



Diseño y creación de aplicaciones de entretenimiento sobre dispositivos móviles

Memoria del proyecto

Autor: Mario Macías Lloret

Tutor: Agustín Trujillo Pino

Facultad de Informática

Universidad de Las Palmas de Gran Canaria
Curso 2004/05

Prólogo

Este proyecto de fin de carrera trata sobre la creación de aplicaciones de entretenimiento para aparatos de telefonía móvil u otros dispositivos de recursos computacionales limitados, como PDAs. Se ha hecho hincapié en aplicaciones multimedia y de entretenimiento, tales como videojuegos, ya que son las que hoy en día disfrutan de una mayor extensión dentro del sector de la telefonía móvil, con un mercado que año a año se incrementa enormemente.

El proyecto se ha dividido en dos partes diferenciadas: la primera, donde se estudian las diferentes metodologías y herramientas para la creación de aplicaciones móviles, y segunda, donde se entra de lleno en las técnicas de programación de videojuegos, desde un punto de vista principalmente de la ingeniería del software.

Tras el primer capítulo, que pretende mostrar al lector en las motivaciones del proyecto, viene la primera parte, donde se estudian los diferentes sistemas disponibles para la programación de aplicaciones móviles (capítulo 2), se comenta la tecnología J2ME, escogida para la realización del proyecto (capítulo 3), los entornos de desarrollo disponibles (capítulo 4), y las herramientas que *Sun Microsystems*, creador de Java, pone a disposición de los programadores (capítulo 5).

En la segunda parte se entra de lleno en la programación en Java para móviles, partiendo de que se conoce el lenguaje Java y sus características, pero se tiene un conocimiento nulo acerca de las API que *Sun* proporciona para tal tarea. El capítulo 6 es una introducción, con un pequeño tutorial en el que se explican los conocimientos básicos de las API de J2ME para la realización de aplicaciones básicas e interfaces de usuario.

En el capítulo 7 se entra de lleno en la realización de videojuegos, donde se comentan las características deseables de una aplicación de entretenimiento para un teléfono móvil, desde el punto de vista técnico, de ergonomía y de diseño de un videojuego. Todos estos conocimientos se pretenden aplicar en el capítulo 8, con la creación del primer videojuego, de corte sencillo, pero que es un primer intento de sacar a la luz las posibilidades creativas de la plataforma.

Esto no debe ser suficiente en un proyecto ambicioso como este, por ello en el capítulo 9 se crea un videojuego de un aspecto más avanzado y profesional. Se explica la estructura de la aplicación propuesta y algunas técnicas para la realización de

videojuegos avanzados.

Gracias a la estructura modular de la aplicación creada en el capítulo 9, en el siguiente capítulo se decidió reaprovechar gran parte del código, modificándolo mínimamente, para crear una librería que permitiera comenzar nuevos videojuegos con una estructura predefinida, con el objetivo de facilitar y agilizar el proceso de creación.

A lo largo del proyecto se han descubierto múltiples campos de utilización de las plataformas móviles como herramientas de ocio. Debido a la imposibilidad de abarcarlos todos, en el último capítulo, junto con las conclusiones del proyecto, se han descrito numerosas líneas de estudio y desarrollo en las cuales podrían dirigirse futuros proyectos.

Índice

Prólogo.....	3
Capítulo 1.Introducción al proyecto.....	9
1.1Evolución de la telefonía móvil.....	9
1.2El caso de los videojuegos.....	9
1.3Situación de España.....	11
1.4¿Qué hace la universidad al respecto?.....	11
1.5Conclusión.....	13
Parte I. Java 2, Micro Edition (J2ME).....	15
Capítulo 2.Sistemas de desarrollo y ejecución de aplicaciones.....	17
2.1Reprogramación de la ROM del terminal.....	17
2.2Programación en lenguajes compilados con las API del sistema operativo.....	17
2.3Programación en J2ME.....	18
2.3.1Programación mediante librerías estándar de J2ME.....	19
2.3.2Programación con J2ME estándar + API del fabricante.....	20
2.3.3Creación de interfaces entre el programa y las API.....	20
2.4Conclusiones.....	21
Capítulo 3.Introducción a J2ME.....	23
3.1Introducción.....	23
3.2La arquitectura J2ME.....	24
3.3Configuraciones.....	24
3.4Perfiles.....	24
3.5Paquetes Opcionales.....	25
3.6Características de J2ME frente a J2SE.....	25
3.7Descriptor, Manifiesto y MIDlet.....	27
Capítulo 4.Entornos de desarrollo.....	29
4.1IntelliJ IDEA.....	29
4.2NetBeans IDE.....	31
4.3Eclipse.....	32
4.4El elemento esencial: J2ME Wireless Toolkit (WTK).....	33
4.5Conclusión.....	34
Capítulo 5.Utilización de J2ME Wireless ToolKit (WTK).....	37
5.1Default Device Selection.....	37
5.2Preferences.....	37
5.3Run MIDP application.....	37
5.4KToolbar.....	38
Parte II. Programación de videojuegos.....	39
Capítulo 6.Programación en J2ME.....	41
6.1Introducción a los MIDlets.....	41
6.1.1Ciclo de vida de un MIDlet.....	41
6.2Interfaces de usuario.....	42
6.2.1Interfaz de alto nivel.....	42
6.2.2Interfaz de bajo nivel.....	44
6.3Un ejemplo práctico.....	45
6.3.1Creación y configuración del proyecto.....	45

6.3.2Clase Ejemplo.....	46
6.3.3Clase MenuPrincipal.....	47
6.3.4Clase Configuracion.....	48
6.3.5Clase Lienzo.....	49
6.3.6Compilación, prueba y empaquetado.....	51
Capítulo 7.Criterios de diseño de un videojuego sobre una plataforma móvil.....	53
7.1Crear un buen videojuego.....	53
7.2Factores técnicos.....	53
7.3Factores de ergonomía.....	54
7.3.1Controles.....	54
7.3.2Pantalla.....	56
7.4Consejos para la mejora de la jugabilidad.....	57
Capítulo 8.El primer juego.....	59
8.1Estructura de la aplicación.....	59
8.2La clase TareaJuego.....	61
Capítulo 9.Creación de un videojuego avanzado.....	67
9.1Características (deseables) de un videojuego.....	68
9.2Estructura de la aplicación.....	69
9.3El contenedor de todo: GestorElementos.....	70
9.4GestorGrafico, lienzo de pintura y manejador de eventos.....	73
9.5El núcleo de todo: GestorJuego.....	73
9.5.1Sincronización avanzada.....	74
9.6Los niveles del juego: la clase Fase.....	77
9.7Clase ObjetoMovil y derivadas.....	79
9.7.1Características de ObjetoMovil.....	80
9.7.2Clase Enemigo.....	80
9.7.3Clase Disparo.....	81
9.7.4Clase Option.....	81
9.8La gestión del sonido.....	81
9.8.1La clase MediaPlayer.....	81
9.8.2La clase GestorSonidos.....	82
Capítulo 10.Creación de una librería para la programación de videojuegos.....	83
10.1Cambios realizados.....	84
10.1.1Clase GestorJuego.....	84
10.1.2Clase GestorElementos.....	84
10.1.3Clase GestorSonidos.....	84
10.1.4Clase Protagonista.....	84
10.1.5Clase Fase.....	84
10.2Utilización de la librería para la creación de videojuegos.....	85
Capítulo 11.Conclusiones y líneas abiertas.....	87
11.1Conclusiones.....	87
11.2Líneas abiertas.....	88
11.2.1Creación de un entorno de desarrollo de videojuegos.....	88
11.2.2Otras líneas abiertas.....	91
Parte III. Java 2, Micro Edition (J2ME).....	93
Apéndice A. JavaDoc de la librería para la creación de videojuegos.....	95
Apéndice B. Código fuente del videojuego “Escabechina”.....	125
Apéndice C. Código fuente del videojuego de demostración.....	135
Bibliografía.....	167

Capítulo 1. Introducción al proyecto

Este capítulo, a modo de introducción, pretende describir la evolución de la telefonía móvil a nivel mundial.

1.1 Evolución de la telefonía móvil

Fue en 1995, cuando en nuestro país empezaron a implantarse las primeras operadoras de telefonía móvil de segunda generación. Al principio era un servicio principalmente utilizado en el ambiente empresarial y por una minoría de particulares. Fue a partir de 1999 que el mercado de la telefonía móvil sufrió un brutal incremento: el teléfono móvil empezó a popularizarse de tal manera que hoy en día es un elemento habitual y casi indispensable en el estilo de vida de los países desarrollados.

Este *boom* en las ventas, estrechamente relacionado con una gran evolución tecnológica (móviles cada día más pequeños, baratos y con más funcionalidades) ha dado pie a que cada día se introduzcan más empresas en el mercado de los servicios a móviles. En la figura 1.1 se muestra una evolución aproximada de los ingresos y áreas de negocio de los servicios de datos para usuarios domésticos bajo telefonía móvil.

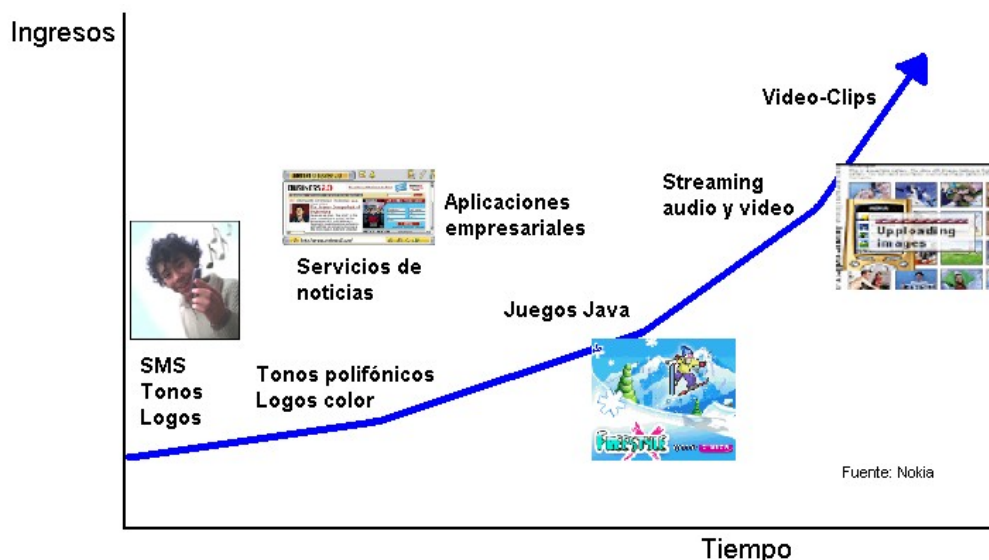


Figura 1.1: evolución de las áreas de negocio de los servicios de datos de la telefonía móvil

1.2 El caso de los videojuegos

Este proyecto estará dedicado a la creación de videojuegos para teléfonos móviles, por lo que ahora nos centraremos en los datos económicos de dicho área. He aquí unos

datos de interés:

- Según los estudios de Nokia, los ingresos de las operadoras mediante la venta de videojuegos móviles pasara del 3% en 2003 al 11% en 2008. Estamos hablando de un **incremento del 370% en cinco años**.
- Sun publicó los siguientes datos en la **Java Expo '04**:
 - Según estimaciones, se producen mundialmente más de 15 millones de descargas de juegos Java cada mes.
 - Hasta 2002, Vodafone Japón reportó 25 millones de descargas de juegos.
 - En mayo de 2003, la compañía japonesa NTT-DoCoMo reportaba entre 100.000 y 200.000 descargas diarias.
 - Desde su creación en octubre de 2003, hasta mediados de 2004, el servicio Vodafone Live! Reportó 3 millones de descargas.
 - Durante 2003, Telefónica Móviles vendió 3 millones de aplicaciones Java.

Debido al fuerte y continuo crecimiento de la telefonía móvil, estos datos son actualmente obsoletos, pero si los combinamos con las expectativas de crecimiento anteriormente expuestas, estamos ante un gran mercado emergente que mueve varios miles de millones de euros al año, y que proporciona miles de puestos de trabajo.

Aparte de estos datos, está contabilizado que la industria del videojuego en general (no sólo de telefonía móvil), hoy en día mueve más dinero que la propia industria cinematográfica.

Como último apunte de este apartado, es digno de tener en consideración el siguiente artículo periodístico extraído de *LaFlecha.net*:

El mercado de juegos en móviles superará los 1.000 millones de dólares

Las compañías de videojuegos y de telefonía móvil están destinadas a hacer dinero este año gracias a la creciente tendencia de los clientes a utilizar sus móviles para destruir a invasores del espacio o golpear bolas en campos de golf virtuales, llevando a este mercado a superar los 1.000 millones de dólares, según un estudio.

12:00 - 29/10/2004 | Fuente: REUTERS

Según la consultora Screen Digest, con sede en Londres, el juego en los teléfonos móviles

también crecerá más de seis veces, hasta los 6.400 millones de dólares, entre el año 2004 y el final de la década.

Actualmente, Japón y Corea van bastante por delante de Norteamérica y Europa en tamaño de mercado, y suponen casi el 80 por ciento de todos los ingresos derivados de los juegos y las descargas, dijo el jueves Screen Digest.

Un confuso laberinto de tarifas para descargas y juegos cobradas por las operadoras móviles en Europa está reduciendo el crecimiento en la región, dijo Screen Digest.

"Pensamos que las operadoras móviles en Europa aún no tienen las estrategias adecuadas para explotar este mercado hasta su completo potencial", dijo el analista jefe de Screen Digest, Ben Keen.

La compañía añadió que se prevé que Norteamérica, pese a tener un mercado de telefonía móvil menos sofisticado que el de Europa y Asia, crezca a un mayor ritmo que esas regiones.

Frente al próspero negocio de los videojuegos, el naciente mercado de los juegos en móviles acaba de empezar a dar señales de vida con la llegada de una nueva generación de aparatos y el desarrollo por parte de compañías de juegos de títulos de calidad para la pequeña pantalla.

Este mismo año, el gigante de los juegos Electronic Arts dijo que reforzaría su producción de juegos para móviles el próximo año, sacando cuatro números uno en ventas como "Fifa Football" y "The Sims" para aparatos móviles.

Sus rivales Eidos, Ubisoft y THQ han estado también invirtiendo cada vez más dinero en el prometedor mercado.

1.3 Situación de España

España, salvo algunas excepciones, siempre ha sido un país bastante atrasado en la creación de videojuegos. Sin embargo, debido a que la creación de videojuegos para móviles no necesita de los grandes recursos que requieren los videojuegos para PC o consola, están empezando a aparecer empresas que crean y distribuyen sus videojuegos dentro y fuera de nuestras fronteras. Claros ejemplos de este tipo de empresas son Gaelco Móviles (www.gaelcomoviles.com), MicroJocs (www.microjocs.com), o GameLoft Ibérica (www.gameloft.com), entre otras.

Aún así, es posible que la situación pudiera ampliarse todavía más si se dispusiera de emprendedores y profesionales suficientemente formados para tal tarea.

1.4 ¿Qué hace la universidad al respecto?

Tradicionalmente, el desarrollo de videojuegos ha sido considerado como algo poco

serio; más un *hobbie* de informáticos que una profesión real. Es por ello que dicho campo está prácticamente olvidado en los planes de estudio de las ingenierías. Esto provoca un déficit de buenos profesionales a la hora de cubrir los puestos de trabajo que, si no cambia, hará que nuestro país pierda la oportunidad de posicionarse internacionalmente ante una nueva, rica y creciente industria.

Aún así, en los últimos años han empezado a surgir asignaturas y cursos en la propia universidad. He aquí algunos:

- *Curso de programación de videojuegos con J2ME y MIDP para móviles*, curso de 120 horas impartido por la Universidad de Salamanca a través de Internet.
- *Curso de programación de videojuegos para PC*, curso semi-presencial de 90 horas impartido por la Universidad de Salamanca.
- *Master en creación de videojuegos*, de 400 horas, impartido por el Institut de Formació Contínua de Barcelona y la Universitat Pompeu Fabra.
- *Master en diseño y producción de videojuegos*, impartido por la Universidad Europea de Madrid.
- *Curso de desarrollo de juegos por ordenador*, de 100 horas, impartido por la Universidad Complutense de Madrid.
- *Diseño y creación de videojuegos*, asignatura de la Universidad de Oviedo.
- *Graduado Multimedia*, diplomatura de título propio impartida por la Universitat Oberta de Catalunya y la Universitat Politècnica de Catalunya.

Tal y como se observa, apenas hay en la universidad española asignaturas que permitan aprender las técnicas de programación de videojuegos. El ingeniero o alumno que desee introducirse o profundizar en este campo, deberá hacerlo mediante cursillos o masters respectivamente, si tiene la suerte de que éstos se impartan en alguna universidad cercana a él.

Otro dato a tener en cuenta es que estos cursos están impartidos principalmente por profesores universitarios, con una escasa presencia de profesionales del mundo empresarial. Considero esto como un grave error, ya que mientras la universidad no se implique más en el estudio y la enseñanza de los videojuegos, este campo seguirá estando principalmente dominado por el al mundo de la empresa, donde los profesionales con experiencia son quienes más puedan aportar a la enseñanza de los conceptos

relacionados con la creación de videojuegos.

1.5 Conclusión

Considerando lo expuesto en este capítulo de introducción, este proyecto nace con la finalidad de dar un pequeño paso más en la introducción del desarrollo de videojuegos en el mundo académico.

Todo el contenido nace de la investigación de un alumno de ingeniería informática en las técnicas y métodos para crear videojuegos, desde un punto de vista principalmente dedicado a la ingeniería del software, recogiendo información de programadores con experiencia e intentando aportar nuevas ideas y mejoras. No se quiere profundizar en el diseño gráfico o sonoro, perteneciente a ramas artísticas, ni en el tema de la computación gráfica, debido a que en la universidad española ya existen departamentos que imparten asignaturas de calidad sobre este campo.

Parte I. Java 2, Micro Edition (J2ME)

Capítulo 2. Sistemas de desarrollo y ejecución de aplicaciones

El siguiente capítulo es un breve compendio sobre los diferentes métodos de los que se dispone hoy en día para el desarrollo y ejecución de aplicaciones sobre aparatos de telefonía móvil.

2.1 Reprogramación de la ROM del terminal

Este es método a utilizar para los teléfonos móviles más antiguos, los cuales constan de un sencillo sistema operativo y diversas aplicaciones encastadas en una ROM. En la mayoría de los casos, las herramientas software necesarias no son más que un ensamblador y un compilador cruzado.

Este es un método complicado, ya que es necesario conocer a fondo el terminal que se está programando y las compañías no suelen facilitar datos técnicos sobre el hardware de los terminales. Además, un simple error puede estropear irreversiblemente el terminal, haciendo incluso que éste pierda la garantía del fabricante. Por todos estos motivos, este método queda totalmente descartado.

