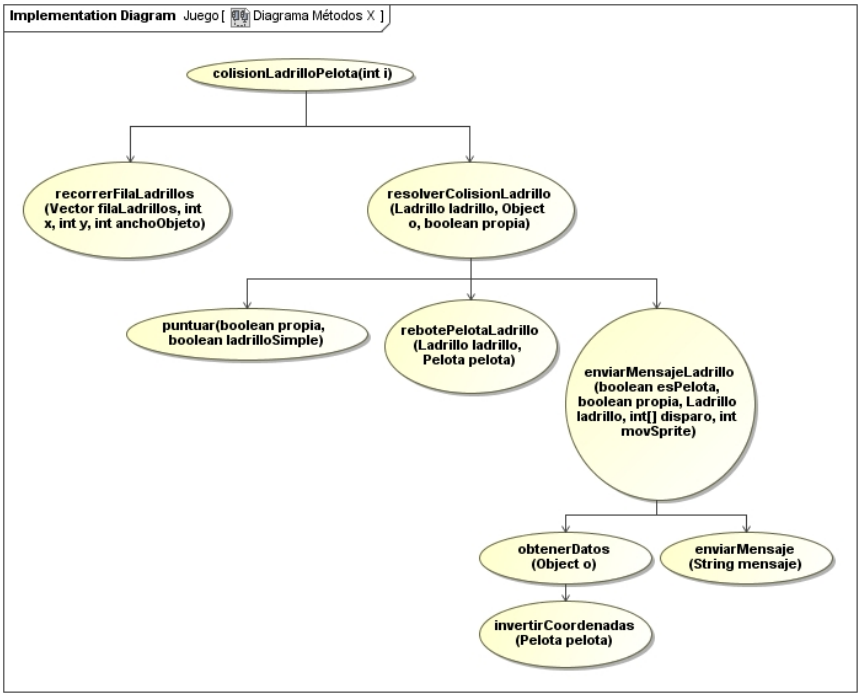


Figura 4.60. Diagrama de métodos invocados desde colisionLadrilloPelota

Para la pelota identificada por el entero *i* que se pasa por parámetro, que representa el índice dentro del vector **pelotas**, se comprobará, de nuevo, mediante *recorrerFilaLadrillos(Vector filaLadrillos, int x, int y, int anchoObjeto)*, si ésta colisiona con algún ladrillo situado a la misma altura a la que se encuentra. Aquí no ocurre como con los disparos, cuyo movimiento era siempre unidimensional; las pelotas, por el contrario, se pueden desplazar sobre ambos ejes de referencia, lo que podría dar lugar a una colisión lateral con un ladrillo. Si no fuera así, dependiendo de hacia dónde se mueva la pelota, se comprobará también si colisiona con algún ladrillo de la correspondiente fila, de la misma manera que se hacía con los disparos. Si finalmente se produce una colisión con algún ladrillo, está se volverá a solucionar desde el método *resolverColisionLadrillo(Ladrillo ladrillo, Object o, boolean propia)*, solo que ahora el parámetro de tipo *Object* será una pelota en lugar de un proyectil, y por tanto, además de realizar las consiguientes tareas de tratamiento del ladrillo después del impacto, adición al gestor de capas de una eventual cápsula, puntuación si procede, y envío del mensaje resultante de tipo *BRICK* al otro extremo, deberá también gestionar el rebote de la pelota tras golpear el ladrillo, si no se tiene activado el *power-up* de no rebote. Esto lo hace a través del método *rebotePelotaLadrillo(Ladrillo ladrillo, Pelota pelota)*, que en función de por dónde golpee al ladrillo, se le asignará a la pelota el nuevo sentido de movimiento y coordenadas.

Además de explorar una posible colisión con un ladrillo, tras desplazar una pelota según su sentido de movimiento, es preciso comprobar si ésta se ha salido de los límites de la pantalla (o ha rebotado al llegar al otro extremo por ser una partida en modo *deathmatch*) o si una nave la ha hecho rebotar para que permanezca en ella. Efectuar estas verificaciones corre por cuenta del método *comprobarExtremos(int i)*, donde *i* vuelve a identificar a la i-ésima pelota del vector **pelotas**. Su diagrama de métodos a los que invoca se expone en la figura 4.61.

En ella se pueden reconocer tres zonas distintas, dos de ellas delimitadas por rectángulos con un trazo discontinuo. Cada una representa una función que se puede diferenciar de las demás. Así, la primera de ellas se centra en resolver una posible colisión entre una pelota y la nave del extremo al que se aproxima; la segunda en enviar un mensaje al otro terminal; y la tercera en determinar qué sucede cuando una pelota

finalmente se pierde. Estas tareas serán de utilidad para que el método cumpla con su funcionalidad, la cual se explicará con mayor detalle a continuación.

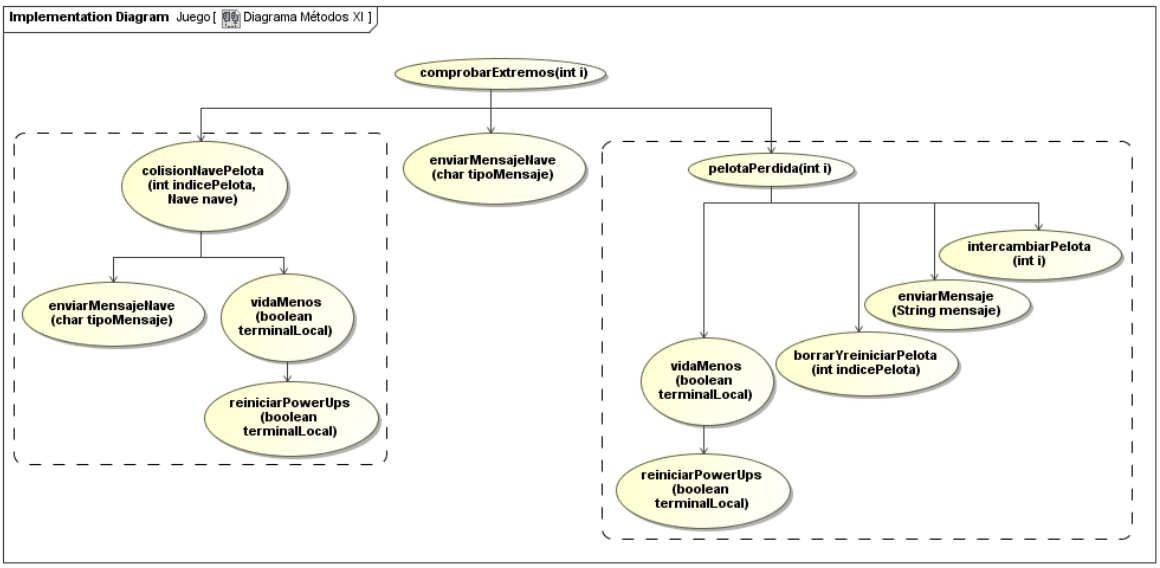


Figura 4.61. Diagrama de métodos invocados desde comprobarExtremos

En esencia, este método se encarga de comprobar, para la pelota indicada, si se encuentra próxima a los límites de la pantalla y tomar las medidas oportunas para cada caso. En concreto, si la pelota va a salirse por algún extremo lateral, debe rebotar al llegar a la pared. Tras el rebote, se envía el correspondiente mensaje de tipo *BALLS* al otro extremo, a modo de sincronización entre ambos terminales.

Por otra parte, si su desplazamiento sobre el eje vertical es hacia abajo y se aproxima al extremo inferior, hay que considerar la posibilidad de que vaya a rebotar sobre la nave de dicho extremo. Tras comprobar si la nave está debajo de la pelota, se llama al método *colisionNavePelota(int indicePelota, Nave nave)* para resolver dicha colisión. En estos casos, conviene determinar sobre qué zona de la nave se produce el rebote. Si éste es lateral, la pelota simplemente rebota sobre el eje horizontal y se informa al otro extremo del rebote producido enviando un mensaje de tipo *BALLSHIP*. Para enviar este mensaje, se invoca a *enviarMensajeNave(char tipoMensaje)*, el cual reunirá la información necesaria con ayuda de los métodos destinados a ese fin (es decir, invocando a *obtenerDatos(Object o)* para que éste se encargue de presentar la información relativa a las pelotas y a la nave involucrada en la colisión correctamente

invertida antes de ser enviada). La figura 4.62 resume el esquema de funcionamiento de este método.