2.2 Programación en lenguajes compilados con las API del sistema operativo

Los teléfonos móviles actuales poseen un sistema operativo medianamente sofisticado, con el que se puede acceder a todas las funciones del terminal mediante unas API proporcionadas por el mismo fabricante. Además, suelen disponer de funciones para la carga sencilla de aplicaciones externas desde un PC doméstico o desde una red. Para crear aplicaciones con este método, las herramientas necesarias son un compilador cruzado que soporte el sistema operativo y el microprocesador del teléfono móvil, y las API de dicho sistema. Estas herramientas suelen ser proporcionadas por el fabricante, en muchos casos de manera gratuita.

El diagrama de la figura 2.2 muestra, de manera simplificada, la estructura mediante la cual funcionaría un programa creado según el presente método.

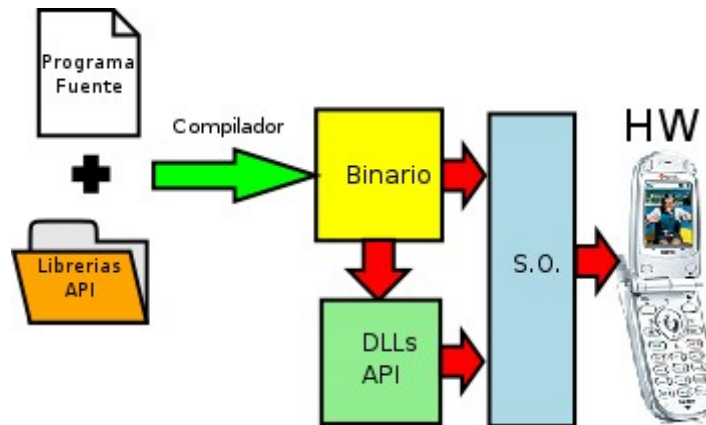


Figura 2.1: compilación y ejecución de un programa para un sistema operativo, un hardware y unas API determinadas.

Son varias las ventajas que pueden decantarnos a realizar aplicaciones mediante esta metodología:

- Sencillez de programación, debido a las facilidades de los lenguajes de alto nivel y la extensa documentación que los fabricantes suelen proporcionar sobre las APIs.
- Al ser un lenguaje compilado, podremos acceder a todas las funciones del dispositivo móvil obteniendo un rendimiento muy alto.

Sin embargo, también hay inconvenientes que hacen que esta metodología pueda no ser la más adecuada para nuestra aplicación:

- No todos los dispositivos móviles disponen de herramientas y librerías para ser programados a tan bajo nivel. Muchos fabricantes actuales se decantan por dar soporte a aplicaciones Java en los móviles y descartan que puedan ser programados a un nivel más bajo de la máquina virtual.
- Las API con las que se programa suelen ser únicamente válidas para una familia reducida de terminales, siendo totalmente necesario reescribir gran parte del programa si se desea exportar a otro modelo o familia de terminales.

2.3 Programación en J2ME

Actualmente, la gran mayoría de modelos que se fabrican incluyen soporte para tecnología Java 2 Micro Edition (J2ME). Este es una versión del lenguaje Java con ciertas limitaciones para poder ser ejecutado en un dispositivo pequeño (y por tanto, menos potente), y algunas librerías añadidas para poder acceder a las funciones de telefonía móvil. Todo esto se ejecuta sobre una máquina virtual adaptada a estas características. En capítulos posteriores se describen las características más importantes de J2ME.

En este capítulo, clasificaremos las posibilidades de desarrollo de aplicaciones J2ME en dos distintas: utilizando únicamente las librerías estándar de J2ME (MIDP, CLDC), o utilizando librerías estándar con librerías específicas del dispositivo. A continuación serán descritas y estudiados sus pros y sus contras. Una vez hecho esto, se propondrá una tercera posibilidad: una posibilidad mixta, que intente aprovechar las ventajas y minimizar los problemas de ambos métodos de desarrollo.

2.3.1 Programación mediante librerías estándar de J2ME

Utilizando sólo los paquetes y librerías estándar de Java, podemos crear aplicaciones que funcionen en cualquier dispositivo con una configuración determinada. En la figura 2.1 se representa la estructura de un dispositivo móvil ejecutando aplicaciones con este método.

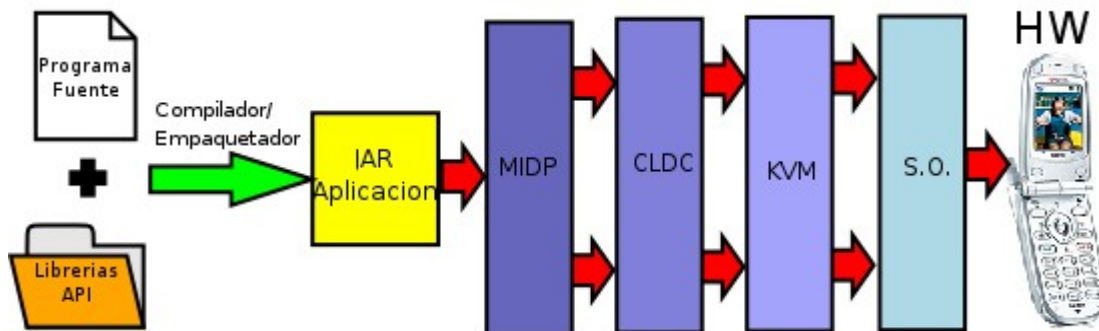


Figura 2.2: ejecución de una aplicación J2ME estándar sobre un dispositivo móvil.

Esta configuración tiene la ventaja de que una vez creada una aplicación para una configuración determinada de móvil (especificada en el archivo *manifest.mf*), ésta puede ejecutarse en cualquier dispositivo que soporte dicha configuración. Sin embargo, son varias las desventajas de usar Java estándar:

- Si se compara la figura 2.1 con la figura 2.2, se podrá observar que entre el programa compilado y el hardware, ahora hay una capa de software más gruesa. Si a esto le añadimos las características de interpretación del bytecode Java, se obtiene una aplicación más lenta que si estuviera programada en C++ (según algunos autores, entre 5 y 20 veces más lenta).
- La tecnología de los aparatos de telefonía aumenta mucho más deprisa que las versiones de J2ME. Entonces, programando con J2ME estándar no es posible acceder a todas las capacidades multimedia y de conectividad de los terminales más modernos. Para subsanar esto, apareció la versión 2.0 del MIDP. Esto subsana en parte este problema, proporcionando nuevas capacidades multimedia y

facilidades para la creación de videojuegos; el problema es que, en el momento de ser realizado este proyecto, el porcentaje de teléfonos con MIDP 2.0 es escaso. Además, MIDP 2.0 se desfasará con el tiempo, y será necesario un MIDP 3.0, etc...

2.3.2 Programación con J2ME estándar + API del fabricante

Con este método (ver figura 2.3) se subsanan los problemas de aprovechamiento del hardware comentados en el apartado anterior.

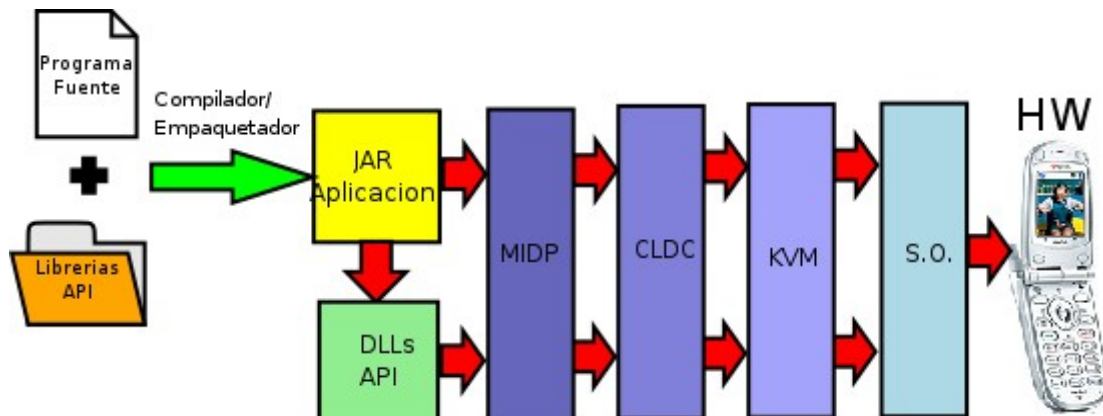


Figura 2.3: ejecución de una aplicación J2ME utilizando las API proporcionadas por el fabricante

Ahora se consigue acceder a todas las funcionalidades del hardware que el fabricante especifique en las API, pero se vuelve de manera un tanto estúpida a la siguiente situación: se acaba con la compatibilidad entre dispositivos de diferentes familias. Por tanto, ¿de qué sirve programar en un lenguaje lento como es Java, si su principal ventaja es la compatibilidad, y de esta manera queda eliminada? Con este método, cada vez que se quiera exportar la aplicación a una familia diferente, deberemos revisar todo el código y cambiarlo para hacerlo compatible con las nuevas API.

En este proyecto se propone una tercera alternativa para la creación de aplicaciones que intente minimizar el problema comentado anteriormente.

2.3.3 Creación de interfaces entre el programa y las API

A menudo, los terminales de diferentes familias y compañías incluyen funcionalidades similares, aunque a la hora de programar se utilicen de manera diferente. Aprovechando esta característica, se propone crear un paquete que haga de interfaz entre el programa y las librerías a la hora de programar. Este paquete podrá ser aprovechado en futuras creaciones.

De esta manera, se puede acceder a las funcionalidades comunes avanzadas que

J2ME no define y, cada vez que se quiera crear una versión del software para otra familia o fabricante, tan sólo hay que crear un paquete “interfaz” nuevo y recompilarlo. Esto no acaba totalmente con el problema de la compatibilidad, pero permite acceder a funciones avanzadas del hardware y se elimina gran parte del coste de crear aplicaciones para el mayor numero de terminales posibles. La figura 2.4 muestra de manera simplificada la metodología a seguir.

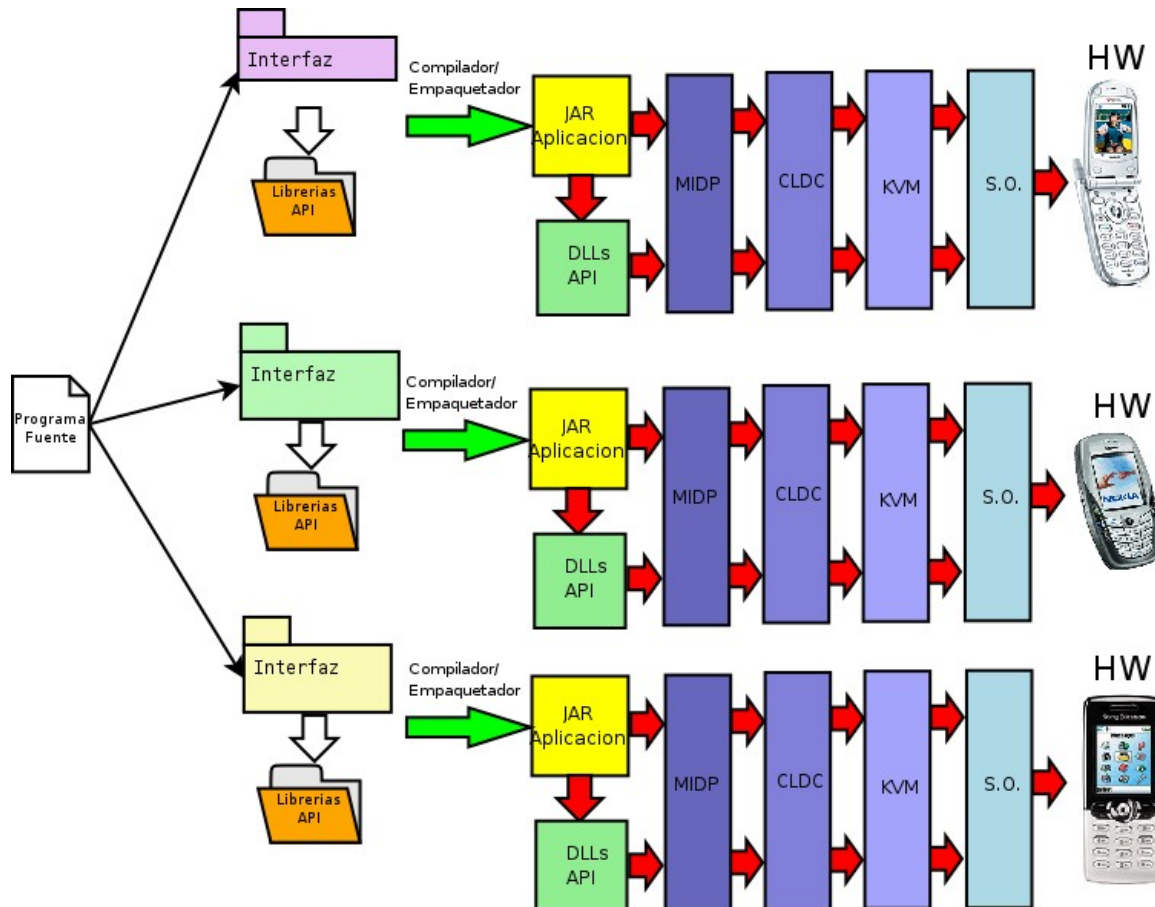


Figura 2.4: metodología de desarrollo de aplicaciones multiplataforma utilizando las API de los fabricantes

2.4 Conclusiones

Descartando la primera alternativa (reprogramación de la ROM), no se puede considerar que ninguna de las opciones propuestas sea mejor que las demás. Todo es cuestión de estudiar caso a caso qué aplicaciones se van a desarrollar y sobre qué dispositivos deben funcionar. A continuación se describen qué casos pueden ser adecuados para cada metodología:

- Desarrollo en C++ o Java con las API del fabricante

- Fabricación de aplicaciones a medida para un dispositivo concreto que requieran un alto aprovechamiento de la máquina.
- Fabricación de aplicaciones comerciales para un dispositivo concreto, como por ejemplo, una compañía que obtiene una licencia de desarrollo para la consola n-Gage y desarrolla un producto que a priori sólo va a ser ejecutado en ese dispositivo.
- ➔ Esta metodología es adecuada para este tipo de aplicaciones, ya que se considera que la portabilidad es una característica secundaria y queremos obtener el máximo rendimiento.
- Desarrollo en J2ME estándar
 - Aplicaciones que no requieran un gran aprovechamiento de las capacidades del dispositivo.
 - Aplicaciones que requieran una portabilidad total de un dispositivo a otro.
- ➔ En este caso, y en el momento de ser escrito este documento, se recomienda la utilización de MIDP 1.0, ya que la versión 2.0 es aceptada por un porcentaje todavía pequeño de dispositivos J2ME.
- Desarrollo con J2ME+API utilizando interfaces
 - Aplicaciones con características multimedia y de conectividad avanzadas, que requieran características multimedia y que puedan ser exportadas a el mayor número posible de dispositivos.
- ➔ El mejor ejemplo para esto podría ser el de una compañía de videojuegos para teléfonos móviles, a la cual le interesa que el producto sea de buena calidad y que pueda ser vendido al mayor número de personas, las cuales, obviamente, poseerán dispositivos de familias diferentes. Una vez compiladas las aplicaciones para los distintos modelos y familias, el usuario sólo debería especificar su modelo de teléfono y obtener el videojuego totalmente compatible.

Capítulo 3. Introducción a J2ME

3.1 Introducción

La plataforma Java 2 Micro Edition (J2ME) proporciona un entorno robusto y flexible para aplicaciones ejecutadas en dispositivos de escasos recursos, como teléfonos móviles, PDAs, u otros sistemas embebidos. Al igual que el resto de plataformas Java, J2ME incorpora una máquina virtual de Java y un conjunto de APIs estándar definidas por expertos de la comunidad Java, como pueden ser desarrolladores de software, hardware, o proveedores de servicios.

J2ME dota a los sistemas embebidos de las características de Java: portabilidad, interfaces de usuario flexibles, un robusto sistema de seguridad, protocolos de red, y soporte para todo tipo de aplicaciones, que pueden ser descargadas dinámicamente. Las aplicaciones basadas en las especificaciones J2ME se escriben una sola vez para ser ejecutadas en un amplio rango de dispositivos, utilizando las características particulares de cada uno.

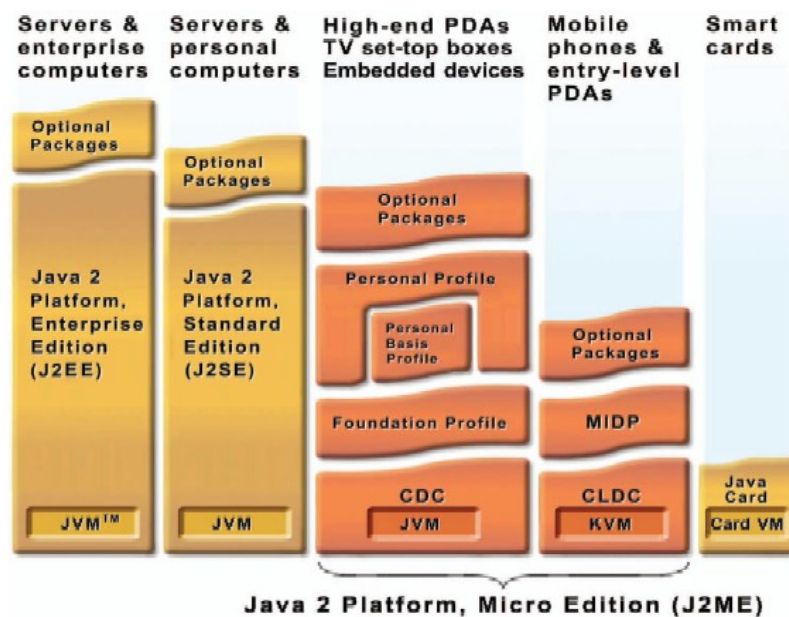


Figura 3.1: J2ME dentro del resto de tecnologías Java (fuente: Sun)

Hoy en día, se podría considerar que es la tecnología de aplicaciones móviles líder del mundo, en cuanto a dispositivos que la utilizan: prácticamente el 100% de los dispositivos móviles de fabricación reciente.

3.2 La arquitectura J2ME

Comprende una variedad de configuraciones, perfiles y paquetes opcionales para elegir por los desarrolladores. Esto permite construir un entorno Java adecuado a los requerimientos de la aplicación y características del dispositivo. Cada combinación está optimizada para un determinado ahorro de memoria, potencia de cálculo y capacidad de entrada/salida, según la categoría del dispositivo. El resultado es una plataforma común que intenta aprovechar todo lo posible las características de cada dispositivo.

3.3 Configuraciones

Las configuraciones están formadas por una máquina virtual y un conjunto mínimo de librerías. Éstas proporcionan la funcionalidad base para un rango particular de dispositivos de similares características. Actualmente hay dos configuraciones:

- Connected Limited Device Configuration (CLDC): orientada a dispositivos pequeños, como teléfonos móviles y PDA. Éstos presentan restricciones en cuanto a características hardware, pero lo más importante es su reducida conectividad, ya que poseen una conexión inalámbrica restringida (de ahí lo de *Limited*), de baja velocidad e intermitente. Este perfil hace uso de la KVM debido a las restricciones de memoria que presentan los dispositivos.
- Connected Device Configuration (CDC): basada en la Máquina Virtual de Java clásica. Apta para dispositivos con conectividad de red permanente y robusta, y sin las limitaciones de tamaño y prestaciones de los dispositivos pequeños. Se asume un mínimo de 512Kb de memoria ROM y 256Kb de RAM. Dispositivos que cumplen esta configuración son los Sistemas de Navegación y PDA de gama alta, entre otros.

3.4 Perfiles

Una configuración debe combinarse con un perfil para proporcionar un entorno de ejecución para una categoría de dispositivos específica. Un perfil es un conjunto de APIs de alto nivel que definen aspectos como el ciclo de vida de la aplicación, las interfaces de usuario, y las propiedades de acceso específicas para el dispositivo. Los perfiles son los siguientes:

- Foundation profile (FP): es el perfil de más bajo nivel. Proporciona una

implementación con capacidad de conexión a red, y es usado por sistemas encastados que no necesiten interfaz de usuario. Para aplicaciones que requieran de interfaces, se puede combinar con *Personal Basis Profile*, explicado más adelante.

- *Mobile Information Device Profile (MIDP)*: Diseñado para teléfonos móviles y PDA de gama baja. Ofrece la funcionalidad básica que requieren las aplicaciones móviles, incluyendo interfaz de usuario, conectividad en red, almacenamiento local de datos y administración de aplicaciones. Combinado con CLDC, proporciona un entorno de ejecución completo, aprovechando las capacidades de los dispositivos de mano, y minimizando tanto memoria como consumo de energía.
- *Personal Profile (PP)*: utilizado en dispositivos que necesiten interfaces gráficas de usuario o soporte para applets, como PDAs de gama alta o consolas de videojuegos. Incluye la librería AWT de Java y soporta aplicaciones y applets diseñados para ejecutarse en entornos de ventanas.
- *Personal Basis Profile (PBP)*: es un subconjunto de PP, que proporciona un entorno de aplicación para dispositivos en red que requieran de un nivel básico de presentación gráfica o que requiera el uso de herramientas gráficas especializadas para las aplicaciones específicas, como puedan ser televisores, ordenadores de a bordo, o pantallas de información. Tanto PP como PBP están situadas como capas encima de FP.

3.5 Paquetes Opcionales

La plataforma J2ME puede expandirse mediante paquetes opcionales. Éstos ofrecen APIs estándar para usar tecnologías como conectividad con bases de datos, mensajería wireless, multimedia, Bluetooth, servicios web, etc.

3.6 Características de J2ME frente a J2SE

Las novedades más importantes que incluye J2ME frente a la edición estándar de Java (J2SE) son las siguientes:

- *Tipos de datos*: en muchos casos, se limitan o eliminan tipos de datos como *float* o *double*, debido al alto coste de memoria y proceso que estos requieren. Cada configuración de terminal dispone de una lista de tipos soportados y cómo se

comportan respecto a la implementación original.

- Preverificación de código: en J2SE, el código se verifica en tiempo de ejecución. Es decir, mientras el programa se ejecuta, la JVM (*Java Virtual Machine*) verifica el tipo de los objetos a los que accedemos, que no accedamos a objetos *null*, que se accede correctamente a los elementos de un *array*, y un largo etcétera... De esta manera, podemos ejecutar código de manera segura. Por cuestiones de rendimiento, en J2SE se ha dividido en dos etapas esta verificación: etapa de preverificación, que se realiza en tiempo de compilación, y verificación online, que se realiza en tiempo de ejecución. De esta manera, se aumenta la velocidad de ejecución de los programas, pero se rebajan las condiciones de seguridad en las que se trabaja.
- Librerías gráficas: se han creado unas librerías gráficas más sencillas, adaptadas al dispositivo móvil. Ha cambiado la clase *Graphics*, i han desaparecido AWT y Swing para utilizar una librería de componentes específicos para aplicaciones sobre terminales móviles.
- No existencia de punto de entrada a la aplicación: la JVM de J2SE utilizaba el método *main* como punto de entrada a la aplicación y así ceder el control de la máquina a ésta. Las aplicaciones J2ME definen unas determinadas funciones, las cuales son llamadas por el dispositivo móvil, para cederles el control de la máquina en momentos puntuales. Con esto se consigue, por ejemplo, que cuando recibamos una llamada al teléfono mientras se utiliza una aplicación, se guarde el estado de ésta, para poder reanudar posteriormente la aplicación por el punto donde se dejó sin perder datos ni trabajo hecho.
- API básica: ésta está muy limitada, debido a las restricciones de memoria y potencia de los dispositivos Java. Así, encontramos que algunas clases como *Math* carecen de la inmensa mayoría de funciones que posee en J2SE o J2EE. Incluso la clase *Object* es inferior a la de las versiones *Standard* y *Enterprise* de Java.

Como apunte, cabe destacar que, en contra de lo que se afirma en varios libros (mirar referencias bibliográficas [García 2003] y [Varios 2003]), J2ME **SÍ** posee recolector de basura. Tan sólo hay que observar 3 hechos:

- Ausencia de algún mecanismo explícito de destrucción para los objetos.

- Los *logs* del entorno de desarrollo y ejecución donde muestran las recolecciones de basura realizadas durante la ejecución.
- Se puede llamar explícitamente al recolector de basura mediante el comando *System.gc()*.

3.7 Descriptor, Manifiesto y MIDlet

Son tres elementos importantes de cara a la carga y ejecución de una aplicación J2ME. El descriptor (*.jad) y el manifiesto (Manifest.mf) son ficheros de texto que aportan información acerca de la aplicación, tales como la Configuración o el perfil requeridos. El MIDlet (*.jar) contiene empaquetada la aplicación en sí, junto con el fichero de manifiesto.

Las características básicas de descriptor y manifiesto son :

- Manifiesto: incluye información sobre la aplicación como es el nombre, fabricante, o la versión.
- Descriptor: informa al programa que se encarga de gestionar las descarga, instalación y gestión de las aplicaciones sobre las características de éstas, para que el gestor pueda asegurarse que el MIDlet es adecuado para el dispositivo en concreto. No se incluye con el paquete, sino que se consulta antes de descargar la aplicación.

Debido a que las herramientas actuales se encargan de generar automáticamente estos tres elementos y, además, permiten gestionar las características de éstos mediante una interfaz gráfica, en este proyecto no se entrará en explicar la estructura y funcionamiento de cada uno.

Capítulo 4. Entornos de desarrollo

Un buen entorno de desarrollo agiliza la velocidad con la que se programa. Para grandes proyectos, no basta con el típico editor de texto que resalta el léxico del lenguaje para facilitar su lectura. Es por ello que la elección de un buen entorno, aunque pueda suponer un gasto económico considerable, acelera drásticamente el proceso de creación, repercutiendo en un menor coste económico a la hora de crear proyectos.

En el siguiente capítulo se estudian algunos de los más importantes entornos de desarrollo disponibles para Java; sus pros y sus contras. Al finalizar se explicarán los motivos por los que se ha escogido para el proyecto un entorno en concreto.

Cabe remarcar que todos los entornos aquí expuestos son multiplataforma. Todos están disponibles para Windows y Linux, y la mayoría de ellos soportan también otros sistemas menos extendidos como Mac OS, Solaris, FreeBSD, HP-UX, AIX, y otros “sabores” UNIX.

4.1 IntelliJ IDEA

Sin duda alguna, el mejor entorno de desarrollo para Java. Las características para realizar esta afirmación son las siguientes:

- Interfaz de usuario muy intuitiva y cuidada.
- Proceso de navegación entre archivos, clases, métodos y atributos muy rápido (ver un ejemplo en la figura 4.1).
- Generación automática y configurable de código, con lo que pulsando un par de teclas se pueden generar estructuras de control de flujo, implementar métodos de interfaces, sobrecargar métodos de funciones padre, etc.
- *Code completion*: con sólo escribir las primeras letras de un elemento, emerge una ventana con todas las posibilidades, pudiendo mostrar sólo las más adecuadas. Por ejemplo, para una asignación a un *String*, podemos mostrar sólo los métodos de una clase que devuelvan *String*. Con escribir el nombre de una clase, indica el paquete donde se sitúa y lo incluye al código automáticamente. Estas son sólo algunas de sus innumerables características, que ahorran al programador el esfuerzo de memorizar todos los paquetes y las clases, o que haya que consultar continuamente la documentación de los paquetes. Ver un ejemplo en la figura 4.2.

- El *debugger* es intuitivo y permite realizar el proceso de depuración de programas ejecutándose en aplicaciones externas. Esto es importante de cara a la programación de programas que se ejecuten en servidores y, en el caso de J2ME, en un emulador externo al entorno.
- No es necesario compilar el código para ver los *warnings* y errores del compilador. Se muestran sobre el código escrito, en el punto exacto donde se produce. Además, da consejos sobre cómo resolverlos.
- Muchas otras características: integración con CVS, soporte XML, refactorización, y un largo etc...

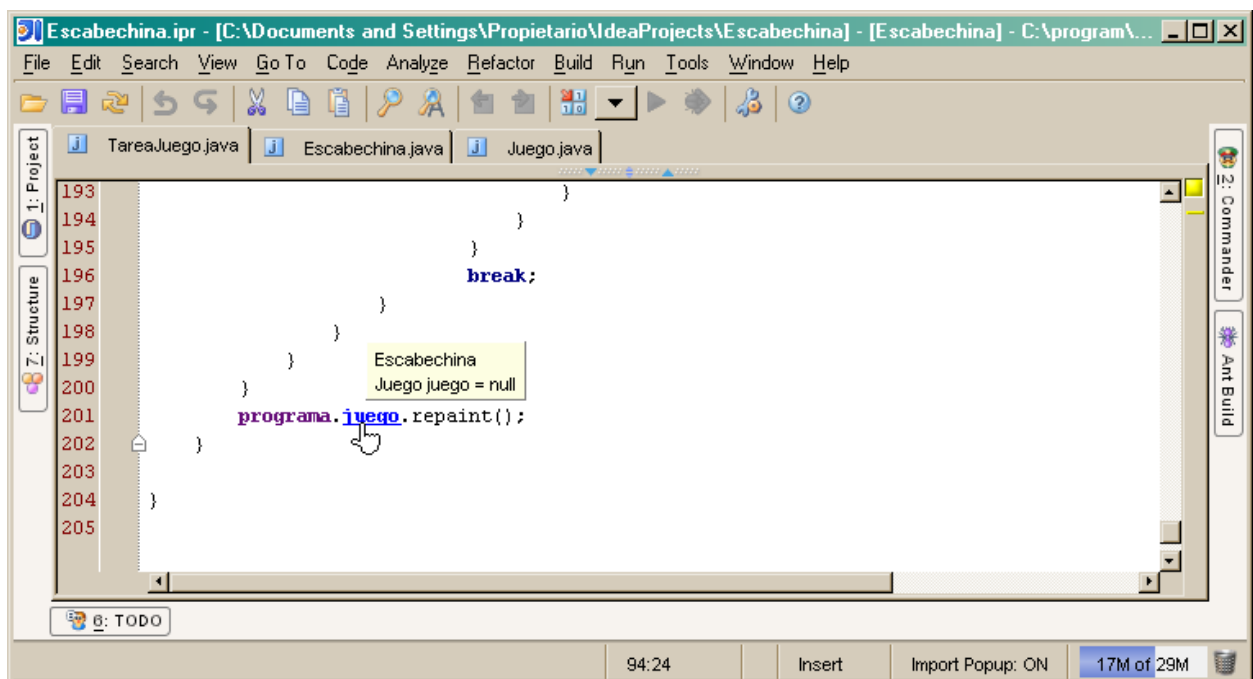


Figura 4.1: posicionando el ratón sobre un elemento se pueden conocer datos acerca de él. Con CTRL +Click se navega hacia la definición o la clase contenedora

Sin embargo, no todo son ventajas. A continuación se describen los problemas encontrados:

- Al estar hecho 100% en Java, consume muchos recursos y en ocasiones puede resultar un poco lento. De todas formas, es bastante más rápido que otros entornos desarrollados 100% en Java (por ejemplo NetBeans).
- Precio. Una licencia de NetBeans cuesta 500\$, haciéndolo desaconsejable para un proyecto educativo puntual como este; aunque una licencia educativa cuesta \$199. La licencia incluye soporte técnico ilimitado vía e-mail, actualizaciones gratuitas y un

40% de descuento en la compra de futuras versiones.

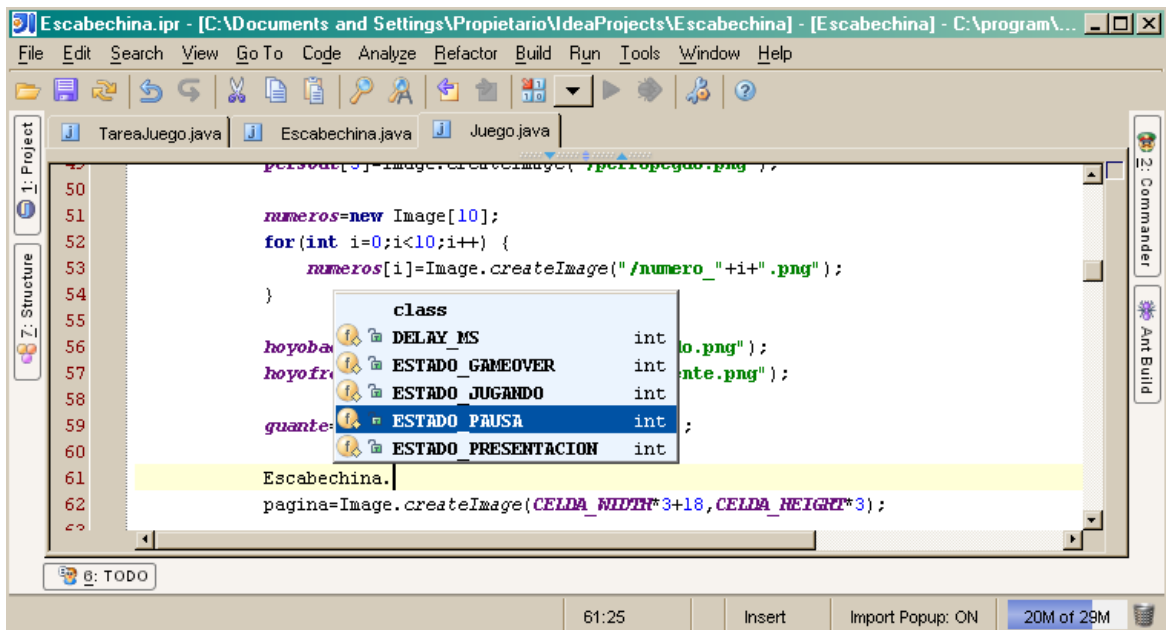


Figura 4.2: con CTRL+Espacio se muestra una ventana emergente con información acerca de el código que puede escribirse en ese lugar

Como apunte final, hay un *plugin* para poder crear, compilar y ejecutar programas J2ME en el entorno de IntelliJ IDEA, aunque es un proyecto poco maduro, lleno de errores y con pocas características interesantes. No obstante, a fecha de Junio de 2005, está anunciada la versión 5.0 de IDEA, la cual incorpora soporte para J2ME.

En conclusión, estamos ante el mejor entorno de programación para Java. No obstante, debido al precio de la licencia, éste no es un entorno adecuado para un proyecto académico, sobretodo habiendo como hay excelentes buenas alternativas libres, a precio 0.

4.2 NetBeans IDE

Entorno de desarrollo completo, con características similares a IntelliJ IDEA. Sin embargo, al igual que IDEA está escrito 100% en Java; el consumo de recursos es inmenso y la velocidad de ejecución puede llegar a ser desesperante. Ni siquiera en el equipo de desarrollo (un Pentium IV 2,66 Ghz con 512 MB RAM) se ha podido trabajar con comodidad, debido a los continuos tiempos de espera entre la acción del usuario y la respuesta del programa.

Como ventajas, decir que es gratuito y de código abierto, mediante la licencia *Sun*

Public License. Y que incorpora una total integración con las herramientas de desarrollo para J2ME (ver figura 4.3).

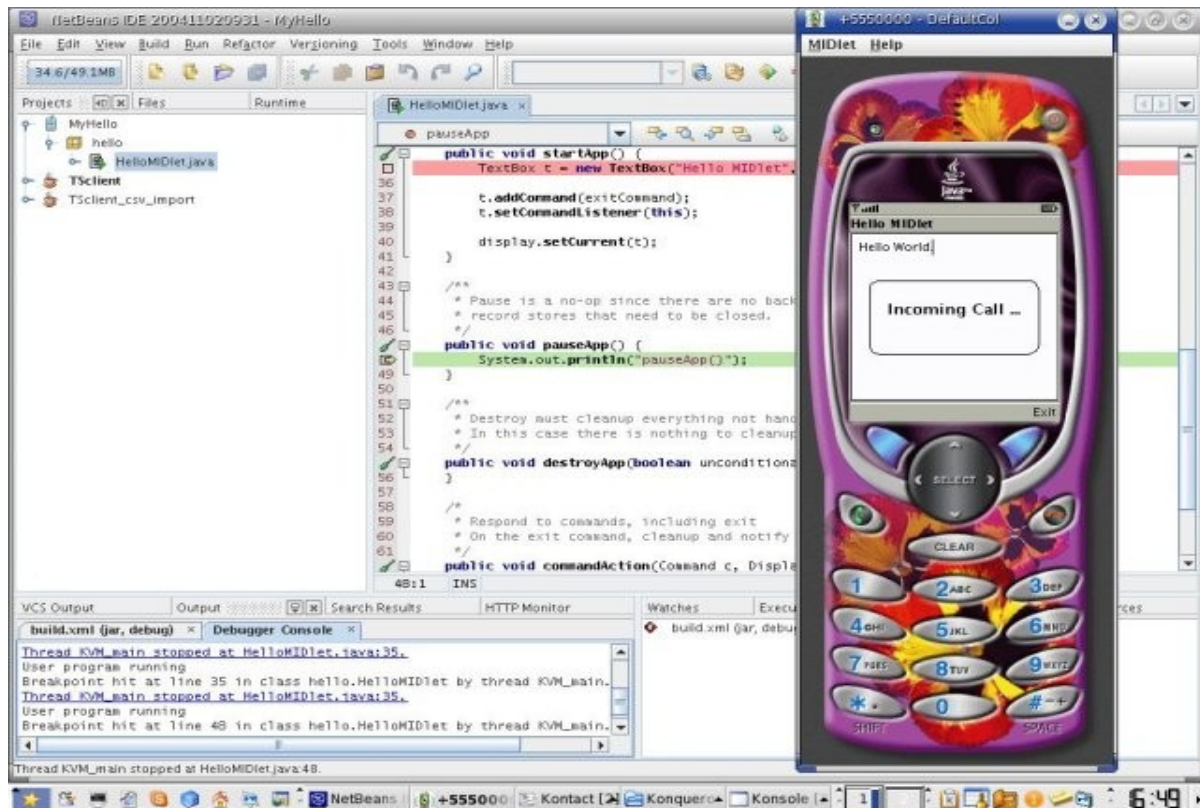


Figura 4.3: depurando una aplicación J2ME con NetBeans

4.3 Eclipse

Muy completo entorno de desarrollo respaldado por grandes empresas, entre ellas IBM. Ofrece gran cantidad de características de navegación, compilación, depurado, auto completado de código, etc. muy similares a IntelliJ IDEA (ver figura 4.5).