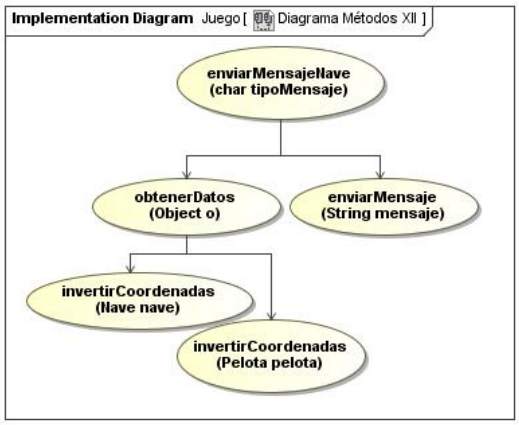


Figura 4.62. Diagrama de métodos invocados desde enviarMensajeNave

Sin embargo, si el rebote es vertical, es preciso considerar el lugar del rebote, pues de ello dependerá qué ángulo se le aplica a la pelota finalmente rebotada, como ya se explicó en la sección anterior. En concreto, en la tabla 4.33 se establece, para cada región, los diferentes valores de movimiento que deberá adquirir la pelota tras colisionar con la nave sobre esa región.

Región	Movimiento tras la colisión
>limites[0]	*(-4,-1)*
limites[0]-limites[1]	*(-3,-2)*
limites[1]-limites[2]	*(-2,-3)*
limites[2]-limites[3]	*(-1,-4)*
limites[3]-limites[4]	*(1,-4)*
limites[4]-limites[5]	*(2,-3)*
limites[5]-limites[6]	*(3,-2)*
<limites[6]	*(4,-1)*

Tabla 4.33. Movimiento de la pelota tras colisionar con la nave

El identificador **limites** hace referencia al atributo del mismo nombre, definido en *Nave.java* y, por tanto, sitúa el límite de las regiones para el ancho de la nave en ese momento. Estos valores de movimiento, obtenidos tras la colisión, deberán ser invertidos en el caso de que se trate de una colisión con la nave del extremo superior de la pantalla.

Si el jugador dispone del *power-up HOLDBALL*, la pelota, en lugar de rebotar, queda adherida a la nave, guardando dicho ángulo de rebote para cuando vuelva a ser

lanzada, bien manualmente, bien de forma automática tras alcanzarse el valor indicado por **contLanzamiento**. Además, enviará al otro terminal el correspondiente mensaje de tipo *BALLSHIP*, de la misma manera que antes. Si la nave no alcanza a hacer rebotar la pelota, ésta se precipita hacia fuera de la pantalla, en cuyo caso se invoca al método *pelotaPerdida(int i)*; no obstante, si se trata de la pelota del otro jugador y es una partida de tipo *deatmatch*, la pelota rebota al llegar a ese extremo y se le envía el consiguiente mensaje *BALLSHIP* al otro terminal.

En general, el envío de mensajes al otro terminal a la hora de resolver una colisión con una nave, un rebote si el modo de juego es *deathmatch* o la pérdida de una pelota, se hace siempre del terminal donde ocurre ese evento en el extremo inferior al otro terminal. Esto es así para garantizar, en caso de que hubiera entre ambos dispositivos un desajuste temporal en el desarrollo de la partida, que la solución que se toma por válida es la misma y es la del extremo inferior. Se ha tomado como referencia lo que ocurre en ese dispositivo (en lugar de utilizar, por ejemplo, el terminal maestro, o de enviar mensajes desde los dos extremos indistintamente) para favorecer la interactividad del videojuego, asegurando que lo que ocurre es lo que ve el jugador al mando del terminal que controla la nave en ese extremo inferior.

En ese mismo sentido, como ya se ha comentado, cuando una pelota llega a perderse por dicho extremo, es cuando se invoca a *pelotaPerdida(int i)*. Este método comprueba si se trataba de la última pelota controlada por el jugador en cuestión (pueden ser más de una, si bien podía haber recogido previamente el *power-up* que triplica el número de pelotas), en cuyo caso debe restarle una vida a dicho jugador –de nuevo, a través de *vidaMenos(boolean terminalLocal)*– y borrar y reiniciar las coordenadas de dicha pelota mediante el método *borrarYreiniciarPelota(int indicePelota)*. En realidad, si le quedaran vidas, se podría reubicar a la pelota encima de la correspondiente nave directamente desde aquí, pero no se hace para dar tiempo a que se ejecute el sonido asociado a la pérdida de una vida, de modo que el último paso, consistente en reubicar a la pelota, se realiza, como ya se ha analizado, desde *comprobarTimers()*, usando como condición que la pelota no esté activa y que el mencionado sonido haya terminado de ejecutarse. Si no fuese la última pelota, habría que comprobar si es necesario un proceso de intercambio de identificadores de éstas

antes de suprimirla definitivamente de la pantalla. En cualquier caso, se envía el correspondiente mensaje de tipo *OUTBALL* al otro terminal.

Si por el contrario, la pelota se desplazase hacia arriba, habría que recurrir de nuevo a *colisonNavePelota(int indicePelota, Nave nave)* si ahora la nave está encima de la pelota que se acerca al extremo superior. De no ser así, habría que considerar otra vez si es una partida *deathmatch* y la pelota es del otro jugador, para decidir si la pelota rebota o no al llegar a ese extremo.

La última de las funciones mencionadas que se deben implementar cuando se aplica el desplazamiento a cada pelota según sus coordenadas de movimiento sobre cada eje, es comprobar si hay colisiones entre ellas mismas. Esto se hace a través del método *colisionPelotas(Pelota pelota1, Pelota pelota2)*. En la figura 4.63 se muestra un diagrama con los métodos que son invocados desde él.

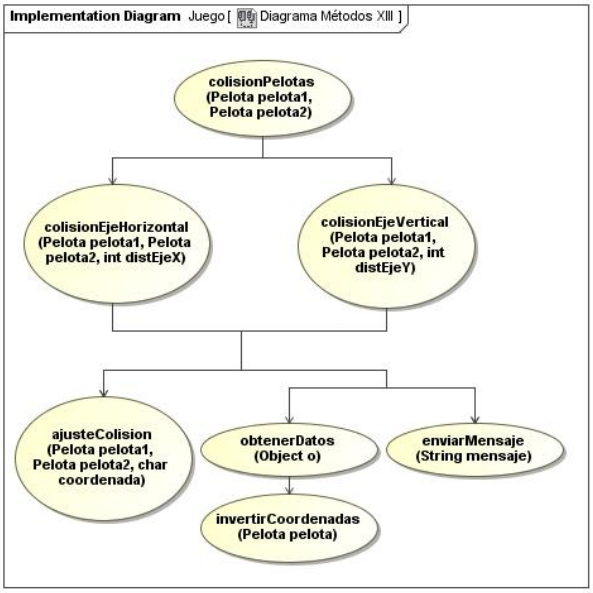


Figura 4.63. Diagrama de métodos invocados desde colisionPelotas

Para determinar si se produce tal colisión, este método estudia sobre qué ejes se ha ocasionado y así poder decidir el sentido del rebote para ambas. En particular, *colisionEjeHorizontal(Pelota pelota1, Pelota pelota2, int distEjeX)* resuelve una posible colisión sobre el eje de abscisas y *colisionEjeVertical(Pelota pelota1, Pelota pelota2, int distEjeY)* hace lo propio sobre el eje de ordenadas. Ambos métodos recurren a

ajusteColision(Pelota pelota1, Pelota pelota2, char coordenada) para adaptar las coordenadas de cada pelota tras la colisión, especificando en *char coordenada* el eje sobre el que se precisa actualizar las posiciones de ambas; y finalmente envían un mensaje de tipo *BALLS* al otro extremo para refrescar la posición de todas las pelotas en el momento de la colisión y así no perder la coherencia en la ejecución de la partida en caso de que hubiera un ligero desfase entre ambos dispositivos.

Todo lo que se ha explicado hasta ahora sobre el funcionamiento del código es el resultado de ejecutar cada ciclo del bucle principal del programa. Sin embargo, como ya se ha puesto de manifiesto en numerosas ocasiones a lo largo de la exposición, es necesario un mecanismo que se encargue de la recepción y tratamiento de los mensajes recibidos desde el otro extremo de la comunicación. El método encomendado a esta labor es, como ha sido ya mencionado en la sección de Bluetooth, *actualizacionRemota(String datos)*, cuyo diagrama de métodos a los que invoca se detalla en la figura 4.64.