Sin embargo, la gran riqueza de Eclipse es que está ideado para ser muy modular, con lo que cualquiera puede añadir características fácilmente, mediante paquetes Java (ver figura 4.4). El resultado de esto es que hay disponibles cientos de *plugins*, algunos gratuitos, otros comerciales, que extienden las características de eclipse hasta donde el programador desee.

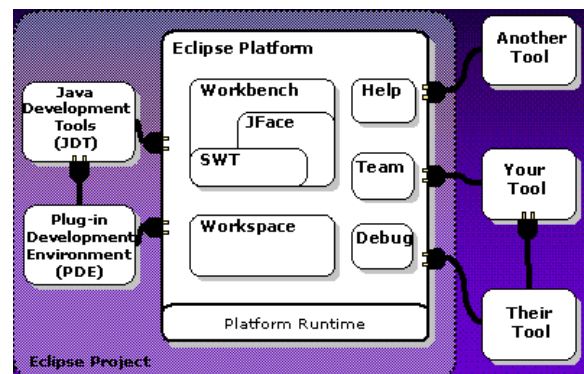


Figura 4.4: estructura de plugins de Eclipse (fuente: eclipse.org)

El núcleo está escrito en Java, pero la interfaz de usuario en C++. La consecuencia

de esto es que la velocidad es bastante buena y el consumo de recursos no llega a los extremos de los otros entornos de desarrollo. Además, está disponible para un gran número de sistemas operativos (Windows, Linux, MacOS, HP-UX, AIX, NetBSD, FreeBSD, y un largo etc.)

Eclipse está distribuido bajo la *Common Public License*, por lo que es gratuito y de código abierto. No así sus añadidos, gran parte de los cuales son de pago.

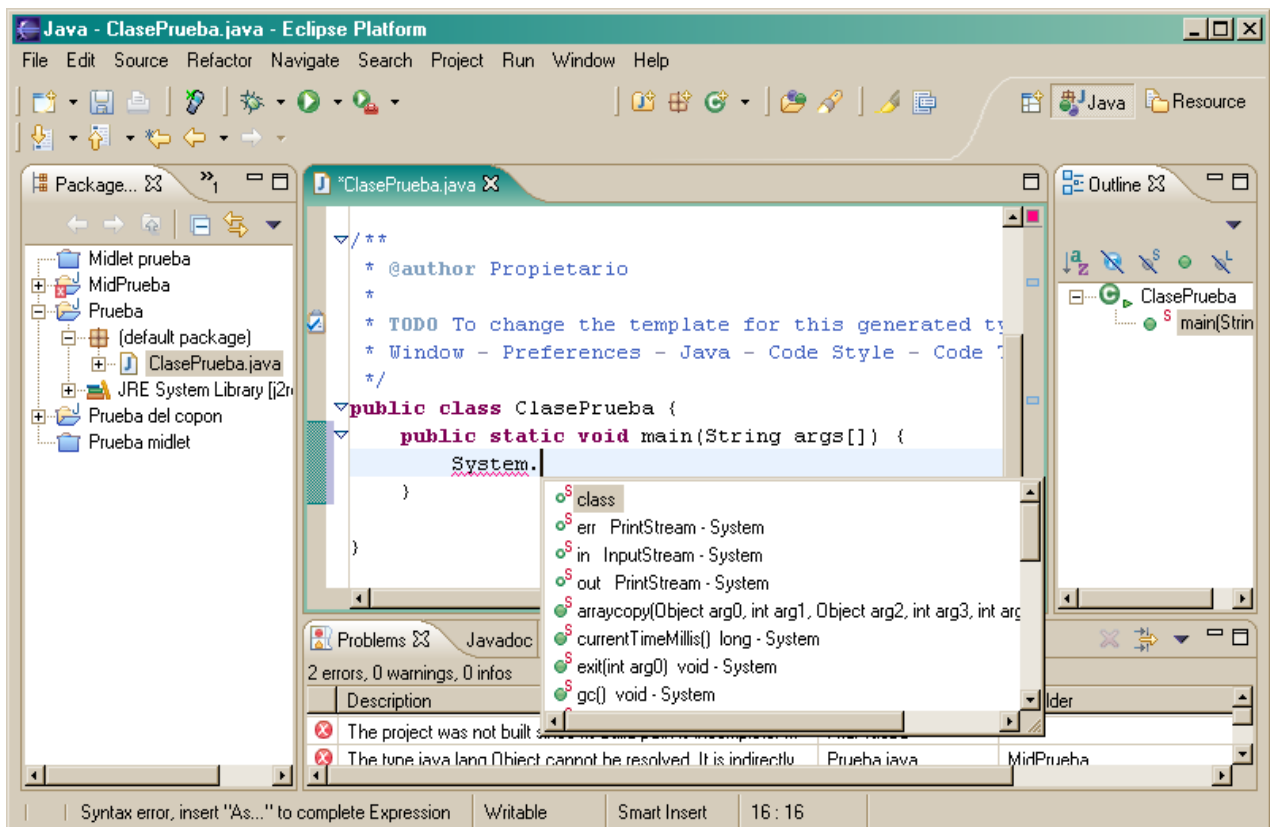


Figura 4.5: captura de Eclipse, donde se muestran características como el auto completado de código, navegación entre clases, etc...

En cuanto a la integración con J2ME, cabe decir que hay gran cantidad de *plugins* que permiten esto, pero el único gratuito, EclipseME, todavía está en un estado muy prematuro, con algunos errores y muestra cierta dificultad a la hora de ser utilizado cómodamente.

4.4 El elemento esencial: J2ME Wireless Toolkit (WTK)

Todas las funcionalidades J2ME de cualquiera de los entornos anteriormente citados dependen de este *kit* de herramientas proporcionado por *Sun Microsystems*. Entre las funcionalidades de éste, las más importantes son las siguientes:

- Compilador de Java para las APIs de J2ME, incluidas en el proyecto..
- Herramientas de creación y configuración automática de descriptor, manifiesto y MIDlet.
- *Debugger*, al cual conectar el entorno de desarrollo externo para depurar paso a paso la aplicación.
- Emuladores varios para probar las aplicaciones sin necesidad de descargarlas en un dispositivo físico (ver figura 4.6).

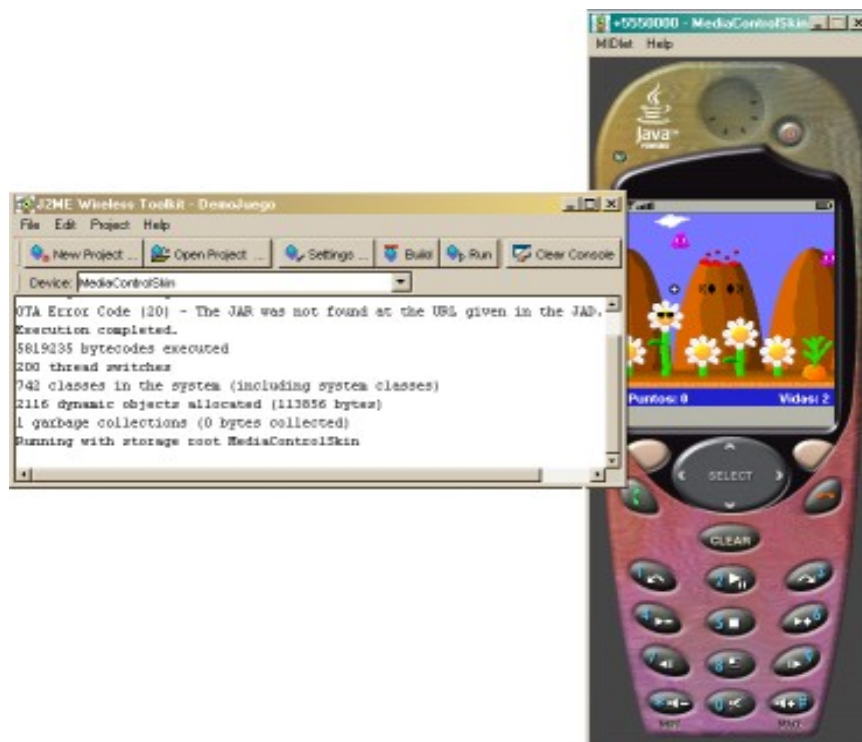


Figura 4.6: J2ME después de compilar y ejecutar en un emulador una aplicación

4.5 Conclusión

Para crear aplicaciones en J2ME, el único elemento realmente imprescindible es el *J2ME Wireless Toolkit*, ya que es el que proporciona las herramientas y librerías necesarias, sobre las que se apoyan los demás entornos de desarrollo. Con WTK y un simple editor de texto plano es posible desarrollar cualquier proyecto.

Sin embargo, debido a las ventajas de trabajar con un entorno integrado, este proyecto se realizará con Eclipse. Como el módulo de integración con J2ME, Eclipse ME, no está suficientemente maduro, se utilizará Eclipse tan sólo como editor y depurador del proyecto, y las tareas de compilación, ejecución y emulación, se realizarán directamente

sobre el entorno de WTK.

Debido a que Eclipse es sólo una ayuda no imprescindible y una elección personal en cuanto al programa para editar código, este proyecto no incorporará ningún tutorial sobre la utilización de Eclipse o cualquier otro entorno de desarrollo, y se asumirá que el código fuente se edita desde un editor de texto común. Sin embargo, el siguiente capítulo es un tutorial básico sobre *Java 2 Micro Edition Wireless Toolkit*, ya que esta herramienta es imprescindible a la hora de crear aplicaciones y es necesario conocer su funcionamiento.

Capítulo 5. Utilización de J2ME Wireless Toolkit (WTK)

En el siguiente capítulo se muestran las diferentes herramientas que J2ME WTK pone a disposición del programador para crear aplicaciones móviles. No se pretende profundizar mucho en el funcionamiento, tarea destinada a la documentación oficial de Sun (ver referencia bibliográfica [Sun 2004]), sino mostrar de manera rápida y sencilla cómo crear y testear aplicaciones con este entorno.

5.1 Default Device Selection

Esta sencilla herramienta nos permite seleccionar el dispositivo emulado sobre el cual testear las aplicaciones creadas. A lo largo del proyecto se utilizará la opción *MediaControlSkin*.



Figura 5.1: Default Device Selection

5.2 Preferences

Nos permite configurar características avanzadas del entorno de ejecución, tales como velocidad, seguridad, acceso a red, etc...

Durante este proyecto sólo se ha utilizado la función "Enable profiling", de la pestaña "Monitor" (figura 5.1), utilizada para comprobar los tiempos de ejecución de cada una de las funciones del programa ejecutado. Muy útil a la hora de intentar mejorar el rendimiento de las aplicaciones...

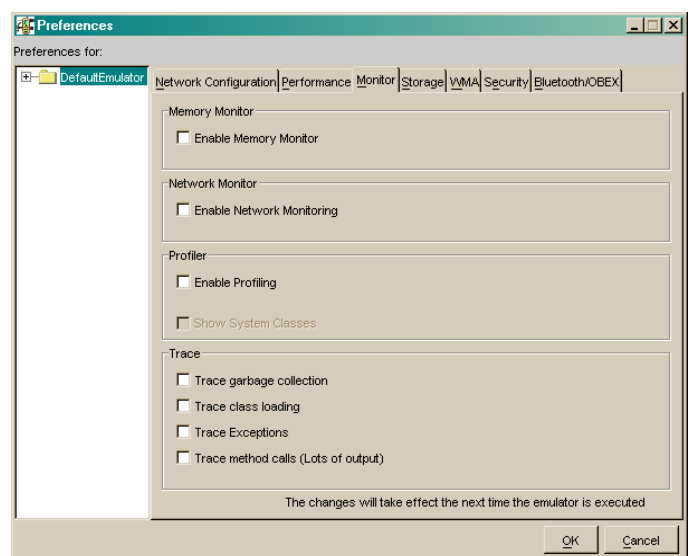


Figura 5.2: preferencias de WTK

5.3 Run MIDP application

Herramienta utilizada para ejecutar directamente en un emulador cualquier aplicación J2ME, seleccionando el archivo *.jad en disco.

5.4 KToolbar

La herramienta principal de la *suite*. Crea, gestiona, compila y ejecuta las aplicaciones J2ME.

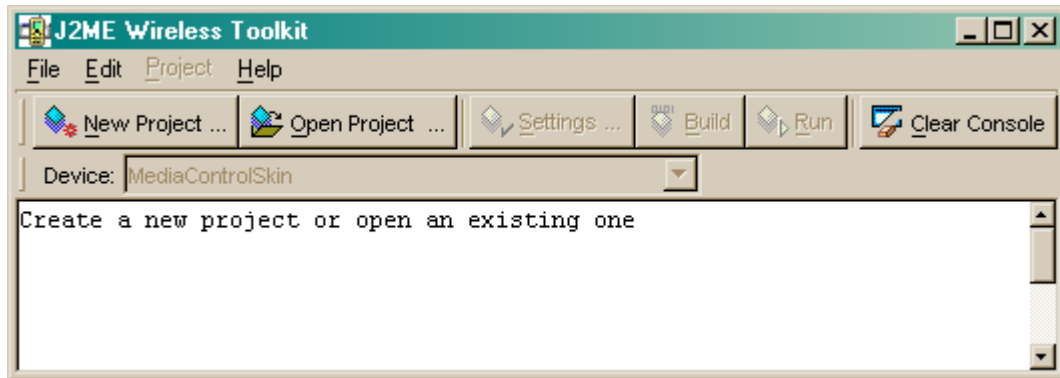


Figura 5.3: ventana principal de KToolbar

A continuación se describen las funciones más importantes y utilizadas durante la realización de este proyecto:

- New project: en el subdirectorio “apps” de la carpeta donde está guardado el WTK se crea un directorio con el nombre que se introduzca en el campo “Project Name”. Así mismo, en el campo “MIDlet Class Name” se debe introducir el nombre de la clase del MIDlet, con la ruta completa, en el formato de Java (ej: para la clase MIDletEjemplo que está en el subdirectorio java/ejemplo/utils, el nombre sería java.ejemplo.utils.MIDletEjemplo).
- Open Project: abre un proyecto ya existente en el directorio “apps” del directorio raíz del WTK.
- Settings: accede a las propiedades del proyecto ya creado. En este diálogo se puede elegir la configuración y el perfil de la aplicación, así como las API opcionales que utiliza (Mobile Media API, 3D API, Wireless API, etc...). También permite ver y editar el contenido de los archivos *Manifest.mf* y **.jad*.
- Build: compila el código fuente del proyecto.
- Run: ejecuta la aplicación en el emulador.

Parte II. Programación de videojuegos

Capítulo 6. Programación en J2ME

El presente capítulo pretende ser un tutorial básico acerca de la programación para J2ME. Al final del capítulo se expone una aplicación sencilla para ayudar a conocer el funcionamiento de la API de J2ME.

Debido al gran número de librerías de que dispone J2ME y al tamaño de éstas, sólo se explicará lo esencial para introducirse el funcionamiento de J2ME y desarrollar aplicaciones básicas. Ésto será suficiente para poder afrontar la programación de videojuegos de los siguientes temas.

Si se desea profundizar en el conocimiento de J2ME, se recomienda estudiar la documentación oficial de Sun (<http://java.sun.com/j2me/docs/>) y leer libros avanzados sobre J2ME, como por ejemplo la referencia bibliográfica [Ortiz 2001]. También es recomendable, mientras se va leyendo el capítulo, consultar la documentación de las API de J2ME, disponibles en el paquete de *Wireless Toolkit*.

6.1 Introducción a los *MIDlets*

Las aplicaciones de J2ME no son programas completos, sino partes que se desarrollan y ejecutan en un entorno limitado por la máquina virtual, el perfil y la configuración. Esto quiere decir que carecen de un punto de entrada como puede ser la función *main* de *Java 2, Standard Edition*.

6.1.1 Ciclo de vida de un *MIDlet*

En J2ME, será el *Java Application Management* (JAM) el que ceda en determinados momentos la ejecución a la aplicación: podrá decidir cuándo se crea o se destruye, además de poder sacar temporalmente la aplicación del estado de ejecución (pausarla). Esto lo hará llamando a los métodos *startApp()*, *pauseApp()*, y *destroyApp()* definidos en el *MIDlet*. En la figura 6.1 se muestra un diagrama de estados acerca del ciclo de vida de un *MIDlet*.