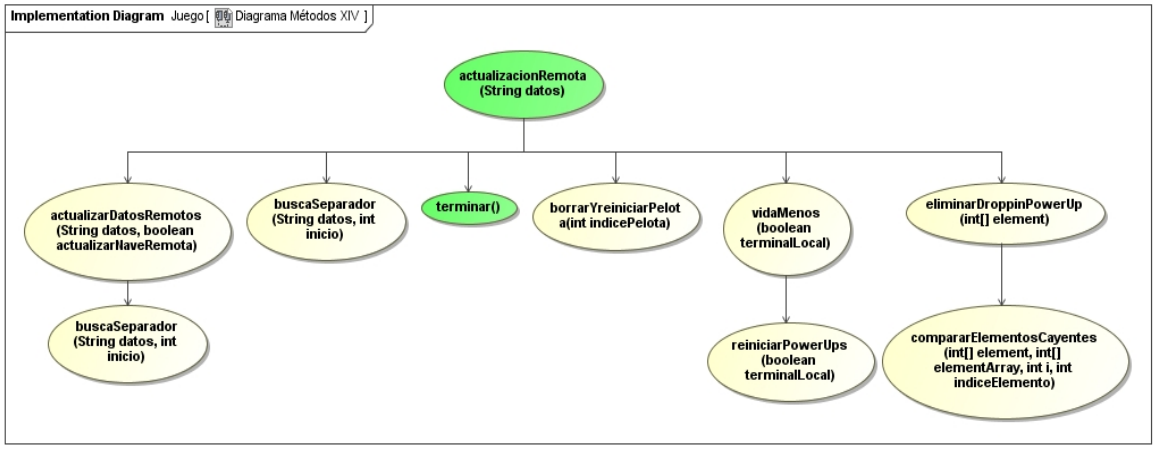


Figura 4.64. Diagrama de métodos invocados desde actualizacionRemota

Al igual que el resto de métodos definidos como públicos en esta clase, *actualizacionRemota* se define como tal porque es invocado desde las clases que definen cada extremo de la comunicación, es decir, *Maestro.java* y *Esclavo.java*, cada vez que se recibe un mensaje de datos.

Para los mensajes en los que viene adjunta información de sincronización de todas las pelotas activas en ese momento (mensajes de tipo *BALLS*, *BALLSHIP* y *BRICK*) se precisará la ayuda de *actualizarDatosRemotos(String datos, boolean*

actualizarNaveRemota) para extraer correctamente la información relativa a la posición de cada pelota y eventualmente la de la nave involucrada en una posible colisión de una pelota con ésta. Adicionalmente, este método devuelve la última posición leída dentro de la cadena de caracteres que se pasa como dato del que extraer la información. Este dato será de utilidad a la hora de indicar en los mensajes de tipo *BRICK* a partir de qué posición, dentro de la cadena, comienza la información relativa al ladrillo y a una posible cápsula de *power-up* asociada a él. El método *buscaSeparador(String datos, int inicio)*, por su parte, servirá, como ya se ha mencionado en anteriores ocasiones, para ayudar a decodificar la información recibida, tanto desde la actualización de la información relativa a pelotas y a una posible nave, como desde el propio método *actualizacionRemota*. Si se recibe un mensaje de tipo *GAMEOVER*, se invoca al método *terminar()*, dado que la partida en curso ha concluido. El método *borrarYreiniciarPelota(int indicePelota)* se utiliza, como es lógico, cuando se pierde una pelota de la pantalla y se recibe el correspondiente mensaje de tipo *OUTBALL*, al igual que *vidaMenos(boolean terminalLocal)* en el caso en que hubiera sido la última pelota activa de ese jugador. Por otra parte, cuando se recibe un mensaje de tipo *POWERUP*, además de activar el correspondiente *power-up* en el otro jugador, hay que eliminar el *sprite* de la cápsula que lo representaba en pantalla. Para ello, se utiliza el método *eliminarDroppinPowerUp(int[] element)*, que a su vez recurre a *compararElementosCayentes(int[] element, int[] elementArray, int i, int indiceElemento)* para eliminar dicho *sprite* de una forma similar a como se eliminaban los proyectiles. Finalmente, la gestión de algunos mensajes recibidos, como *RESTART* o *THROW*, se ha preferido delegarla en *comprobarTimers()*, bien porque correspondía más bien a la ejecución de un temporizador o bien porque resultaba más cómodo almacenar la información en un vector para luego ser analizada desde el bucle de ejecución principal.

4.8. MIDLET

Toda aplicación J2ME necesita tener implementado un *MIDlet*. En el caso de esta aplicación, el *MIDlet* desarrollado presenta el diagrama que se muestra en la figura 4.65. Su nombre, *BArkanoid.java*, es el resultado de unir el nombre del juego original

en el que está inspirado el videojuego implementado (“Arkanoid”) con la inicial de la tecnología que utiliza para dar el soporte multijugador (“Bluetooth”).

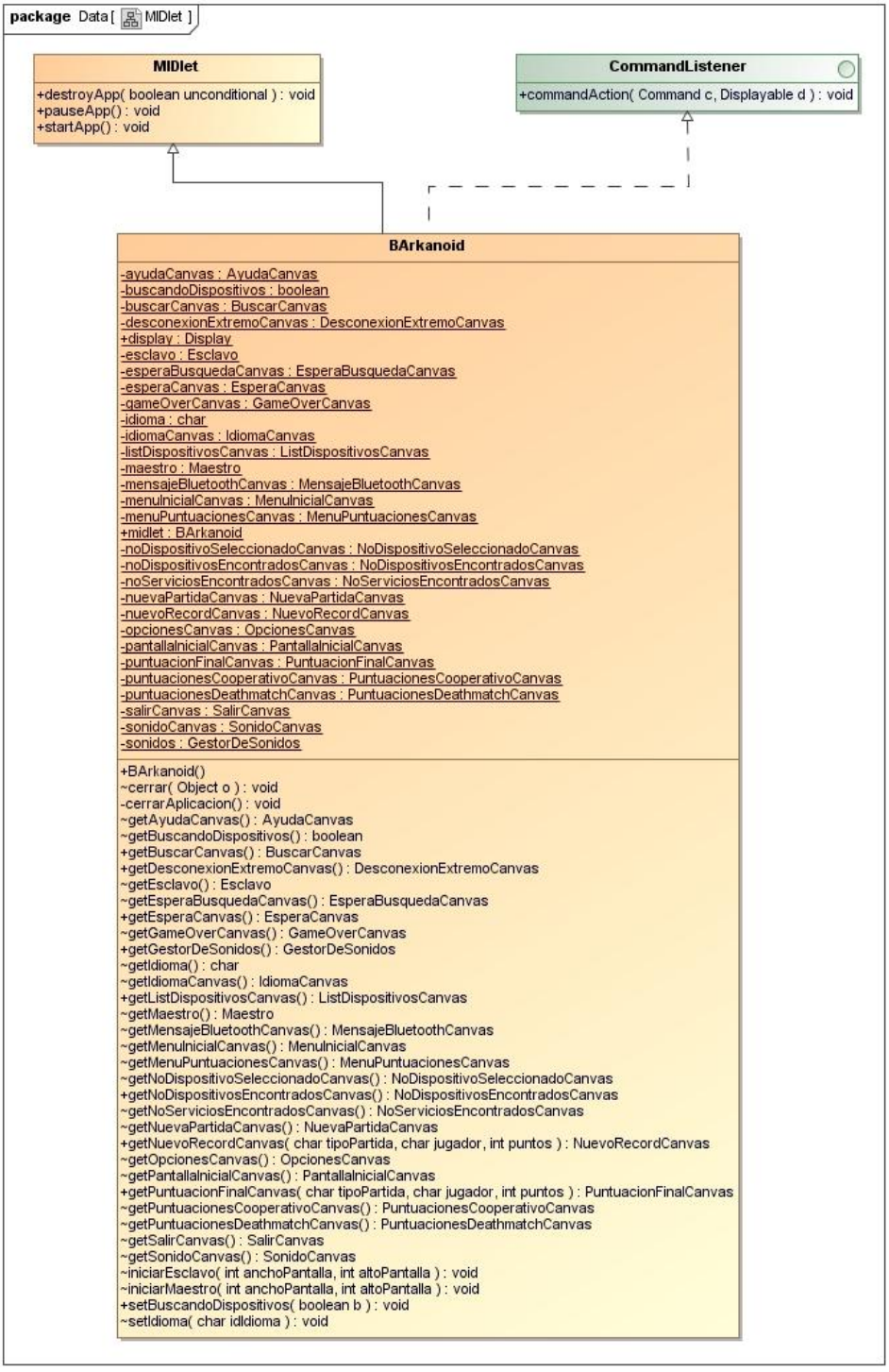


Figura 4.65. Diagrama de la clase BArkanoid.java

Como cualquier *MIDlet*, extiende la clase *javax.microedition.midlet.MIDlet*. Dado que se trata de una clase abstracta, ésta contiene una serie de métodos abstractos

que, al heredar de ella, deberán ser implementados en el *MIDlet*. Estos métodos son los encargados de controlar el ciclo de vida del *MIDlet*, descrito ya en la sección 2.3.2.3. e ilustrado en la figura 2.5:

- *startApp()*: como ya se comentó, el *MIDlet* empieza en estado *Detenido*, se carga en memoria, se ejecuta el constructor y, si no ha habido ningún problema en el constructor, pasa a estado *Activo*, ejecutando este método.

- *pauseApp()*: en caso de que se requiera detener la ejecución del *MIDlet* (por ejemplo, debido a que se reciba una llamada en el móvil y el dispositivo necesite utilizar la pantalla y el teclado), éste pasará al estado *Detenido* mediante la llamada a este método. Para reanudar la ejecución del *MIDlet*, se podrá hacer invocando nuevamente el método *startApp()*.

- *destroyApp(boolean unconditional)*: cuando el entorno de ejecución o el usuario deciden cerrar la aplicación, se hace a través de este método, forzando al *MIDlet* a entrar en el estado *Destruido*, por lo que la aplicación libera los recursos usados y guarda los datos persistentes. Este método se ejecutará cuando se cierre definitivamente la aplicación desde el menú de salida, implementado por **salirCanvas**. Desde él se llamará al método *cerrar()*, que será el encargado de cerrar el extremo de la aplicación y el propio *MIDlet*, una vez se haya ejecutado *cerrarAplicacion()*, invocado desde este mismo método.

Además, el *MIDlet* desarrollado para el videojuego también implementa la interfaz *CommandListener*. Esta interfaz contiene un único método abstracto a implementar, *commandAction(Command c, Displayable d)*, que servirá para poder indicar la acción a realizar cuando se produzca un evento en el *Command c* del objeto *Displayable d*. En concreto, se utilizará para poder detectar el abandono de la partida por parte de alguno de los jugadores tras pulsar el correspondiente botón de salida de la partida. Esta interfaz se ha preferido implementar sobre el *MIDlet*, en lugar de hacerlo directamente sobre la clase *Juego.java*, que es la que lleva el peso de la ejecución de la partida, porque, como ya se comprobó en la sección anterior, ya implementa otro interfaz y una misma clase no puede implementar varias interfaces. De todos modos,

una de las funciones asignadas al *MIDlet* es la de controlar en todo momento qué pantallas se muestran al usuario y proporcionar los métodos necesarios para cambiar de una pantalla a otra, por lo que implementar la interfaz *CommandListener* aquí, sigue teniendo sentido.

Aparte de los métodos abstractos que debe implementar la clase como consecuencia de heredar de *MIDlet* y de implementar la interfaz *CommandListener*, ésta define un importante número de atributos y métodos, de los que la mayor parte están relacionados con las diferentes pantallas de menú de la aplicación. La tabla 4.34 recopila todos estos atributos y los correspondientes métodos para acceder a ellos.

Con los métodos de tipo *get* aquí definidos, el control de las pantallas que se muestran en el terminal se hace desde el propio *MIDlet*. Cabe añadir que los métodos encargados de mostrar por pantalla la puntuación final obtenida en la partida o el nuevo record alcanzado, si así fuera, requieren una información adicional que no se necesita en el resto de pantallas: el *friendly name* de los terminales de los jugadores, junto con la puntuación obtenida y el tipo de partida en el que se ha alcanzado dicha puntuación.

Atributo	Método de acceso	Descripción
ayudaCanvas	*getAyudaCanvas()*	Pantalla de ayuda
buscarCanvas	*getBuscarCanvas()*	Menú de Búsqueda de dispositivos
desconexionExtremoCanvas	*getDesconexionExtremo()*	Mensaje de desconexión remota
esperaBusquedaCanvas	*getEsperaBusquedaCanvas()*	Mensaje de búsqueda en curso
esperaCanvas	*getEsperaCanvas()*	Mensaje de espera de inicio de partida
gameOverCanvas	*getGameOverCanvas()*	Menú de fin de partida
idiomaCanvas	*getIdiomaCanvas()*	Menú de selección de idioma
listDispositivosCanvas	*getListDispositivosCanvas()*	Menú con los dispositivos encontrados
mensajeBluetoothCanvas	*getMensajeBluetoothCanvas()*	Advertencia para habilitar Bluetooth
menuInicialCanvas	*getMenuInicialCanvas()*	Menú inicial de la aplicación
menuPuntuacionesCanvas	*getMenuPuntuacionesCanvas()*	Menú de puntuaciones en cada modo
noDispositivoSeleccionadoCanvas	*getNoDispositivoSeleccionadoCanvas()*	Mensaje de no selección de dispositivo
noDispositivosEncontradosCanvas	*getNoDispositivosEncontradosCanvas()*	Mensaje de no dispositivos encontrados
noServiciosEncontradosCanvas	*getNoServiciosEncontradosCanvas()*	Mensaje de no servicios encontrados
nuevaPartidaCanvas	*getNuevaPartidaCanvas()*	Menú de nueva partida
nuevoRecordCanvas	*getNuevoRecordCanvas(...)*	Mensaje de nuevo record alcanzado
opcionesCanvas	*getOpcionesCanvas()*	Menú de opciones de la aplicación
pantallaInicialCanvas	*getPantallaInicialCanvas()*	Pantalla de presentación del videojuego
puntuacionFinalCanvas	*getPuntuacionFinalCanvas(...)*	Mensaje con la puntuación final
puntuacionesCooperativoCanvas	*getPuntuacionesCooperativoCanvas()*	Menú de puntuaciones en cooperativo
puntuacionesDeathmatchCanvas	*getPuntuacionesDeathmatchCanvas()*	Menú de puntuaciones en *deathmatch*
salirCanvas	*getSalirCanvas()*	Menú de abandono de la aplicación
sonidoCanvas	*getSonidoCanvas()*	Menú para habilitar o no el sonido

Tabla 4.34. Atributos y métodos definidos para las pantallas de menú

Por otra parte, la clase también necesita de otros atributos que se analizan a continuación:

- **buscandoDispositivos**: es un booleano que determina si se está llevando a cabo la búsqueda de dispositivos remotos (*true*) o no (*false*). Sirve para coordinar la presentación de las pantallas de búsqueda en curso y de dispositivos finalmente encontrados tras la búsqueda. A través de los métodos *getBuscandoDispositivos()* y *setBuscandoDispositivos(boolean b)* se puede acceder a este método.

- **idioma**: indica el idioma seleccionado para la aplicación. Para facilitar el proceso de lectura, en lugar de tener que consultarlo desde RMS cada vez que muestre al usuario una pantalla de la aplicación, este atributo recoge cuál es el idioma escogido y se lee directamente desde aquí. Los métodos que gestionan este atributo son *getIdioma()* y *setIdioma(char idIdioma)*.

- **sonidos**: este atributo contiene todos los sonidos que, en caso de estar habilitada la correspondiente opción, se escucharán durante la ejecución del programa. Desde el método *getGestorDeSonidos()*, se obtiene para reproducir, parar o comprobar si está reproduciéndose cualquier sonido o para habilitar o deshabilitar el sonido de la aplicación.

- **esclavo**, **maestro**: son las instancias de las clases *Esclavo.java* y *Maestro.java*, respectivamente. En caso de que el jugador que maneja el terminal decida crear una partida, ejecutará el método *iniciarMaestro(int anchoPantalla, int altoPantalla)*, mientras que si por el contrario decide unirse a una partida creada previamente por otro jugador, ejecutará el método *iniciarEsclavo(int anchoPantalla, int altoPantalla)*. Posteriormente, una vez se haya decidido el rol que va a implementar el terminal en la comunicación, se podrá obtener a partir de los métodos *getEsclavo()* o *getMaestro()*, según corresponda.

- **display**, **midlet**: son los únicos dos atributos definidos como públicos en el *MIDlet*. El primero se utiliza allí donde se necesite indicar qué pantalla hay que

mostrar en el terminal, mediante una llamada al método *setCurrent(Displayable nextDisplayable)* de la clase *Display*. Por ejemplo, desde el método *startApp()* se indica que la aplicación debe comenzar preguntando al usuario si desea habilitar la opción de sonido o no. Para ello, la aplicación deberá mostrar por pantalla el consiguiente menú que permita elegir sobre esta cuestión, y lo hace mediante la instrucción *display.setCurrent(getSonidoCanvas())*.

Por su parte, **midlet** es una instancia del propio *MIDlet* de la aplicación, y se utiliza para permitir que desde *Juego.java* se puedan registrar en el *listener* de comandos del *MIDlet* los eventos capturados por los comandos definidos en esa clase, de manera que puedan ser gestionados desde el *MIDlet*. Esto lo hace mediante la instrucción *setCommandListener(BArkanoid.midlet)*.