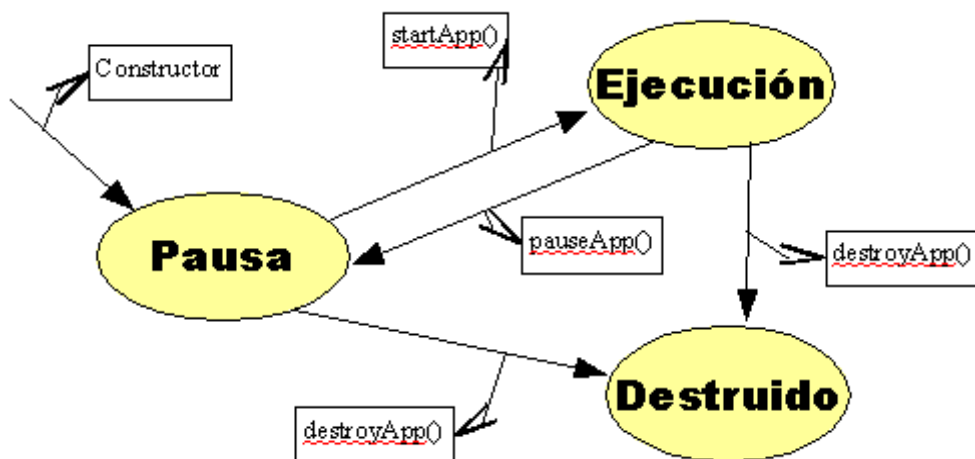


Figura 6.1 ciclo de vida de un MIDlet

6.2 Interfaces de usuario

Hay dos maneras de establecer la interacción entre la aplicación y el usuario:

- Interfaz de alto nivel: conjunto de clases proporcionadas por J2ME para proporcionar funcionalidades de entrada y salida de datos independientes del dispositivo. El programador sólo necesitará preocuparse de la gestión de los datos, ya que J2ME se encargará de dibujar en pantalla y obtener los datos de la manera más adecuada para el dispositivo donde se esté ejecutando la aplicación.
- Interfaz de bajo nivel: conjunto de clases para acceder directamente a los recursos del dispositivo (pantalla, teclado, etc...). El programador se encargará del dibujo en pantalla de dicha interfaz y de la gestión de los eventos del teclado, por lo que puede ser una tarea bastante complicada, debido a la gran variedad de pantallas, teclados, y dispositivos de entrada que existen. Como consecuencia, una interfaz programada a bajo nivel puede no funcionar correctamente en algunos dispositivos extraños e inusuales.

Todas las clases destinadas a las interfaces de usuario, tanto de alto como de bajo nivel, son subclases de *Displayable*. Ésta clase tiene dos clases derivadas: *Canvas*, para las interfaces de bajo nivel, y *Screen*, para las de alto nivel.

6.2.1 Interfaz de alto nivel

Se crea a partir de las clases derivadas de *Screen*:

- Alert: es una pantalla que muestra una información al usuario y espera un tiempo antes de mostrar el siguiente elemento *Displayable*.

- List: una pantalla que muestra una lista de opciones para elegir. Puede ser de tipo EXCLUSIVE o MULTIPLE, para selecciones individuales o múltiples respectivamente, o del tipo IMPLICIT, con la cual podremos definir un conjunto de acciones a realizar para el elemento seleccionado, mediante una lista de comandos.
- TextBox: es una pantalla que permite al usuario editar texto.
- Form: pantalla compuesta por un conjunto de elementos derivados de la clase *Item*, tales como imágenes, cajas de texto, agrupaciones de opciones, etc... A continuación se describen sus funciones:
 - ChoiceGroup: similar a la clase *List*. Implementan la misma interfaz: *Choice*. Su función es similar, *ChoiceGroup* puede introducirse en un formulario junto con otros elementos.
 - CustomItem: no tiene ninguna funcionalidad en concreto. Está creado para que mediante subclases de ésta, el programador pueda crear nuevos elementos de formulario.
 - DateItem: componente editable para presentar y manejar fechas y horas.
 - Gauge: componente gráfico que representa una barra de progreso. Dado un valor entre 0 y el valor máximo definido, dibuja una barra de tamaño proporcional a estos valores.
 - ImageItem: elemento que contiene una imagen.
 - Spacer: utilizado para crear separaciones entre elementos del formulario. No permite la interacción con el usuario.
 - StringItem: elemento que contiene una cadena de caracteres. Su finalidad es únicamente informativa, ya que no es editable por el usuario.
 - TextField: componente de texto editable. Similar a la clase *TextBox*, con la diferencia de que *TextField* está creada para ser introducida en un formulario y *TextBox* para ser visualizada a pantalla completa.

Hasta ahora, se han explicado los diferentes elementos que se pueden visualizar en pantalla. Ahora es necesario saber cómo puede interactuar el usuario con ellos.

A todos estos elementos se les pueden asociar comandos mediante la clase *Command*: una clase que guarda información semántica sobre una acción. Ésta puede

ser una etiqueta, un tipo de comando (OK, CANCEL, BACK, etc...) o un índice de prioridad. Mediante el método *addCommand()*, de las clases *Item* y *Displayable* se pueden asociar comandos y, cuando se ejecute un comando, será necesario tener asociada una clase que se encargue de recogerlo y tratarlo. Deberá asociarse una clase, creada por el programador, que implemente la interfaz *CommandListener*, y asociada al *Item* o *Displayable* mediante el método *setCommandListener()*. Ésta clase *CommandListener* tratará los comandos cuando se produzcan mediante el método *commandAction()*.

6.2.2 Interfaz de bajo nivel

Ésta se realiza a partir de la clase *Canvas*, cuya función principal es el dibujado en pantalla mediante la clase *Graphics*. Por otro lado, también gestiona diversos eventos relacionados con la entrada y salida, gestionados mediante la sobrecarga de los siguientes métodos de *Canvas*:

- showNotify: método llamado inmediatamente antes de que el objeto *Canvas* se muestre en pantalla. Puede sobrecargarse para inicializar algunos datos relacionados con la presentación del *Canvas* antes de que éste sea mostrado.
- hideNotify: llamado inmediatamente después de que el objeto *Canvas* salga de pantalla.
- keyPressed: llamado cuando el teclado registra la pulsación de una tecla.
- keyRepeated: llamado cada cierto intervalo de tiempo definido, si la tecla se mantiene pulsada.
- keyReleased: llamado al dejar de pulsar una tecla.
- pointerPressed: llamado cuando el dispositivo apuntador es pulsado.
- pointerReleased: llamado cuando el dispositivo apuntador deja de estar pulsado.
- pointerDragged: llamado cuando el dispositivo apuntador se mueve mientras se mantiene pulsado.
- paint: llamado cada vez que es necesario refrescar los datos gráficos de pantalla. Este método se encarga de dibujar en el *Canvas*, mediante un objeto de clase *Graphics*.

6.3 Un ejemplo práctico

Una vez explicada la estructura básica de un programa en J2ME, así como sus clases más importantes, se realizará una pequeña aplicación de muestra, con la cual aplicar algunos de los conceptos explicados en este tema.

La aplicación en sí constará de un sencillo *Canvas* sobre el que se dibujará un texto y un mapa de bits. Además, mediante menús y formularios podremos configurar los colores del *Canvas*, así como el texto a mostrar.

Además, se mostrará paso a paso cómo se crea el proyecto, se configura, se compila y ejecuta con WTK.

6.3.1 Creación y configuración del proyecto

1. Abrir J2ME, Wireless Toolkit
2. Hacer click sobre *New project*
3. Introducir, como nombre de la aplicación *"MIDlet prueba"*, y como nombre de la clase del MIDlet, *"Ejemplo"*
4. En el diálogo de configuración (ver figura 6.2), configurar la plataforma más sencilla. Escoger como *Target Platform* la opción *Custom*. Seleccionar los perfiles MIDP 1.0 y CLDC 1.0, desactivar *Wireless Messaging API* (opción *No WMA Support*) y dejar sin marcar todas las demás API opcionales.
5. En el directorio de instalación de WTK, subdirectorio *apps*, se habrá creado una carpeta llamada MIDlet prueba. A partir de ahora ese será el directorio del proyecto.

Ahora sólo queda editar el código de la aplicación: en el subdirectorio *src* se deberán crear archivos de texto pertenecientes a las clases. El nombre de estos será *<nombre_clase>.java*.

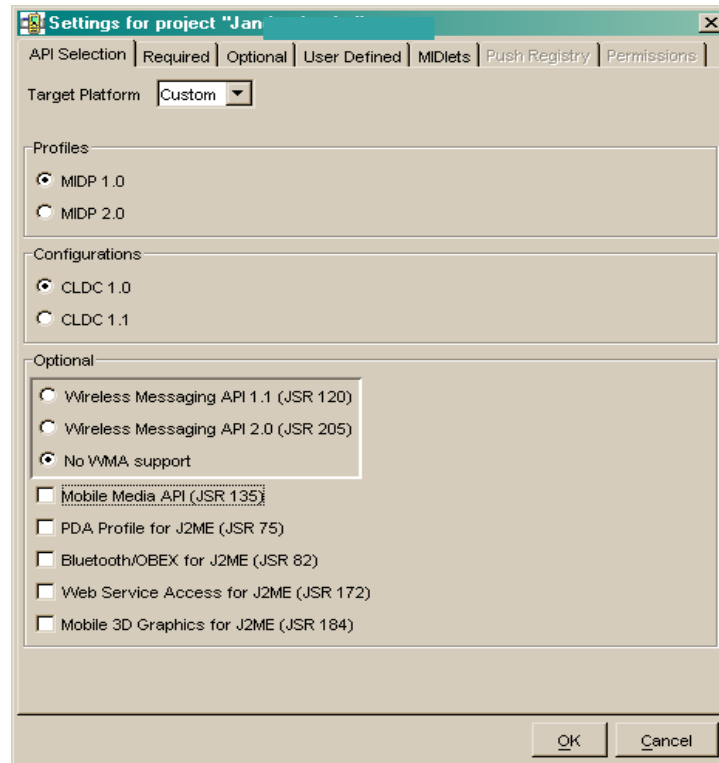


Figura 6.2: configuración de la aplicación recién creada

6.3.2 Clase Ejemplo

En el fichero Ejemplo.java está la clase que sobrecarga la clase MIDlet, donde empieza la ejecución del programa.

```

1  import javax.microedition.midlet.MIDlet;
2  import javax.microedition.midlet.MIDletStateChangeException;
3  import javax.microedition.lcdui.Display;
4
5  public class Ejemplo extends MIDlet {
6
7      protected void startApp() throws MIDletStateChangeException {
8          Display.getDisplay(this).setCurrent(new MenuPrincipal(this));
9      }
10
11     protected void pauseApp() { }
12
13     protected void destroyApp(boolean arg0)
14         throws MIDletStateChangeException { }
15
16 }

```

Listado 6.1: Ejemplo.java

En el listado 6.1 se puede comprobar que, al ser un programa tan sencillo, no será necesario guardar ni destruir ningún dato en el momento de pausar o destruir la aplicación. La única tarea que realiza esta clase es la de asignar los recursos de la pantalla a la clase *MenuPrincipal*. Se puede ver en la línea 7 cómo obtiene la pantalla

mediante *Display.getDisplay(this)*, y cómo asigna a ésta un objeto del tipo *Displayable*, mediante el método *setCurrent*.

6.3.3 Clase *MenuPrincipal*

Esta clase muestra el menú principal del programa, donde el usuario podrá elegir entre salir de la aplicación, configurar la visualización, o visualizar la imagen. Extiende la clase *List*.

En las líneas 10 a 14 del listado 6.2 puede verse cómo se construye como una lista del tipo IMPLICIT, y agrega 3 opciones: “Mostrar”, “Configurar”, y “Salir”. Además, se designa a sí misma como *commandListener* (línea 14). Para ello es requisito que implemente la interfaz *CommandListener* de J2ME (línea 4), y la función *commandAction* de dicha interfaz (línea 19). En este último método, puede verse cómo se comprueba la opción escogida y, a partir de ésta, sale del programa, muestra el objeto *Lienzo* (subclase de *Canvas*) con los dibujos, o muestra el objeto *Configuracion* (subclase de *Form*) donde se configuran las opciones de dibujado.

```

1  import javax.microedition.lcdui.*;
2  import javax.microedition.midlet.MIDlet;
3
4  public class MenuPrincipal extends List implements CommandListener
5  {
6      private MIDlet midlet;
7      private Configuracion config;
8
9      public MenuPrincipal(MIDlet midlet) {
10         super("Menú Principal",List.IMPLICIT);
11         append("Mostrar",null);
12         append("Configurar",null);
13         append("Salir",null);
14         setCommandListener(this);
15         this.midlet=midlet;
16         config=new Configuracion(midlet,this);
17     }
18
19     public void commandAction(Command command, Displayable displayable) {
20         String opcion=getString(getSelectedIndex());
21         if(opcion.equals("Salir")) {
22             midlet.notifyDestroyed();
23         } else if(opcion.equals("Configurar")) {
24             Display.getDisplay(midlet).setCurrent(config);
25         } else if(opcion.equals("Mostrar")) {
26             Display.getDisplay(midlet).setCurrent(
27                 new Lienzo(midlet,this,config.getColorFondo(),
28                     config.getColorTexto(),config.getTexto()));
29         };
30     }
31 }
32 
```

Listado 6.2: *MenuPrincipal.java*

6.3.4 Clase Configuración

Esta clase se encarga de guardar los datos de configuración, así como de mostrar en pantalla el formulario para llevar a cabo tal configuración.

Para ello, se añade al formulario un elemento del tipo *TextField* con el valor inicial de “Hola Mundo”, y dos elementos *ChoiceGroup* de selección exclusiva y las opciones “blanco”, “negro”, “azul”, “rojo” y “verde”.

Esta clase también se define a sí misma como *CommandListener*. El único comando que debe controlar esta vez es el comando “volver” (línea 19).

Como en la clase *Lienzo* será necesario obtener los datos de la configuración deseada por el usuario, se han definido funciones que obtengan los datos de los *Item* del formulario (líneas 24 a 33).


```

1  import javax.microedition.lcdui.*;
2  import javax.microedition.lcdui.TextField;
3  import javax.microedition.midlet.MIDlet;
4  public class Configuracion extends Form implements CommandListener {
5      private TextField texto=
6          new TextField("Texto","Hola mundo",15,TextField.ANY);
7      private String opciones[]={ "blanco", "negro", "azul", "rojo", "verde" };
8      private ChoiceGroup colorFG=
9          new ChoiceGroup("Color
texto",ChoiceGroup.EXCLUSIVE,opciones,null);
10     private ChoiceGroup colorBG=
11         new ChoiceGroup("Color
fondo",ChoiceGroup.EXCLUSIVE,opciones,null);
12     private MIDlet midlet;
13     private MenuPrincipal menuprincipal;
14     public Configuracion(MIDlet midlet,MenuPrincipal menuprincipal) {
15         super("Formulario de configuración");
16         append(colorFG);
17         append(colorBG);
18         append(texto);
19         addCommand(new Command("Volver",Command.BACK,1));
20         setCommandListener(this);
21         this.midlet=midlet;
22         this.menuprincipal=menuprincipal;
23     }
24     public String getTexto() {
25         return texto.getString();
26     }
27     public int getColorTexto() {
28         return getColor(colorFG);
29     }
30     public int getColorFondo() {
31         return getColor(colorBG);
32     }
33     private int getColor(ChoiceGroup choice) {
34         String color=choice.getString(choice.getSelectedIndex());
35         if(color.equals("blanco")) {
36             return (255*256+255)*256+255;
37         } else if(color.equals("negro")) {
38             return 0;
39         } else if(color.equals("azul")) {
40             return 255;
41         } else if(color.equals("rojo")) {
42             return 255*256*256;
43         } else { //verde
44             return 255*256;
45         }
46     }
47     public void commandAction(Command command, Displayable displayable) {
48         if(command.getCommandType()==Command.BACK) {
49             Display.getDisplay(midlet).setCurrent(menuprincipal);
50         }
51     }
52 }

```

Listado 6.3: clase Configuracion

6.3.5 Clase Lienzo

Es la clase derivada de *Canvas*, la cual se encarga únicamente de dibujar la pantalla

a partir de los datos de configuración que se le pasan por parámetros en el constructor (línea 12). Observar que en el constructor se carga una imagen de mapa de bits (líneas 20 a 23), por lo que habrá que crear una y guardarla como “prueba.png” en el directorio *res* del proyecto.

```

1
2  import javax.microedition.lcdui.*;
3  import javax.microedition.midlet.MIDlet;
4
5  public class Lienzo extends Canvas implements CommandListener {
6      private int colorFondo,colorTexto;
7      private String texto;
8      private Image imagen;
9      private MIDlet midlet;
10     private MenuPrincipal menuPrincipal;
11
12     public Lienzo(MIDlet midlet,
13                 MenuPrincipal menuPrincipal,
14                 int colorFondo,
15                 int colorTexto,
16                 String texto) {
17         this.colorFondo=colorFondo;
18         this.colorTexto=colorTexto;
19         this.texto=texto;
20         try {
21             imagen=Image.createImage("/prueba.png");
22         } catch(Exception e) {
23             System.out.println("Error cargando la imagen:
24             "+e.getMessage());
25         }
26         this.midlet=midlet;
27         this.menuPrincipal=menuPrincipal;
28         addCommand(new Command("Volver",Command.BACK,1));
29         setCommandListener(this);
30
31     protected void paint(Graphics graphics) {
32         int w=getWidth(), h=getHeight();
33         graphics.setColor(colorFondo);
34         graphics.fillRect(0,0,w,h);
35         graphics.setColor(colorTexto);
36         graphics.drawString(texto,w/2,10,Graphics.TOP|Graphics.HCENTER);
37         graphics.drawImage(imagen,w/2,h/2,Graphics.VCENTER|
38         Graphics.HCENTER);
39     }
40
41     public void commandAction(Command command, Displayable displayable) {
42         if(command.getCommandType()==Command.BACK) {
43             Display.getDisplay(midlet).setCurrent(menuPrincipal);
44         }
45     }
46

```

Listado 6.4: Lienzo.java

Para poder salir de la imagen y volver al menú inicial, es necesario crear un comando del tipo BACK en el constructor (línea 27). Al igual que las otras clases, *Lienzo*

se establece a sí misma como *CommandListener*, y trata el evento del comando en su función *commandAction* (líneas 40-44).

La función principal e imprescindible de *Lienzo*, como toda clase *Canvas* es la función *paint()* (líneas 31-38), donde se puede observar cómo a partir de un objeto *Graphics*, de manera sencilla rellena el fondo de un color, dibuja un texto de otro color, y dibuja el mapa de bits cargado en el constructor en el centro de la pantalla, utilizando los métodos *getWidth()* y *getHeight()* de éste, ya que a priori no se sabe la resolución de la pantalla, ya que puede variar según el dispositivo.

6.3.6 Compilación, prueba y empaquetado

Ahora sólo es necesario compilar la aplicación mediante el botón *Build* de KToolbar, corrigiendo los errores que el compilador pueda darlo. Una vez compilado, se probará en el emulador (figura 6.3) mediante el botón *Run*.



Figura 6.3: capturas de pantalla de la mini-aplicación creada como ejemplo

El último paso es, una vez comprobado que todo funciona bien, es crear los archivos Ejemplo.jar y Ejemplo.jad, para poder distribuir la aplicación en cualquier dispositivo J2ME. Esto se hace a través de la opción del menú de KToolbar, *Project --> Package --> Create Package*, con lo que se crearán los archivos *jad* y *jar* en el subdirectorio *bin* del proyecto.

Capítulo 7. Criterios de diseño de un videojuego sobre una plataforma móvil

En este capítulo se enumerarán los elementos más importantes a tener en cuenta a la hora de diseñar un buen videojuego sobre una plataforma móvil.