CAPÍTULO 5: PLAN DE PRUEBAS

En este capítulo se detallarán las pruebas realizadas a la aplicación desarrollada para verificar su correcto funcionamiento. Para efectuar dichas comprobaciones, se ha tomado un conjunto significativo de muestras, abarcando las pruebas desde el emulador y las realizadas con móviles reales del mismo modelo, del mismo fabricante pero modelo distinto y de diferentes fabricantes, de forma que cubrieran el espectro de posibilidades de la manera más completa posible. Por otro lado, estas mismas pruebas sirvieron para examinar el funcionamiento tanto de la lógica del videojuego como del enlace Bluetooth establecido entre ambos terminales o instancias del emulador.

5.1. PRUEBAS CON EMULADORES

Durante el desarrollo de la aplicación se ha empleado el emulador Sun Wireless Toolkit para probar las versiones iniciales, antes de pasar a ejecutar el programa sobre móviles reales. Si bien sus resultados no siempre se corresponden con lo que después ocurre en los terminales físicos reales, ha sido de gran utilidad a la hora de comprobar el funcionamiento de la aplicación, especialmente a nivel de la lógica del videojuego, evitando tener que instalar en los dispositivos móviles cada nueva versión que se generaba. Esto ha sido posible gracias a que este emulador permite la ejecución de múltiples instancias sobre la misma máquina y entre ellas ha sido posible simular la comunicación por Bluetooth.

El conjunto de las pruebas realizadas desde el emulador abarca la práctica totalidad de la funcionalidad exigida a la aplicación, desde el manejo de los menús hasta el desarrollo de una partida en los dos modos de juego posibles. Tan solo aspectos como controlar las dos instancias al mismo tiempo para mover la nave o lanzar la pelota han tenido que reservarse para su posterior comprobación sobre los terminales físicos, dado que al ejecutar varias instancias en el ordenador solo se puede controlar una al mismo tiempo. Bajo este epígrafe se especifican las diferentes pruebas llevadas a cabo para verificar el funcionamiento de cada contenido de la aplicación.

Por un lado están las pruebas relativas al manejo de la aplicación y a la creación y búsqueda del servicio basado en el perfil de puerto serie, o servicio SPP:

- Navegación por las ventanas de la aplicación: comprobación de que todas las ventanas, botones y opciones funcionan correctamente y se visualiza en cada momento la pantalla que procede.
- Comprobación de que el texto del menú de ayuda puede representarse en los casos en que no quepa el texto completo en la pantalla del terminal, pudiendo leer el resto del texto desplazándose con las flechas hacia arriba y hacia abajo. En este caso, se verifica también si se dibuja una pequeña flecha en los extremos hacia donde quede texto sin representar.
- Creación del servicio SPP.
- Búsqueda de dispositivos remotos y de servicios SPP.
- Notificación al jugador en caso de que no haya ningún dispositivo remoto en disposición de ser descubierto.
- Notificación al jugador en caso de que el dispositivo remoto seleccionado no ofrezca ningún servicio SPP.
- Notificación al jugador en caso de que no haya seleccionado ningún dispositivo remoto para intentar unirse a una partida.

Por otra parte, las pruebas efectuadas para verificar el inicio satisfactorio de una nueva partida son las que siguen:

- Inicio de la primera partida, creándola en un terminal y uniéndose a ella desde el otro.
- Inicio de una nueva partida después de haber terminado otra por agotamiento del número de vidas de al menos uno de los dos jugadores, aceptando iniciarla desde el menú de fin de partida.
- Inicio de una nueva partida después de haber terminado otra por agotamiento del número de vidas de al menos uno de los dos jugadores, aceptando iniciarla desde el menú principal, después de rechazar el inicio desde el menú de fin de partida.
- Inicio de una nueva partida después de haber abandonado otra previamente iniciada.

Una vez se ha iniciado la partida, se efectuaron una serie de pruebas básicas, destinadas, la mayoría de ellas, a verificar el correcto funcionamiento de la lógica del videojuego para iniciar variables y resolver los eventos de juego. Se detallan a continuación:

- Comprobación de que el mapa que se carga en pantalla es el que corresponde cuando se va a iniciar un nuevo mapa.
- Comprobación de que se visualizan correctamente solo los ladrillos que caben en la pantalla de juego.
- Inicialización correcta de todas las variables al comenzar un nuevo mapa.
- Comprobación de que las naves no se salen de los extremos laterales, definidos por las paredes que delimitan el ancho de la pantalla de juego, cuando se desplazan hacia los lados.
- Comprobación de que la pelota es lanzada manualmente cuando se pulsa el botón de disparo.
- Comprobación de que la pelota es lanzada de forma automática si antes de *TIMER_THROW* ciclos de ejecución del bucle principal no se ha realizado manualmente.
- Comprobación de que la pelota se posiciona adecuadamente sobre la nave y con los valores de movimiento por defecto para cada eje cada vez que es reubicada sobre la nave para iniciar una nueva vida.
- Comprobación de que dos pelotas rebotan correctamente al colisionar entre ellas.
- Comprobación de que una pelota rebota correctamente al colisionar con una pared lateral.
- Comprobación de que una pelota rebota correctamente al colisionar con el extremo contrario al de la nave del jugador al que pertenece si la partida es en modo *deathmatch*.
- Comprobación de que se asigna el ángulo de rebote adecuado a una pelota tras impactar sobre una nave.
- Comprobación de que una pelota rebota correctamente tras impactar con un ladrillo, si no tiene activado el *power-up NOBOUNCE*.

- Comprobación de que se desprende la cápsula del *power-up* que se ha asociado al ladrillo recién impactado por una pelota o por un proyectil.
- Comprobación de que el *sprite* que representa a la cápsula desprendida gira hacia el lado correcto en cada terminal.
- Comprobación de que la velocidad de cada pelota que controla un mismo jugador aumenta de velocidad progresivamente cada *TIMER_SPEED* ciclos de ejecución del bucle principal mientras permanezca activa.

Por lo que a la gestión y tratamiento de *power-ups* respecta, se realizaron las siguientes pruebas:

- Activación y desactivación correcta de *power-ups* en uno y otro jugador.
- Comprobación de que la cápsula que contiene el *power-up* desaparece de la pantalla de los dos terminales, una vez ha sido recogida por la nave hacia la que iba.
- Comprobación de que la pelota del jugador queda adherida a la nave controlada por él cuando tiene activado el *power-up HOLDBALL*.
- Comprobación de que la pelota conserva el ángulo de rebote tras impactar con la nave y quedar adherida a ella cuando el jugador tiene activado el *power-up HOLDBALL*.
- Comprobación de que la velocidad de cada pelota activa controlada por el jugador se reduce, si no está ya en el valor mínimo, cuando habilita el *power-up SPEEDDOWN*.
- Comprobación de que la velocidad de cada pelota activa controlada por el jugador se incrementa, si no está ya en el valor máximo, cuando habilita el *power-up SPEEDUP*.
- Comprobación de que la velocidad movimiento de la nave controlada por el jugador se reduce, si no está ya en el valor mínimo, cuando habilita el *power-up STEPDOWN*.
- Comprobación de que la velocidad movimiento de la nave controlada por el jugador se incrementa, si no está ya en el valor máximo, cuando habilita el *power-up STEPUP*.

- Comprobación de que el número total de pelotas activas controladas por el jugador pasan a ser tres cuando habilita el *power-up THREEBALLS* y en ese momento solo controla una única pelota.
- Comprobación de que se activa la tercera pelota cuando el jugador recoge con la nave la cápsula del *power-up THREEBALLS* y en ese momento controla dos pelotas.
- Comprobación de que no se activa ninguna pelota más si el jugador habilita el *power-up THREEBALLS* y en ese momento controla ya tres pelotas.
- Comprobación de que el tamaño de la nave se reduce progresivamente para encogerse tras activar el *power-up SHIPGROWDOWN*.
- Comprobación de que el tamaño de la nave se incrementa progresivamente para ensancharse tras activar el *power-up SHIPGROWUP*.
- Comprobación de que se representa por pantalla una estela de color verde para cada pelota activa controlada por el jugador si tiene activado el *power-up NOBOUNCE*.
- Comprobación de que las estelas de color verde que deja cada pelota controlada por el jugador cuando tiene activado el *power-up NOBOUNCE* se representan correctamente en el dispositivo remoto.
- Comprobación de que una pelota no rebota al colisionar contra un ladrillo rompible si tiene activado el *power-up NOBOUNCE*.
- Comprobación de que la nave controlada por el jugador cambia de aspecto cuando se activa el *power-up SHOT*.
- Comprobación de que al pulsar el botón de disparo con el *power-up SHOT* activado se lanzan dos proyectiles desde la nave controlada por el jugador.
- Comprobación de que el jugador no puede volver a lanzar proyectiles inmediatamente después de haber disparado, sino que tiene que esperar un pequeño espacio de tiempo, determinado por *TIMER_SHOT*.
- Comprobación de que los proyectiles disparados por el jugador cuando tiene activado el *power-up SHOT* se representan correctamente en la pantalla del otro dispositivo.
- Comprobación de que la nave controlada por el jugador recupera su aspecto original cuando se desactiva el *power-up SHOT*.

- Comprobación de que el número de vidas disponibles del jugador se incrementa en una unidad, si no ha llegado al máximo de vidas permitido, tras activar el *power-up EXTRALIFE*.

Las pruebas referentes al sonido de la aplicación, comprenden los siguientes test que a continuación se enumeran. Se entiende que las pruebas en las que se reproduce un efecto de sonido solo tienen sentido si está habilitado el sonido en la aplicación:

- Comprobación de que el sonido de la aplicación queda habilitado o deshabilitado activándolo o no desde la pantalla inicial de confirmación de sonido.
- Comprobación de que el sonido de la aplicación queda habilitado o deshabilitado activándolo o no desde el menú de opciones de la aplicación.
- Ejecución de la melodía de presentación al mostrar la pantalla de presentación del videojuego y finalización de la melodía al pasar al menú principal.
- Ejecución del efecto de sonido de inicio de una partida cuando se crea una nueva o se inicia la búsqueda del dispositivo que ha creado esa partida.
- Ejecución del efecto de sonido de inicio de una nueva vida cada vez que un jugador estrena una vida de las que le quedan.
- Ejecución del efecto de sonido de rebote de la pelota al colisionar sobre una nave.
- Ejecución del efecto de sonido de rebote de la pelota al colisionar con un ladrillo rompible.
- Ejecución del efecto de sonido de rebote de la pelota al colisionar con un ladrillo irrompible.
- Ejecución del efecto de sonido de rebote entre dos pelotas.
- Ejecución del efecto de sonido asociado a la pérdida de una vida por parte de uno de los jugadores.
- Ejecución del efecto de sonido tras finalizar la partida.
- Ejecución del efecto de sonido asociado al *power-up HOLDBALL*, en lugar del sonido de colisión con la nave, cuando la pelota impacta y se adhiere a ella con el *power-up HOLDBALL* activado.
- Ejecución del efecto de sonido asociado al *power-up EXTRALIFE* cuando se activa dicho *power-up* en alguno de los jugadores.

- Ejecución del efecto de sonido asociado a los *power-ups SHIPGROWDOWN* y *SHIPGROWUP*, cuando se activa alguno de estos *power-ups* por algún jugador.
- Ejecución del efecto de sonido de disparo de proyectiles cuando éstos son lanzados por el jugador (se asume activado el *power-up SHOT*).

Una vez se completaron estas pruebas, se efectuaron a continuación otras cuyo objetivo es verificar el funcionamiento de la lógica del videojuego en su conjunto. Son las que se detallan a continuación:

- Comprobación de que ambos terminales pueden enviar y recibir mensajes correctamente a través del enlace Bluetooth.
- Comprobación de que se mantiene la coherencia en la ejecución de la partida en ambos terminales en todo momento.
- Comprobación de que se resta una vida al jugador que maneja la nave sobre la que rebota una pelota del otro jugador cuando la partida es en modo *deathmatch*.
- Comprobación de que no se resta una vida al jugador que maneja la nave sobre la que rebota una pelota del otro jugador cuando la partida es en modo cooperativo.
- Comprobación de que se resta una vida a un jugador cada vez que se pierde de la pantalla la última pelota controlada por él.
- Comprobación de que se suman 100 puntos por cada ladrillo eliminado, rompible tras un impacto, y 200 por cada ladrillo rompible tras dos impactos.
- Comprobación de que se ha alcanzado un nuevo record y de que se informa convenientemente por pantalla, en caso de superar la marca de alguna de las puntuaciones registradas para ese modo de juego.
- Lectura y escritura correcta de registros de puntuaciones obtenidas y de idioma seleccionado para la aplicación.
- Verificación del posicionamiento de cada componente junto con el funcionamiento de la aplicación al simular instancias con tamaños de pantalla diferentes.
- Comprobación de que al abandonar una partida ya comenzada, se finaliza y se vuelve al menú principal de la aplicación, forzando en el otro dispositivo la aparición de un mensaje notificando la desconexión del extremo remoto.

- Cierre satisfactorio de la aplicación, liberando la conexión Bluetooth establecida y los recursos asignados.

5.2. PRUEBAS CON DISPOSITIVOS MÓVILES REALES

Tras haber pasado satisfactoriamente cada test efectuado en el emulador, se procedió a ejecutar el programa en móviles reales. Como es lógico, estos terminales precisaban disponer de conectividad Bluetooth para poder jugar al videojuego.

Para la mayor parte de las pruebas, se utilizó una pareja de móviles del mismo modelo, en concreto dos teléfonos Nokia E65, para testear la ejecución de la aplicación en dos móviles de características similares. Posteriormente, se incluyó en este plan de pruebas un Sony Ericsson K750i para analizar el comportamiento del programa utilizando dos móviles de distinto fabricante. Finalmente, se añadieron dos modelos diferentes de Nokia, en concreto el Nokia N93 y el E61, para comprobar el funcionamiento en diferentes modelos de móviles del mismo fabricante. El sistema operativo presente en los teléfonos Nokia que se han utilizado para las pruebas es Symbian, mientras que el terminal de Sony Ericsson dispone de un sistema operativo propietario.

En este sentido, además de las pruebas anteriormente descritas para ser ejecutadas sobre el emulador, y que necesitaron ser igualmente validadas en los terminales reales, se realizaron las siguientes comprobaciones:

- Ejecución de la aplicación en dos terminales del mismo modelo.
- Ejecución de la aplicación en dos terminales del mismo fabricante, pero distinto modelo.
- Ejecución de la aplicación en dos terminales de distinto fabricante.

Para cada uno de estos escenarios, se tuvieron en cuenta algunos test adicionales a los ya descritos en el anterior punto, exclusivos para los dispositivos reales, que se enumeran a continuación:

- Inicio satisfactorio de la aplicación.
- Comprobación de que el terminal responde adecuadamente a cada evento de teclado.
- Funcionamiento correcto del sonido de la aplicación.
- Comprobación de que se ejecuta correctamente el envío y la recepción simultánea de mensajes en cada terminal.

5.3. ANÁLISIS DE LOS RESULTADOS

Las pruebas realizadas sobre los móviles reales constataron que no todo lo que funciona correctamente en el emulador funciona también en los dispositivos físicos, especialmente en lo referente al establecimiento del enlace Bluetooth y el envío y recepción de mensajes a través de dicha conexión. Por ejemplo, enviar mensajes al otro terminal desde el mismo hilo de ejecución por el que los recibe, no provoca errores de funcionamiento en el emulador, pero sí en los teléfonos móviles, llegando incluso a bloquear la aplicación en el terminal.

A esto hay que añadir que el retardo introducido en el envío y la recepción de mensajes entre las dos instancias lanzadas del emulador es prácticamente despreciable, mientras que en los móviles este lapso de tiempo entre una acción y otra puede ser variable entre parejas diferentes de terminales e incluso durante una misma partida con los mismos dispositivos. No obstante, dicho retraso siguió siendo lo suficientemente pequeño como para que no afectara al desarrollo natural de una partida, si bien los eventos que generan el envío de información de sincronización se suceden con la frecuencia necesaria como para contrarrestar dicho desfase.

Por otra parte, es de notar que, mientras en el emulador no era necesario habilitar la visibilidad Bluetooth de cada instancia (viene activada por defecto), en los terminales móviles se hacía imprescindible habilitar esta opción antes de ejecutar la aplicación, de lo contrario no se podía establecer el enlace y, en consecuencia, el inicio de la partida no podía tener lugar. Este hecho motivó la inclusión de una pantalla en el sistema de ventanas de la aplicación que advirtiera de la necesidad de activar primero la capacidad Bluetooth del dispositivo.