7.1 Crear un buen videojuego

En este proyecto, se considerará un **buen videojuego** todo aquél que cumpla dos objetivos:

1. Espectacularidad: el videojuego ha de tener un buen diseño gráfico y una música adecuada al tipo de juego, que sepa acompañar correctamente la acción de este. No basta sólo con esto, sino que además el videojuego deberá tener cierta calidad técnica y aprovechar las características que el hardware del dispositivo pone a disposición del programador; pero con cierta moderación, ya que se corre el peligro de sacrificar dosis de jugabilidad por querer centrarse demasiado en los alardes técnicos.
2. Jugabilidad: sin duda alguna, el objetivo de mayor prioridad. Nunca se debe disminuir la jugabilidad en pro de un juego más espectacular técnicamente. No hay una fórmula para obtener una buena jugabilidad, pero sí se considerarán algunos factores necesarios, aunque no imprescindibles: facilidad de manejo, dificultad ajustada y creciente, no repetitividad, etc...

Estos dos objetivos no siempre son fáciles de conseguir, y menos en dispositivos móviles, debido a dos factores: **características técnicas** y **ergonomía**. A continuación se explicarán ambos y se enumerarán posibles criterios a seguir para diseñar un buen videojuego.

7.2 Factores técnicos

A la hora de diseñar un videojuego para un dispositivo móvil, no se debe olvidar que éstos tienen unas características técnicas limitadas y, por tanto, no es posible realizar videojuegos de gran espectacularidad técnica. Y no sólo eso, sino que las aplicaciones tampoco deben tener una gran carga de proceso, es decir, gran número de elementos a controlar a la vez (por ejemplo, muchos enemigos en pantalla), una inteligencia artificial

compleja y costosa computacionalmente, o cálculos matemáticos intensos, ya que todas estas características consumen demasiados recursos de procesador y puede hacer que el juego sea demasiado lento.

Afortunadamente, estos factores tienen cada vez menos importancia, ya que la tecnología móvil avanza a pasos de gigante, y hoy en día empezamos a disfrutar de dispositivos móviles con varios megas de RAM, aceleración 3D, microprocesadores potentes, y otras características que hacen que se puedan realizar videojuegos que pocos años atrás estaban reservados a videoconsolas y ordenadores personales (figura 7.1).



Figura 7.1: Colin McRae
2005 para N-Gage

7.3 Factores de ergonomía

Con los dispositivos móviles actuales, este es el factor a tener más en cuenta a la hora de diseñar un videojuego.

7.3.1 Controles

A medida que la tecnología avanza, no sólo se consiguen dispositivos más potentes, sino que cada vez son más pequeños. Aunque gracias a las nuevas características multimedia, las pantallas van adquiriendo tamaños y definiciones aceptables, esto se compensa con teclados pequeños, lo cual puede suponer un serio problema a la hora de controlar la acción del juego: si ya de por sí es poco cómodo utilizar un teclado numérico para jugar, si éste además es de reducido tamaño puede dar como resultado una dificultad excesiva para manejar los controles y, como consecuencia, que el juego, por muy jugable que sea *a priori*, adquiera una dificultad excesiva que haga perder toda la jugabilidad que debiera tener.

En la mayoría de teléfonos, los controles se tendrán que realizar mediante el teclado numérico. Esto quiere decir que los botones para desplazar el personaje, así como los botones de acción, están en un pequeño espacio de 3x4 teclas. Lo ideal es que todo el control se pueda realizar mediante una sola mano, ya que es bastante incómodo tener dos manos en un teclado numérico de tamaño pequeño. La recomendación para este caso es que se utilicen las teclas 2, 4, 6, y 8 para controlar la dirección (más 1, 3, 7, 9 en caso que se quieran usar diagonales) y, para simplificar los controles, un sólo botón de acción, que será el centro (5). En el juego, es recomendable que no sean frecuentes los momentos en que haya que moverse y pulsar el botón de acción a la vez (ejemplo: un

mata-marcianos), ya que la acción implica que se deban soltar los botones de dirección.

En la actualidad, cada vez son más los teléfonos que vienen con un sencillo *control-pad* (figura 7.2). Esto proporciona mucha comodidad a la hora de jugar, por lo cual facilita la tarea del diseño del juego. Ahora se puede asir el teléfono con las dos manos, sin que un dedo estorbe a otro, por lo que, mientras con una mano se mueve al personaje, con la otra se puede realizar la acción. También se podrán implementar varias acciones, cada una con su botón. Para no complicar el manejo, se recomienda que como máximo sean 2 acciones de uso frecuente como por ejemplo, un botón para disparar y otro para saltar.

Para los *control-pads* actuales, hay que tener en cuenta que son de sólo 4 direcciones, con lo cual, si queremos usarlos habrá que suprimir el movimiento en diagonal del personaje.



Figura 7.2: cada día es más habitual que el teléfono móvil disponga de un sencillo control-pad

7.3.2 Pantalla

Otro problema que, como ya se ha comentado, va mejorando con el tiempo, es el tamaño de la pantalla. Los teléfonos con mayores prestaciones en la actualidad suelen tener definiciones de unos 150x200 pixels, aproximadamente. Esto causa que, en muchas ocasiones, el diseñador tenga que escoger entre espectacularidad gráfica o jugabilidad. Para determinados juegos, si se crean unos *sprites* (elementos móviles) demasiado grandes, ocuparán demasiado espacio en pantalla, reduciendo así el campo de acción y la jugabilidad. Si, en cambio, se crean unos *sprites* demasiado pequeños, puede ser que el jugador tenga que hacer un esfuerzo visual demasiado grande, y que la jugabilidad se vea reducida por que los elementos no se identifican bien, o por el esfuerzo visual necesario. Además, y como problema secundario, con tamaños de *sprite* pequeños se reduce el detalle gráfico de los elementos y, por tanto, la espectacularidad del juego.

La recomendación que se hace en este proyecto, es que los elementos del juego sean suficientemente grandes como para ser identificados fácilmente, pero que el campo de visión de la escena del juego sea suficientemente grande como para poder prever con antelación las situaciones del juego (por ejemplo: que podamos divisar si se acerca un

enemigo y reaccionar a tiempo).

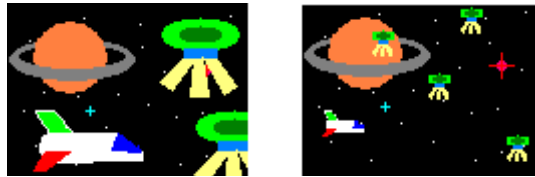


Figura 7.3: A menudo, habrá que escoger entre espectacularidad y jugabilidad, siendo en ocasiones imposible de proporcionar esta última característica.

Si el juego que se está diseñando no puede cumplir las recomendaciones aquí expuestas, quizás es que no es un tipo de juego adecuado para realizarse sobre un dispositivo móvil.

7.4 Consejos para la mejora de la jugabilidad

A continuación se describen algunos elementos que, aunque no son imprescindibles, sí pueden ayudar a la mejora de la jugabilidad:

- No repetitividad: es importante que el juego sea variado, ya que si no, la partida se hará aburrida y demasiado larga. Esto se puede conseguir con variedad de personajes, pantallas, escenarios, objetivos.
- Tener un objetivo determinado y bien definido: desde el principio, el jugador ha de saber cuál es el objetivo de la partida. Es recomendable dividir la partida en varios objetivos a conseguir, secuencialmente y uno a uno.
- Dificultad creciente y ajustada: tanto si hay demasiada dificultad como si el juego es demasiado sencillo, puede ser aburrido. Lo aconsejable es una dificultad pequeña al principio, mientras el jugador aprende a desenvolverse, y que esta vaya creciendo de manera gradual. El usuario no debe escoger el nivel de dificultad al principio (el clásico “fácil, normal, difícil” a que nos tienen acostumbrados muchos videojuegos), ya que esto es delegar en el usuario una decisión que el diseñador no ha sido capaz de tomar.
- Motivación: el jugador debe poder observar sus progresos en el juego y debe ser estimulado para seguir adelante. Esto se puede conseguir con un buen argumento, con la adquisición de nuevas habilidades (armas, poderes, etc...), u otros factores que el diseñador considere de interés.

- Estado del teléfono: éste no debe influir sobre la jugabilidad. Es decir, si el teléfono tiene el sonido apagado, esto no debe ser una desventaja de cara al juego.

Capítulo 8. El primer juego

Para iniciar la familiarización con el desarrollo de juegos para teléfonos móviles, se ha desarrollado un videojuego sencillo, capaz de ser ejecutado en cualquier teléfono móvil con soporte para J2ME y pantalla en color. Las características previas al diseño de este videojuego son las características mínimas que tiene que tener un teléfono móvil con J2ME:

- MIDP 1.0 y CLDC 1.0
- Pantalla de 96x54 pixels
- Teclado numérico solamente.

A partir del hardware especificado, se deduce que el juego deberá tener las siguientes características:

- Manejo sencillo
- Elementos de tamaño grande y perfectamente distinguibles, debido al reducido tamaño de la pantalla.
- Escenario de acción reducido, imprescindible para que la jugabilidad sea buena en un juego de estas características.

Cumpliendo todas estas especificaciones, se ha creado el clásico juego de “matar topos”, en el que unos personajes van asomando por agujeros y hay que golpearlos para ganar puntos (ver figura 8.1).

En el apéndice B se puede observar el código fuente de la aplicación.



Figura 8.1: "Escabechina", nuestro primer juego

8.1 Estructura de la aplicación

Antes de realizar la aplicación, hay que definir los estados que ésta tendrá: presentación, juego y pausa.

- Presentación: pantalla inicial con el título del juego. Aquí se podrá elegir entre jugar una partida o salir de la aplicación.
- Juego: parte principal, donde se desarrolla toda la acción. Desde aquí, se debe poder pausar el juego.
- Pausa: estado de pausa, donde se puede elegir entre continuar el juego o volver a la pantalla de presentación. A este estado se llega desde el juego, o automáticamente mediante el evento *pauseApp()* del MIDlet, por si se recibe una llamada de teléfono, para no perder la partida.
- Fin de juego: pantalla final, en la que se muestra la puntuación total y el mensaje de fin de juego.

En la figura 8.2 se muestra un diagrama con los diferentes estados del juego.

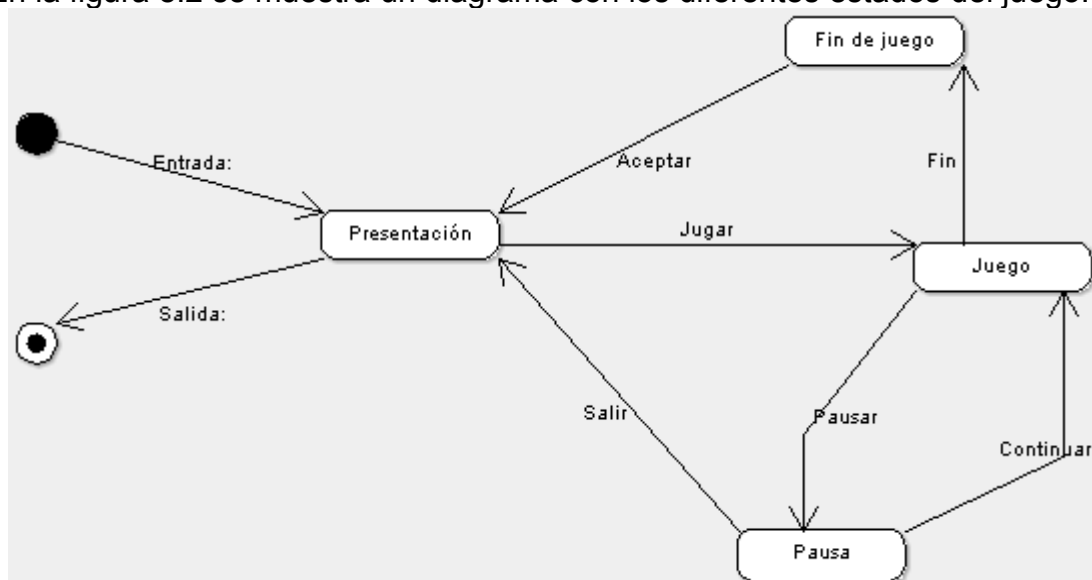


Figura 8.2: diagrama con los distintos estados del juego

Antes de programar, es necesario identificar las clases necesarias para la realización del juego (figura 8.3), que a continuación serán descritas:

- Escabechina: clase principal de la aplicación. Se encarga de controlar los diferentes

estados de la aplicación y, en consecuencia, el resto de clases creadas para la ésta.

- PausaCanvas, PresentacionCanvas: *Canvas* para la presentación y la pausa. Su única funcionalidad es mostrar las imágenes correspondientes mientras se espera la entrada de un comando por parte del usuario para pasar al estado de juego o para salir de la aplicación.
- Juego: a partir de los datos de la clase *TareaJuego*, dibuja en pantalla la representación gráfica de ésta. Es también el encargado de recibir los eventos del teclado y comunicárselos a *TareaJuego*.
- TareaJuego: a través del método *run* que sobrescribe de la superclase *TimerTask*, y que se ejecuta 20 veces por segundo, controla el estado de todos los elementos del juego.

De todas las clases implementadas la principal es *TareaJuego*, que controla todos los elementos y la acción del juego. Por tanto, es la única que merece ser desgranada y explicada paso a paso.

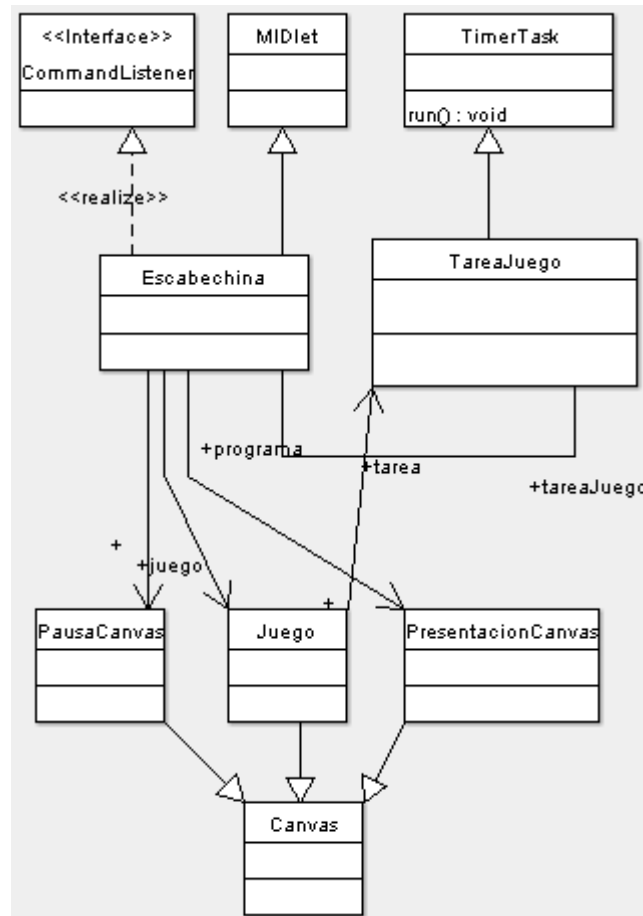


Figura 8.3: diagrama de clases de la aplicación

8.2 La clase *TareaJuego*

Ésta es una clase derivada de la clase *java.util.TimerTask*, cuyo método *run()* se ejecutará 20 veces por segundo desde un temporizador creado en la clase *Escabechina* (ver código adjunto a continuación). Esto es, cada 50 ms se llamará a dicha función, donde se controlarán todos los elementos del juego y se llamará al método *paint()* de la clase *Juego*, derivada de *javax.microedition.lcdui.Canvas*.

```

73         if(timer==null) {
74             timer=new Timer();
75             timer.schedule(tareaJuego,DELAY_MS,DELAY_MS);
76         }
    
```

¿Por qué hacer esto? ¿No sería más fácil no utilizar ningún temporizador y sustituirlo por un bucle que fuera llamando a *run()* hasta que se finalizara el juego? Sí, sería más fácil pero son varios motivos los que obligan a realizarlo con un temporizador:

- Evitar el colapso de la aplicación, e incluso del teléfono: esto se debe a que la aplicación debe otras tareas y eventos, como pueden ser los eventos del teclado, llamadas entrantes, etc. Si el programa no sale del bucle infinito, no sabemos cómo

puede reaccionar el sistema ante un simple evento sin atender.

- Sincronizar la velocidad: la velocidad del juego variará dependiendo del número de elementos a controlar, del tamaño de los gráficos a dibujar, de la velocidad del dispositivo, etc. Así, en diferentes dispositivos, e incluso en diferentes etapas del juego, se obtendrán diferentes velocidades, siendo imposible controlarlas. De esta manera queda asegurado que se obtendrán 20 imágenes por segundo, ni una más, ni una menos.

Este método difiere del de otros autores, como el de la referencia bibliográfica [García 2003], que propone crear un *thread* aparte para el juego y dentro de éste un bucle, que ejecute todas las acciones de control del juego y que, al finalizarlas, se quede en estado de espera (*sleep*) un tiempo determinado, en el que el dispositivo podrá atender otros eventos. Este método no es tan bueno como el propuesto en este proyecto, por el siguiente motivo: no se puede tener la certeza de que se obtendrá un número determinado de *frames*/segundo, ya que el tiempo de ejecución de las tareas del juego no va incluido en el tiempo entre imagen e imagen, al llamarse al método *sleep* después de la ejecución de éstas. Además, el tiempo de ejecución de dichas tareas variará según los factores que comentábamos antes. Se puede intuir que se obtendrá un número de *frames*/segundo cercano al que se busca, pero no se puede asegurar (ver figura 8.4).

El sistema utilizado para la realización del juego de este capítulo, sin embargo, también es susceptible a errores: la suma del tiempo de ejecución y el tiempo de dibujado en pantalla no puede ser superior al tiempo de entre imagen e imagen definido, ya que esto podría bloquear el sistema, al no haber tiempo de suspensión alguno y ejecutar una iteración del código cuando todavía no se hubiera acabado la anterior.