CAPÍTULO 6: CONCLUSIONES Y LÍNEAS FUTURAS

6.1. CONCLUSIONES FINALES

El videojuego multijugador desarrollado en el presente proyecto se basa en el original "Arkanoid" para un solo jugador, desarrollado por Taito en 1986. En particular, se trata de una adaptación para dispositivos móviles, tomando como referencia las características gráficas y sonoras y el funcionamiento general del videojuego (manejo, rebotes, vidas…). No obstante, el carácter multijugador de la versión implementada abre nuevas posibilidades en lo que a modos de juego se refiere. Concretamente, se han desarrollado dos posibles modos de juego, uno cooperativo y otro no cooperativo (*deathmatch*) que, si bien introducen pequeñas variantes con respecto a la funcionalidad del videojuego original, respetan el planteamiento general y la idea de juego del mismo, conservando el modo de manejar la nave para hacer rebotar la pelota del jugador sobre ella e incluyendo muchos de los *power-ups* definidos en él y en sus posteriores secuelas.

Los dos modos de juego mencionados sirven para que los dos jugadores compitan sobre un mismo conjunto de diez niveles, cada uno con su propia distribución para los ladrillos que los componen. Al terminar el décimo nivel, se continúa por el primero de nuevo, repitiéndose este proceso mientras la partida se mantenga activa. Ésta se puede dar por terminada cuando uno de los dos jugadores haya agotado todas sus vidas disponibles o cuando alguno de ellos la haya abandonado de forma voluntaria pulsando sobre la correspondiente opción. Asimismo, la aplicación almacena las tres mejores puntuaciones que se registren para cada uno de estos modos.

Como se ha comentado, gran parte de los *power-ups* finalmente implementados, han sido inspirados en los del videojuego original y en sus secuelas. La versión multijugador que se ha desarrollado cuenta con el *power-up* que atrapa la pelota; los que disminuyen o aumentan la velocidad de las pelotas controladas por el jugador que lo activa y del movimiento de la nave cuando ésta es desplazada hacia la izquierda o hacia la derecha; el *power-up* que triplica el número de pelotas; los que ensanchan o encogen la nave; el que deshabilita el rebote de la pelota al impactar contra un ladrillo rompible; el *power-up* de disparo; y el que proporciona una vida extra al activarlo.

Para hacer más intuitivo el manejo en el videojuego, se ha dispuesto en la parte inferior de la pantalla de cada dispositivo la nave que maneja el jugador. Para ello, ha sido necesario un proceso de inversión de la imagen que se muestra en cada momento de la partida en un terminal con respecto a la que se representa en el otro.

Por otra parte, los menús y los diferentes mensajes de texto que integran el sistema de ventanas de la aplicación, se muestran en un único estilo de representación común para todas ellas, con el mismo fondo de imagen del propio videojuego, compuesto de un fondo de color negro con estrellas, simulando el espacio. Estas opciones y textos se pueden presentar en tres idiomas diferentes, a determinar por cada jugador: español, inglés o italiano.

La aplicación también dispone de una serie de efectos de sonido y una melodía de presentación del videojuego, que sonarán en el terminal si se habilita la opción de sonido. Esta activación se puede realizar, bien nada más iniciar la ejecución en el dispositivo, o bien posteriormente desde el menú de opciones. Los efectos de sonido, como ya se ha comentado al inicio del presente epígrafe, reproducen fielmente el apartado sonoro del videojuego original, ejecutándose los correspondientes a inicios de nuevas partidas, pérdidas e inicios de vidas disponibles, colisiones, recogida de algunos *power-ups*, lanzamiento de eventuales proyectiles o fin de partida.

A la hora de elaborar el código de la aplicación, se ha empleado un lenguaje de alto nivel, en concreto, Java ME. El hecho de programar la aplicación mediante un lenguaje de estas características facilita la tarea de confeccionar el código, ya que éste se compone de instrucciones más fáciles de entender por el ser humano, pero como contrapartida se aleja del lenguaje máquina, convirtiéndolo en un código menos eficiente y más pesado, ya que necesita atravesar más capas de abstracción para llegar al nivel del lenguaje máquina.

La realización de este proyecto mediante dicho lenguaje, ha permitido conocer y profundizar sobre técnicas de programación propias de lenguajes concebidos para dispositivos con capacidades limitadas, como es el caso de los dispositivos móviles.

Además, ha servido para poner de manifiesto la necesidad de desarrollar el código de la aplicación en diferentes hilos o *threads* que se ejecutan de forma concurrente.

Por otro lado, se ha utilizado un protocolo de comunicación a través del enlace Bluetooth que satisficiera las necesidades de comunicación entre ambos terminales, en concreto, el protocolo RFCOMM. Para poder mantener la coherencia en la ejecución del videojuego en los dos dispositivos, se ha requerido desarrollar un mecanismo de sincronización sobre este enlace. Dicho mecanismo, además, se ha implementado en aras de la interactividad del videojuego, de forma que interfiriera lo menos posible con la ejecución del mismo.

La metodología empleada para desarrollar este proyecto ha consistido en implementar sucesivas versiones que fueran incluyendo progresivamente complejidad y nuevos elementos al desarrollo, considerando desde el primer momento el modo multijugador y la correspondiente necesidad asociada de estar en comunicación con el dispositivo remoto.

Se ha partido desde el modelo más básico, consistente en dos pelotas en movimiento que rebotan al colisionar entre ellas, con las paredes laterales y con el extremo superior de la pantalla del dispositivo que es local a cada una de ellas; en el extremo inferior, rebotando con un ángulo aleatorio para comprobar de igual forma el funcionamiento para todos los ángulos que posteriormente se iban a tener en cuenta en los sucesivos rebotes con las naves.

A continuación, se incluyeron las naves, y con ellas la interacción de cada jugador con el manejo del videojuego, pudiendo desplazar éstas de un lado para otro y posteriormente, habiendo fijado un punto de inicio que situaba a cada pelota sobre su correspondiente nave, lanzar la pelota pulsando el botón de disparo.

Más adelante se definieron los ladrillos que, combinándolos adecuadamente, dieron forma a los diez mapas que componen el videojuego actual. Una vez se incorporaron los mapas con sus ladrillos, se comenzó a desarrollar uno a uno los diferentes tipos de *power-ups* que finalmente se han implementado, junto con sus cápsulas asociadas y los eventuales efectos que pudieran añadir a la partida (proyectiles,

más pelotas, estela de color verde que pudiera dejar la pelota…). Además, se introdujeron los efectos de sonido relativos a cada evento de juego (colisiones con ladrillos, rebotes con naves, disparos, recepción de algunos *power-ups*…) y la melodía de presentación del mismo.

Finalmente, se diseñó el mecanismo para el conteo de puntos, el control del número de vidas que le quedan a cada jugador y se concretaron en el código las particularidades que distinguen a cada modo de juego definido para esta versión multijugador, al tiempo que se incorporó el mecanismo de lectura y almacenamiento de puntuaciones e idioma seleccionado mediante RMS.

Paralelamente al desarrollo del videojuego, se fue dando forma al sistema de ventanas y pantallas de menú de la aplicación. En un primer momento, se compuso a partir de los formularios y listas que proporciona el propio lenguaje de programación. Estos elementos, sin embargo, no permiten configurar la disposición de sus componentes en la pantalla, ni su presentación, impidiendo establecer cualquier criterio sobre el tipo de letra o el juego de colores para representar cada ítem, lo que redunda en una apariencia un tanto tosca y pobre. Además, el fondo de imagen es dependiente del terminal sobre el que se ejecuta el programa, lo que conlleva que la misma aplicación pueda adoptar diferentes aspectos según en qué dispositivo se lance.

De aquí surgió entonces la idea de uniformar la estética del programa, definiendo una clase común que se encargara de establecer los parámetros tales como el tipo de letra, tamaño, color de cada tipo de recuadro para los textos, imagen de fondo, etc., de la misma manera que una hoja de estilo CSS define los aspectos de formato y presentación de contenidos para las diferentes páginas que componen un mismo sitio web.

Esta decisión, además, traería consigo un par de ventajas añadidas: por un lado, se lograría de una forma bastante práctica la separación automática entre la parte del código encargada de la presentación de la información y la parte que se ocupa del tratamiento de la misma; y por otro, el hecho de aglutinar en una sola clase todas las características visuales y de estilo, agiliza considerablemente la tarea de corregir o de modificar alguna de ellas y que sea efectivo para todas las pantallas de menú.

6.2. LÍNEAS FUTURAS

Durante el desarrollo de este proyecto, se ha puesto de manifiesto que en la elaboración de un videojuego, por sencillo que sea, se pueden considerar distintas alternativas y variantes, según qué elementos se desee tener en cuenta y añadir finalmente, de manera que deja la puerta abierta a multitud de posibilidades y nuevas características que el programador debe decidir si incluirlas o no, en función de su complejidad o de si se alejan o no del concepto o la esencia del propio videojuego.

Como ya se ha comentado en el punto anterior, la propia naturaleza multijugador del videojuego implementado permite jugar con nuevas posibilidades y nuevos modos de juego. Además de los modos ya mencionados, el cooperativo y el *deathmatch*, se podían haber definido otros modos de juego que aprovecharan igualmente esta particularidad, como, por ejemplo, uno en el que se contara cada mapa superado por victorias para uno u otro jugador según el número de ladrillos eliminados o el total de puntos acumulados, o un modo de juego en que se dispusiera de un tiempo limitado para cada nivel.

En esta línea, también se podía haber añadido alguna característica inspirada en el videojuego “Breakout”, comentado en el primer capítulo, como aquella en que las filas de ladrillos se desplazaban progresivamente hacia abajo. Para la versión multijugador, podría haberse definido, por ejemplo, un sentido de movimiento hacia arriba para las filas que se encuentren en la mitad superior de la pantalla y hacia abajo para las que se encontraran en la mitad inferior, o concretar esta propiedad en un nuevo *power-up* que produjera un movimiento de las filas de ladrillos en sentido contrario al jugador que lo recoge, para perjudicar al otro jugador.

Igualmente, se podrían haber incluido algunas características propias de las distintas versiones de “Arkanoid”, expuestas también en el capítulo 1, como la posibilidad de encontrar ladrillos que se desplazan horizontalmente, con movimiento pendular con respecto a su posición original, o que desaparecen y vuelven a aparecer tras haber sido eliminados.

Como posible mejora del videojuego implementado, en lugar de determinar qué ladrillos se pueden representar por pantalla en función del tamaño final de ésta, se podría haber reducido el tamaño de cada ladrillo para que se representaran siempre todos los ladrillos que componen cada mapa. Esto conllevaría un cambio en la implementación, dado que ahora habría que evaluar primero cuál debería ser el tamaño de cada ladrillo para poder representarlos a todos en la pantalla del dispositivo y después, en función de ese tamaño, determinar las nuevas coordenadas que debería adoptar tras la corrección. Además, en lugar de tomar un fichero de imagen, que tiene unas dimensiones fijas, como recurso para componer el *sprite* que representa cada ladrillo, sería más práctico pintar en pantalla, con el color correspondiente, la zona delimitada por la posición y las dimensiones de cada uno. Y para no perder la proporcionalidad del volumen de los elementos representados, sería igualmente necesario redimensionar las pelotas y naves. Esto último debería hacerse también, no solo por una cuestión estética, sino por evitar situaciones ambiguas en las que, por ejemplo, el nuevo sentido de movimiento de una pelota tras impactar con la nave fuera otro distinto al que le corresponde en realidad por no tener el tamaño apropiado para ese caso.

Adicionalmente, podría permitirse al jugador la posibilidad de jugar no sólo a los niveles que aporta el propio videojuego, sino de desarrollar además sus propios mapas con ayuda de un editor de niveles, como ocurre con muchos videojuegos multijugador. Esta opción alargaría la vida útil del producto, si bien ahora cada jugador dispone de los mapas originales, los que crea él mismo y los que haya podido confeccionar cualquier otro jugador con el que desee iniciar una partida.

Por otra parte, queda también abierta la posibilidad de modificar la aplicación de manera que su apariencia se deje a gusto del usuario, a partir de una serie de *skins* predefinidos que determinen el juego de colores empleado para representar cada ítem de las pantallas de menú, en lugar de disponer de un único estilo gráfico. Para ello, bastaría con añadir a la clase *GUICanvas.java* los conjuntos de atributos que almacenan el código de cada color para estos nuevos estilos, y adicionalmente insertar una nueva pantalla de menú encargada de gestionar la elección de un *skin*, en caso de que se desee cambiar la apariencia de la aplicación.

BIBLIOGRAFÍA Y REFERENCIAS

BIBLIOGRAFÍA

Sergio Gálvez Rojas, Lucas Ortega Díaz, "Java a tope: J2ME", Dpto. de Lenguajes y Ciencias de la Computación, E.T.S. de Ingeniería Informática, Universidad de Málaga, 2003.

Manuel Jesús Prieto, "Desarrollo de juegos con J2ME", Ra-Ma Editorial, 2005.

Agustín Froufe Quintas, Patricia Jorge Cárdenes, "J2ME: Manual de usuario y tutorial", Ra-Ma Editorial, 2004.

Grady Booch, James Rumbaugh, Ivar Jacobson, "El Lenguaje Unificado de Modelado", Addison-Wesley, 2006.

Página de referencia para consultar las librerías disponibles en Java ME, para el perfil MIDP 2.0:
http://download.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html

Página que recopila multitud de artículos relacionados con Java ME, sobre CLDC, MIDP, tutoriales de iniciación, programación de videojuegos, ejemplos de uso de librerías y funciones…:
http://www.developer.nokia.com/Community/Wiki/Category:Java_ME

Sub-foro de la comunidad de developer.nokia.com dedicado a Java ME:
http://www.developer.nokia.com/Community/Discussion/forumdisplay.php?3-Mobile-Java

REFERENCIAS

[1] Killer List of VideoGames, web dedicada a mantener una amplia base de datos de máquinas recreativas.
http://www.klov.com/game_detail.php?game_id=9074

[2] Artículo relacionado con la crisis del sector de los videojuegos de 1983 de la versión inglesa del portal Web de la Wikipedia.
http://en.wikipedia.org/wiki/North_American_video_game_crash_of_1983

[3] Web de la aDeSe (Asociación española de Distribuidores y Editores de Software de Entretenimiento).
http://www.adese.es/

[4] Noticia del 24 de Marzo de 2010, publicada en el portal de tecnología tuexpertojuegos.com, titulada "Los videojuegos venden en España más que el cine y la música juntos".
http://www.tuexpertojuegos.com/2010/03/24/los-videojuegos-venden-en-espana-mas-que-el-cine-y-la-musica-juntos/

[5] Conclusiones de la última edición del informe Mobile Life, el estudio anual sobre tendencias en telefonía móvil, elaborado por la compañía TNS.
http://www.tns-global.es/actualidad/noticias/4-de-cada-10-usuarios-de-movil-espanoles-se-conectan-a-paginas-web-a-traves-de-su-telefono(292)/

[6] Noticia del 29 de Junio de 2011, publicada en la página de Radio Televisión Española, titulada "La corta vida de los teléfonos móviles, un modelo de producción insostenible".
http://www.rtve.es/noticias/20110629/corta-vida-moviles-modelo-insostenible/444449.shtml

[7] Artículo de pixfans.com, portal dedicado al mundo de los videojuegos, titulado "Máquinas arcade, pinballs, futbolines y tragaperras", y que repasa la historia de cada una de ellas.
http://www.pixfans.com/historia-de-las-maquinas-recreativas-pinballs-futbolines-arcades-y-tragaperras/

[8] Artículo de la versión inglesa del portal web de la Wikipedia dedicado al videojuego "Breakout".
http://en.wikipedia.org/wiki/Breakout_(video_game)

[9] Página oficial del equipo de desarrollo de MAME.
http://mamedev.org/

[10] Página de descarga del videojuego "Super Breakout" para MAME.
http://www.rom-world.com/file.php?id=21752

[11] Artículo de la versión inglesa del portal web de la Wikipedia dedicado al videojuego "Arkanoid".
http://en.wikipedia.org/wiki/Arkanoid

[12] Artículo de la versión española del portal web de la Wikipedia dedicado al videojuego "Arkanoid".
http://es.wikipedia.org/wiki/Arkanoid

[13] Artículo en Meristation sobre la versión de "Arkanoid" para la consola Nintendo DS, "Arkanoid DS".
http://www.meristation.com/v3/des_analisis.php?id=cw49005b087186b&idj=cw47f3477d0ea17&pic=DS

[14] Artículo de la versión española del portal web de la Wikipedia dedicado a Java.
http://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)

[15] Web del IDE Eclipse.
http://www.eclipse.org/

[16] Web del IDE NetBeans.
http://www.netbeans.org/

[17] Web del emulador de dispositivos MIDP Sun Wireless Toolkit.
http://java.sun.com/products/sjwtoolkit/download.html

[18] Web de la API para J2ME: JSR82.
http://java.sun.com/javame/reference/apis/jsr082/

[19] Página oficial de la aplicación GIMP.
http://www.gimp.org/

[20] Artículo de la versión española del portal web de la Wikipedia dedicado al programa GIMP.
http://es.wikipedia.org/wiki/GIMP

[21] Página de descarga de SoundTap desde el portal softonic.com.
http://soundtap.softonic.com/

[22] Página de descarga del videojuego “Arkanoid” para MAME.
http://www.rom-world.com/viewscreenshots.php?id=4939

[23] Artículo de la versión española del portal web de la Wikipedia dedicado al programa MAME.
http://es.wikipedia.org/wiki/Multiple_Arcade_Machine_Emulator