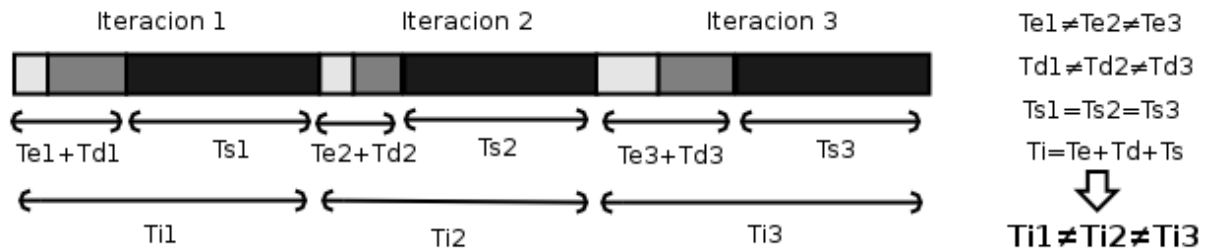
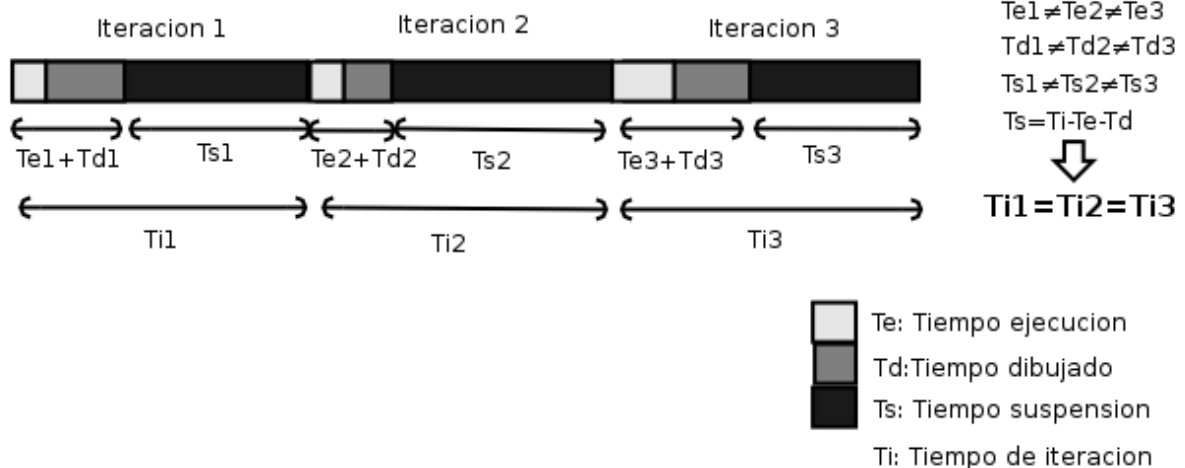
Thread aparte + bucle + sleep Thread:**Timer:**

Figura 8.4: la única manera de obtener un número exacto de frames/segundo es mediante la utilización de timers

Para un juego sencillo como este, el tiempo de dibujado y ejecución son muy menores al tiempo de iteración, por lo que no habrá problemas. En capítulos posteriores, con la creación de un juego más complejo, se explicará un método más eficiente para conseguir imprimir al juego la velocidad que se quiera, pudiendo variar el *frame rate* (número de imágenes por segundo) en función del tipo de máquina o de las variaciones de las necesidades de computación del juego en cada momento.

A continuación se explicará el contenido de la función *run()* de la clase *TareaJuego*, que es donde reside el control de todo el juego.

Primeramente, la clase *Juego* recibe los comandos del teclado. Es decir, comprueba cuándo se han pulsado las teclas del 1 al 9 y envía un mensaje a *TareaJuego*, indicándole el agujero al que irá a golpear el guante mediante la variable miembro *posicionguante*. En la función *run()* de *TareaJuego*, se comprueba si donde el guante está golpeando hay un muñeco, si éste no ha sido golpeado ya, y si se le puede golpear, es decir, está a una altura que varía entre la máxima posible y la máxima menos un margen de error. Con esto, facilitamos la tarea de golpear el muñeco, pero no la hacemos

demasiado fácil como para que el muñeco pueda ser golpeado aún cuando estuviera casi escondido. En la figura 8.5 se muestra de manera gráfica las condiciones de alturas necesarias para que se produzca el acierto a la hora de intentar golpear un muñeco.



Figura 8.5: sólo se podrá golpear el muñeco cuando éste esté a una mínima altura.

A continuación, la función *run()* comprueba si es necesario volver a sacar otro muñeco por el tablero, de manera aleatoria. Para ello, se asigna un intervalo de tiempo entre aparición y aparición, y se comprueba si ya se ha superado. Para ir aumentando la dificultad, este intervalo se irá reduciendo poco a poco. Además, se irá acortando el tiempo en que el muñeco se queda en su posición de máxima altura, y también se irá acelerando la velocidad con que éste sube y baja. A continuación se muestra el código que realiza esto, donde *intervalo* es la variable que indica el tiempo que debe transcurrir entre que asoma una cabeza y la siguiente, *nivel* es una variable que va aumentando para ser restada a *intervalo* y así ir aumentando la dificultad, *velocidad* indica la velocidad de las cabezas en subir y bajar, y *maxtmparriba* indica el tiempo en que las cabezas estarán quietas en su punto máximo.

```

129         if(intervalo<800) {
130             intervalo-=45;
131             if(intervalo<(400-nivel)) {
132                 //cada vez que sobrepase el intervalo, subimos el nivel un poco
133                 intervalo=700-nivel;
134                 if(nivel<105) {
135                     nivel+=15;
136                 }
137             }
138         }
139         else {
140             intervalo-=intervalo/7;
141         }
142         velocidad=(3-(intervalo/700));
143         if(velocidad<1) velocidad=1;
144
145         maxtmparriba-=10;
146         if(maxtmparriba<150) {
147             maxtmparriba=150;
148         }

```

Observar que la variable *maxtmparriba* oscila con el tiempo. Esto se hace para que, cuando la dificultad es muy grande, el nivel vuelva a bajar y el jugador pueda descansar. Para que, a pesar de la oscilación, la tendencia de la dificultad sea siempre creciente, se utiliza la variable *nivel*, que irá siempre acortando el tiempo de intervalo.

Al final de la función, se comprueba el estado de los 9 hoyos del tablero, que pueden estar vacíos o no. En el caso de no estarlo (hay asomado un personaje), el programa se encarga de moverlos según sea su estado (*ESTADO_SUBE*, *ESTADO_PARADO*, *ESTADO_BAJA*). Si cuando el personaje ha bajado por completo, éste no ha sido golpeado, se restará una “vida” a la partida, acabándose ésta cuando el número total de vidas sea 0. Para dar oportunidades al jugador, cada 20 aciertos seguidos se aumentará en 1 el número total de vidas.

Capítulo 9. Creación de un videojuego avanzado

Una vez cumplida la familiarización con la programación para J2ME, en el siguiente capítulo se explicará el proceso de desarrollo seguido para crear un videojuego que, si bien es posible que no esté al nivel de las producciones actuales (principalmente debido a la falta de un equipo de desarrollo numeroso), sí pueda tener una calidad aceptable desde el punto de vista de un videojuego comercial.

A diferencia de el juego creado en el capítulo anterior, para el del capítulo presente no se impondrán limitaciones hardware tan estrictas. La única limitación es que todas las librerías utilizadas sean las propuestas por Sun, dejando de un lado las API de terceros (Nokia, Samsung...). Los requisitos mínimos para la aplicación serán:

- Pantalla en color, con varios miles de colores.
- Definición de pantalla de al menos 150x180 pixels.
- Librerías para la reproducción de contenidos multimedia MMAPI.
- CLDC 1.1
- MIDP 2.0

En el momento de realización de este proyecto, los requerimientos mínimos son altos. Esto no debe ser un problema, ya que con los rápidos avances en el campo de la telefonía móvil multimedia, es cuestión de meses que cualquier teléfono de gama media incorpore todas estas especificaciones. Por otra parte, es bueno que en un proyecto académico, que no requiere de los beneficios económicos necesarios de todo buen proyecto comercial, se haga especial hincapié en tecnologías futuras que, aunque su actual difusión sea pequeña, en el futuro tengan posibilidades de ser ampliamente utilizadas. Estudiar y desarrollar con herramientas y dispositivos obsoletos sería poco más que “redescubrir la rueda”.

Como juego de demostración, que intente utilizar al máximo las características para la creación de videojuegos que incorpora MIDP 2.0, se ha creado un *shoot'em-up* (es decir, el clásico mata-marcianos).



Figura 9.1: algunas capturas de pantalla del videojuego creado en este capítulo

9.1 Características (deseables) de un videojuego

- Múltiples niveles: cada uno con dificultad creciente a la anterior. Con diversidad de decorados y personajes, para no hacer repetitivo el juego.
- Múltiples enemigos: que vayan apareciendo de manera combinada para hacer más variado el transcurso de la fase. Al final de cada nivel aparecerá un “enemigo final”, como prueba necesaria para acabar éste y pasar al siguiente.
- Variedad de armas: que el jugador pueda ir obteniendo en el transcurso del juego, de manera que sus poderes/habilidades vayan aumentando.
- Buena calidad técnica y artística: gráficos y música bonita para amenizar el transcurso de la partida. Además, que se intente aprovechar al máximo las características multimedia del dispositivo.

Como se verá, debido a que este no es un proyecto artístico sino técnico, este videojuego no cumple al 100% todos los requisitos especificados, pero la estructura del programa es tal que permite fácilmente crear nuevos elementos (armas, niveles, enemigos...) de tal manera que se pudieran cumplir las características deseables, en caso de continuar con el proyecto. Las “carencias” del juego son:

- Hay un sólo nivel, debido a que una vez creado el primero, crear más no aporta ninguna novedad desde el punto de vista de la programación. Se pueden crear fácilmente más niveles mediante sobrecargas de la clase abstracta *Fase*.
- Sólo hay un tipo de arma disponible. Al igual que con los niveles, crear más es trivial, sobrecargando la clase *Disparo*.

- En cuanto a la calidad artística, debido a que este es un proyecto de ingeniería informática y no de arte digital, es probable que los resultados no sean de gran espectacularidad. El creador de este proyecto se ha esmerado en la medida de sus posibilidades para intentar hacer un juego estéticamente bonito.

9.2 Estructura de la aplicación

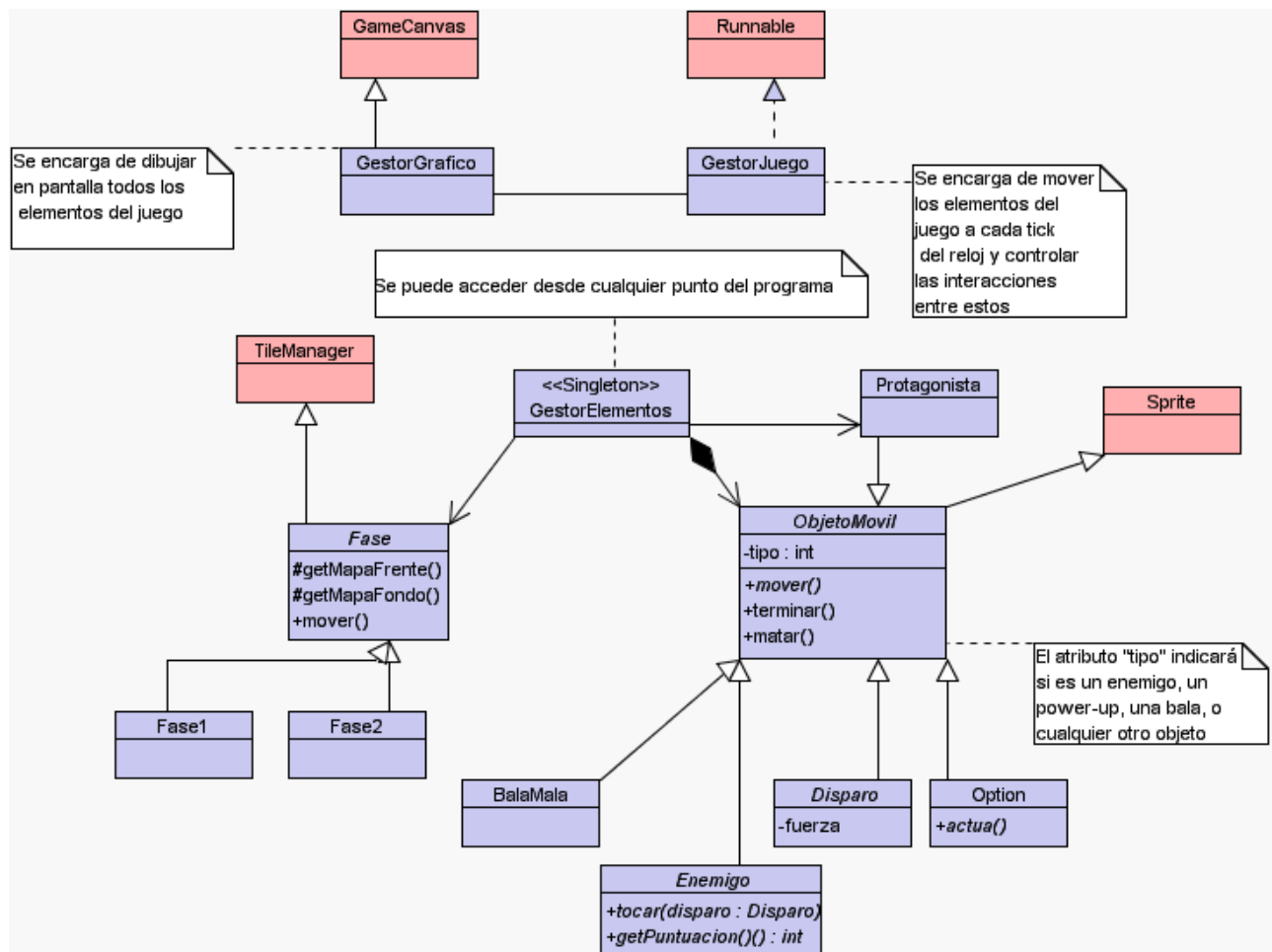


Figura 9.2: diagrama de clases del videojuego

En la figura 9.1 se puede observar el diagrama de clases de la aplicación. Esta estructura intenta realizar de manera modular, con el fin de proporcionar una aplicación fácilmente escalable, las siguientes tareas:

- Simplificar y automatizar la utilización de los recursos multimedia (gráficos y sonido).
- Bucle principal del juego, donde se trate la representación en pantalla de todos los elementos del juego, así como las interacciones entre éstos.

- Creación de clases base para simplificar la creación de elementos, tales como escenarios, objetos móviles, personajes, etc...

Para más información se puede consultar el apéndice A, donde se encuentra documentada la utilización de las clases básicas, y el apéndice C, donde se muestra el código fuente de la aplicación.

9.3 El contenedor de todo: *GestorElementos*

Esta clase es la encargada de gestionar cualquier elemento móvil del videojuego, como pueden ser enemigos, disparos, nave protagonista, etc.

Es una clase del tipo *singleton*, es decir, sólo puede haber una instancia de ésta ejecutándose a la vez, ya que la única manera acceder a ésta es mediante el método *singleton()*, que se encarga de crear una instancia del tipo *static*, y se encarga de retornar siempre la misma. Esto, además, permitirá acceder a esta clase desde cualquier punto de la aplicación.

```
private static GestorElementos ge=null;

public static GestorElementos singleton() {
    if (ge==null) {
        ge=new GestorElementos();
    }
    return ge;
}
```

Como se puede observar en el diagrama de la figura 9.1, esta clase tiene tres atributos principales:

- *fase*, atributo de clase *Fase*, donde se guarda los datos referentes a la gestión del nivel de juego que se está ejecutando.
- *prota*, atributo de clase *Protagonista*, el cual contiene la nave manejada por el jugador.
- Una lista con todos los objetos móviles (clase *ObjetoMovil*) del juego.

La única característica digna de interés en esta clase radica en la gestión de dichos objetos en memoria, ya que deben ser almacenados en una lista, la cual deberá ser recorrida secuencialmente, y a la que se le deben poder agregar y eliminar nuevos objetos. Esto en principio debería ser algo trivial, ya que Java pone a disposición del programador infinidad de estructuras de datos complejas que se manejan sencillamente mediante iteradores.

El problema vino dado por que J2ME sólo soporta la estructura *Vector*. Esto da dos problemas:

- Es una estructura que no se adapta a los requerimientos del programa (recorrido secuencial, agregar y eliminar elementos de manera rápida y sencilla, produciéndose el agregado por uno de los extremos y la eliminación en cualquier punto), con lo que puede resultar demasiado lenta. Recordar que todas estas acciones se harán varias veces por segundo.
- Otro problema es que para acceder secuencialmente, no se puede hacer directamente, sino convirtiendo el vector a un objeto de la clase *Enumeration*. Esto no sólo ralentizará más la ejecución del programa, sino que llenará demasiado rápido el *heap* del dispositivo (se crean varios *Enumeration* por segundo), por lo que la frecuencias de llamadas al recolector de basura será alta, con la consecuente ralentización del programa. En la figura 9.3 se muestra una captura del monitor de memoria donde se puede ver que la cantidad de objetos *Enumeration* ocupa una cantidad de memoria muy superior a la de cualquier otro. Además, el número de dichos objetos crecía continuamente a ritmo vertiginoso.

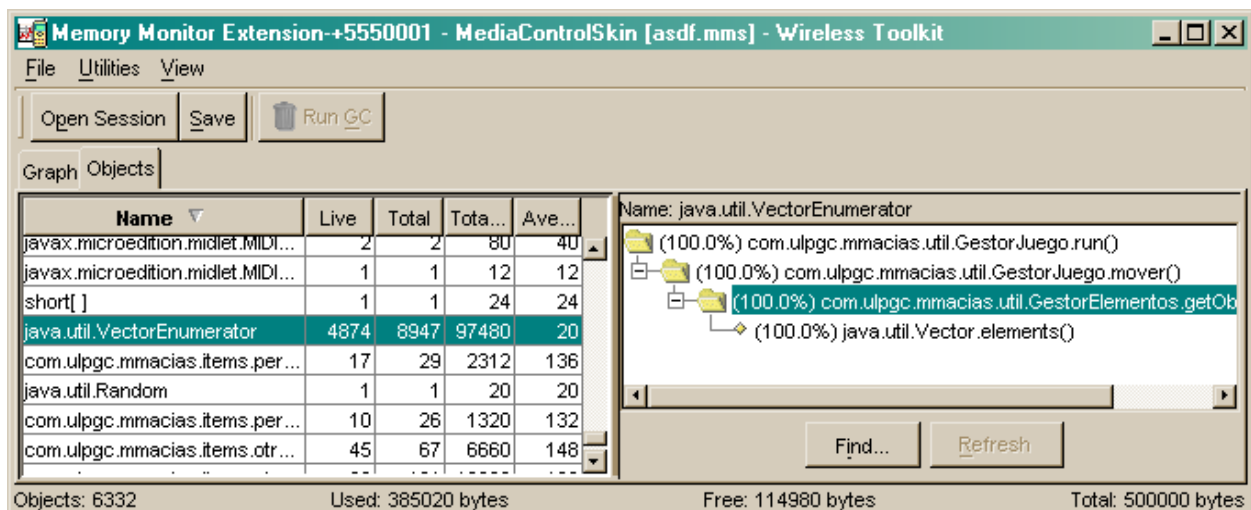


Figura 9.3: monitorización de la memoria del sistema que ocupa cada clase

La solución para conseguir una mejora del rendimiento como la que se refleja en la figura fue crear una estructura de datos sencilla; en concreto una lista enlazada, donde se agreguen los elementos al principio y se puedan eliminar en cualquier posición. Para ello se creó la estructura *Nodo*, que tiene la siguiente estructura:

```
private class Nodo {
    public ObjetoMovil objeto=null;
    public Nodo siguiente=null;
}
```

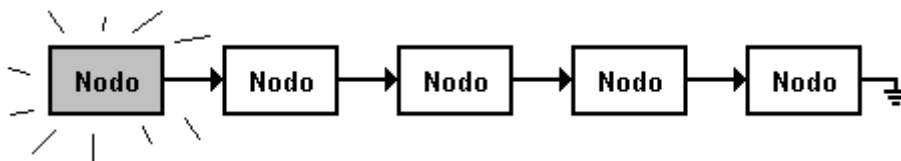
Donde cada nodo de la lista tendrá una referencia al objeto a tratar y otra al siguiente nodo de la lista.

A la hora de añadir o eliminar un elemento de la lista (ver figura 9.4), habrá que indicar a *GestorElementos* la referencia a la instancia de *ObjetoMovil* con la que operar. Como la inserción se realiza al principio de la lista, esta operación será de complejidad **O(1)**. La eliminación de un objeto es más lenta, ya que hay que buscarlo en la lista antes de eliminarlo. Como consecuencia, la eliminación media será de complejidad **O(n/2)**, donde n es la longitud de la lista. Se podría reducir a **O(1)** mediante algunos trucos, como guardar en el propio *ObjetoMovil* una referencia al nodo que le apunta, pero esto sería una mala solución desde el punto de vista de la ingeniería del software, ya que se eliminaría la independencia entre el objeto y su contenedor. Además, en el caso que nos ocupa, la lista pocas veces contiene más de 10 elementos, por lo que su total recorrido es rápido.

Lista de elementos



Inserción de un nodo



Eliminación de un nodo

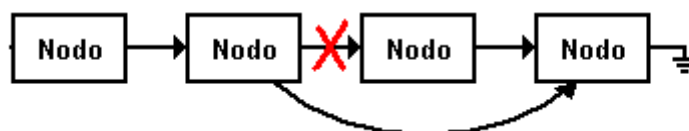


Figura 9.4: operaciones básicas con la lista de elementos

En la figura 9.5 se puede observar el uso de la memoria con la utilización de objetos *Vector* y su mejora en cuanto a utilización de memoria cuando se usa la lista simple. La línea horizontal discontinua representa el tamaño del *heap* asignado en ese momento.

Cada vez que la línea indicadora “cae” bruscamente indica que en ese momento se ha producido una recolección de basura.

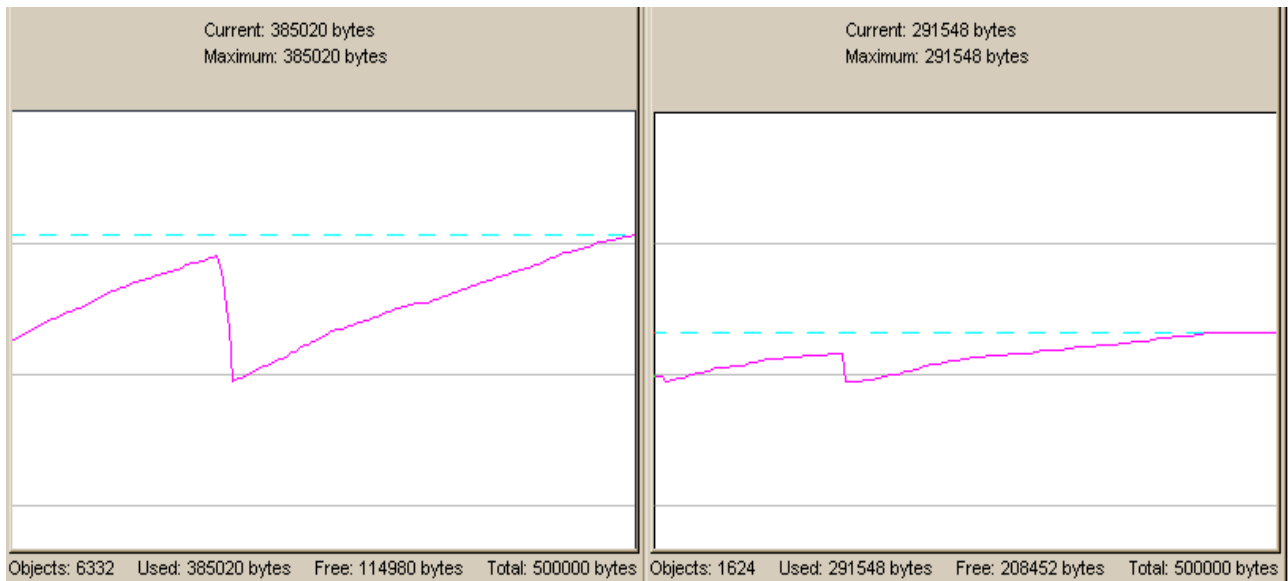


Figura 9.5: a la izquierda, uso de memoria utilizando Vector. A la derecha, utilizando una lista sencilla

9.4 GestorGrafico, lienzo de pintura y manejador de eventos

Esta clase se encarga de dibujar los elementos de la pantalla de juego. Sobrecarga la clase *GameCanvas*, en cuyo método *paint()* se dibuja el escenario del juego (clase *Fase*) y el marcador inferior cada vez que cambia la puntuación o el número de vidas.

También se encarga de capturar los eventos *keyPressed* (tecla pulsada) y *keyReleased* (tecla soltada) para indicar a la nave protagonista cuándo debe moverse y cuándo debe parar.

9.5 El núcleo de todo: GestorJuego

Esta es la clase principal y más importante del proyecto. Se encarga de controlar el estado de la aplicación, de mover los elementos y gestionar las interacciones entre éstos. Además debe controlar que todo estas tareas se hagan a la velocidad adecuada, para que el videojuego corra a la misma velocidad independientemente de la plataforma donde se esté ejecutando.

En el capítulo anterior se explicaron unas técnicas básicas para igualar la velocidad del juego en cualquier tipo de dispositivo. Se comentó que eran técnicas demasiado simples y susceptibles a errores cuando el programa fuera muy complejo. A continuación, se abre un paréntesis para explicar dos técnicas más avanzadas.

9.5.1 Sincronización avanzada

Se explicarán dos métodos para conseguir que el usuario tenga la sensación que el juego transcurre a la misma velocidad independientemente de la plataforma: fórmulas matemáticas y salto de *frames* (*frame-skipping*).

La primera consiste en determinar la posición de los objetos mediante fórmulas matemáticas calculadas en función del tiempo, mediante un *Timer* que vaya contando el tiempo o mediante el reloj del sistema (*System.currentTimeMillis()*). Con este método, el número de veces que se calculen los elementos del juego y se dibujen en pantalla variará según la potencia o la sobrecarga del sistema, pero el usuario tendrá la sensación que se mueve siempre a la misma velocidad. Sólo variará la suavidad con la que se muevan (ver figura 9.6). Además, esta técnica proporciona gran realismo al juego, ya que los elementos se podrán comportar mediante fórmulas físicas.

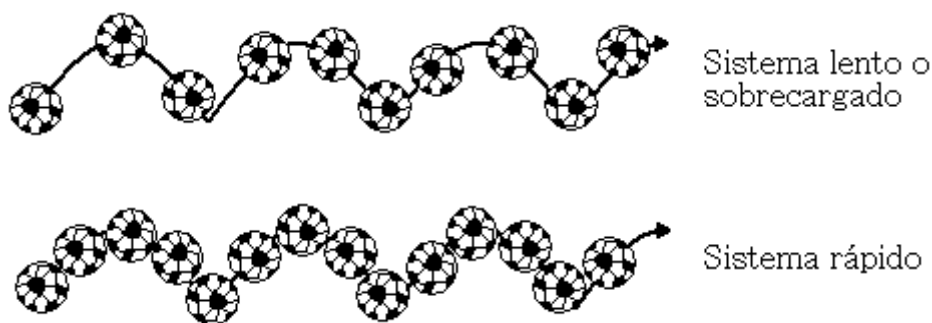


Figura 9.6: a velocidades de ejecución altas, un elemento se moverá más suavemente, pero nunca más rápido

Este tipo de implementación tiene dos problemas difícilmente resolubles en un programa J2ME:

- **Complejidad:** determinados comportamientos pueden ser difíciles de modelar mediante fórmulas matemáticas, añadiendo además un mayor coste de cálculo, ya que la posición siempre se calculará de manera absoluta (el resultado es una posición en pantalla) y no de una manera relativa (el resultado es cuantos pixels debe moverse desde donde está). Es decir, no es lo mismo tener que calcular una función del tipo $x = x' + aceleracion * tiempo$ que su equivalente en movimientos relativos $x = x + aceleracion$ (la diferencia será mucho más grande en funciones más complejas). Además, J2ME proporciona unas herramientas matemáticas extremadamente pobres (suma, resta, multiplicación, división y poco más). Incluso en la versión 1.0 del CLDC sólo se permite operar con enteros.

- Colisiones: si se va con cuidado, este es un problema menor. Este problema se da cuando los movimientos se realizan demasiado bruscamente; en ese momento, puede ser que dos objetos destinados a colisionar se “salten” el uno al otro, sin que se produzca esta colisión. Las colisiones se suelen detectar cuando determinadas regiones de dos objetos coinciden en un mismo espacio. Habría que hallar formas alternativas para descubrir las colisiones independientemente de la posición de los elementos en un determinado momento. Esto es inviable debido a las ya mencionadas limitaciones matemáticas de J2ME, así como la escasa potencia de cálculo de los aparatos móviles.

La segunda opción, el salto de *frames* (o frame-skipping), es el método escogido en este videojuego. Lo aquí explicado puede diferir de las técnicas de *frame-skipping* utilizadas por otros autores.

Lo que primeramente debe hacer el programador es escoger el número de veces por segundo que el bucle principal del juego debe ejecutarse (variable *FRAMES_S*). Si todo va bien, cada $1000/FRAMES_S$ milisegundos se debe ejecutar una iteración del bucle principal, donde se hagan todos los cálculos necesarios para mover los elementos del juego, así como dibujar éstos en pantalla.

Hasta ahora, esto no varía demasiado de la técnica del *Timer* utilizada en el tema “El primer juego” y, como entonces se explicó, si el tiempo de gestión y pintado de los elementos es superior al tiempo entre *frame* y *frame*, el juego se ralentizará.

En este momento es cuando es útil el *profiler* de la aplicación J2ME WTK (figura 9.7). Como se puede observar, casi el 90% del tiempo de ejecución se reparte en dos funciones: *BasicPlayer.realize()* y *LayerManager.paint()*. *Realize* se ejecuta sólo 3 veces al principio de la partida para pre-cachear datos de sonido, por lo que no ralentiza la posterior ejecución del juego. Entonces, se puede deducir que la función *paint* de *LayerManager* ocupa casi el 90% del tiempo de ejecución de la aplicación.

La solución propuesta para evitar las ralentizaciones cuando el tiempo de cálculo y pintado sea superior al tiempo entre *frames* es la de calcular y gestionar todos los elementos, pero no pintarlos. Para ello sólo es necesario contar las iteraciones realizadas hasta el momento y, si estas no llegan al promedio impuesto (*FRAMES_S*), saltarse el dibujo de pantalla y ejecutar inmediatamente la siguiente iteración para recuperar el tiempo perdido.

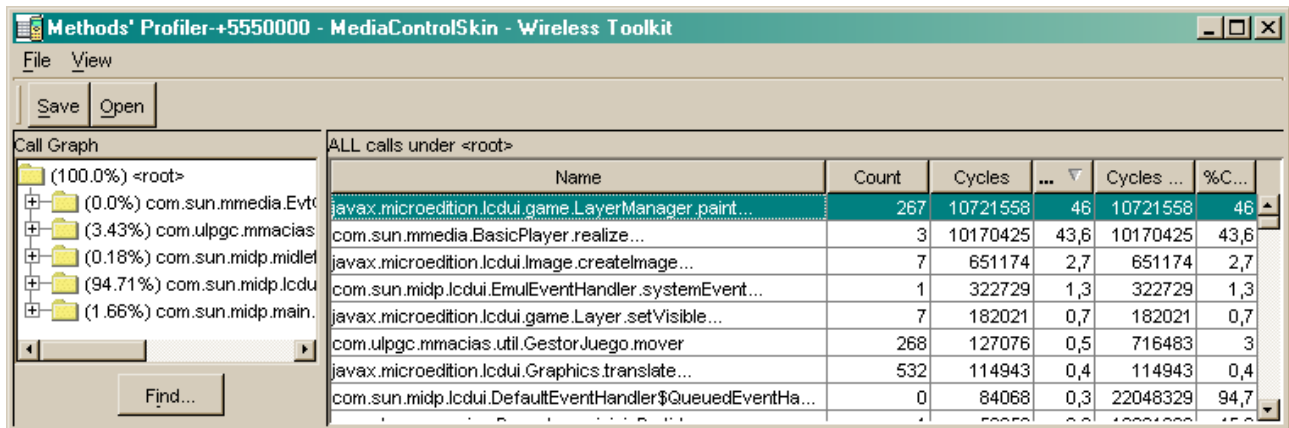


Figura 9.7: profiling de la ejecución del juego

A continuación se muestra el interior del bucle principal de la clase *GestorJuego*, donde se puede ver la implementación del *frame-skipping*:

```

1 mover();
2 iteraciones++;
3 tiempoactual=System.currentTimeMillis();
4 long idealiter=FRAMES_SEGUNDO*(tiempoactual-tiempoanterior)/1000;
5 if(iteraciones>=idealiter) {
6     gg.repaint();
7     try {
8         long tiemposiguiente=(iteraciones+1)*1000/FRAMES_SEGUNDO
+tiempoanterior;
9         tiempoactual=System.currentTimeMillis();
10        long diferencia=tiemposiguiente-tiempoactual;
11        if(diferencia>0) {
12            Thread.sleep(diferencia);
13        }
14    } catch (InterruptedException e) {
15        System.out.println(e.getMessage());
16    }
17 }

```

En la línea 4, se calcula las iteraciones que debería llevar la aplicación en caso que todo hubiera funcionado a la velocidad correcta (variable *idealiter*). Si coincide con el número actual de iteraciones, se redibuja la pantalla y se suspende la ejecución del *thread* hasta el siguiente *frame*. Si no fuera así, no se repintaría en pantalla e inmediatamente iría a ejecutar la siguiente iteración.

Esta implementación también tiene un defecto, y es que en un dispositivo muy potente no se verán las mejoras en forma de un movimiento más suave, ya que éste está limitado por el propio programa. La ventaja es que es más sencillo y rápido de utilizar que el sistema de funciones matemáticas.

9.6 Los niveles del juego: la clase *Fase*

Esta clase deriva de la clase *LayerManager* del MIDP 2.0. *LayerManager* se encarga de gestionar los *Layer* a dibujar. Un objeto de la clase *Layer* es un elemento a dibujar, como pueda ser el decorado, un enemigo, un disparo... En las páginas 100-102 se encuentra documentada la utilización detallada de esta clase.

Es una clase abstracta, por lo que no se puede instanciar. En ella se recogen todas las operaciones comunes a cualquier fase del juego. En el juego de demostración se ha creado la clase *Fase1* como derivada de *Fase*, donde se especifican los elementos concretos para el primer nivel de juego (decorados, enemigos, etc...).

Los atributos principales de *Fase* son los objetos de la clase *TiledLayer* llamados *frente* y *fondo*, que son los planos de decorado frontal y de fondo. Se han utilizado dos planos para el decorado, donde el frontal que se desplaza (*scroll*) a una velocidad mayor que el plano del fondo para dar la sensación de profundidad. Es la llamada técnica de *scroll parallax*.

La particularidad de un *TiledLayer* es que no guarda el escenario entero en un mapa de bits gigante, ya que esto ocuparía una cantidad de memoria enorme. Para ahorrar memoria, se guarda un mapa de bits con fragmentos de escenario (*tiles*) y estos se irán copiando repetidas veces en pantalla para construir el escenario completo. A modo de ejemplo, en la figura 9.8 se muestra la colección de *tiles* utilizados en la clase *Fase1* para construir el escenario del primer nivel.

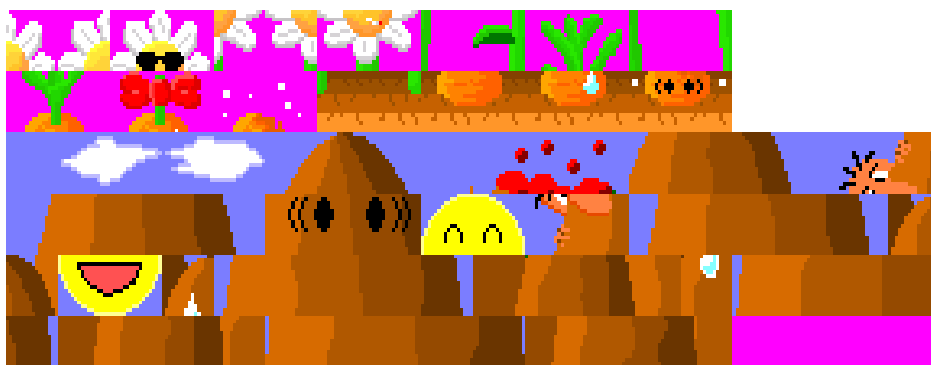


Figura 9.8: tiles utilizados en el juego de demostración

Para construir el escenario con *tiles*, se utiliza la función *setCell* de la clase *TiledLayer*, donde se indica qué *tiles* hay que dibujar, y dónde dibujarlos. Para ello, se ha creado la función *rellenarTiles*, mostrada a continuación. Esta función se llama utilizando

como parámetros los métodos abstractos *getMapaFrente* y *getMapaFondo*, los cuales deben ser sobrescritos por la clase derivada que implemente el nivel al cual pertenecen dichos mapas.

```
protected void rellenarTiles(TiledLayer tl, byte[][] tilemap) {
    for(int rows=0; rows<tilemap.length; rows++) {
        for(int cols=0; cols<tilemap[rows].length; cols++) {
            tl.setCell(cols, rows, tilemap[rows][cols]);
        }
    }
}
```

La función *mover()* será llamada desde cada iteración del bucle principal de la función *run()* de la clase *GestorJuego*, y se encargará solamente de mover los dos *TiledLayer* mientras se va avanzando en el juego. A parte de esto, en la función que implemente el nivel, la cual será instanciada, el método *mover()* se sobrecarga para añadir los distintos enemigos a la lista de la clase *GestorElementos* en el momento que el programador requiera. Para explicar esto, primeramente se muestra el trozo de código que lo implementa y a continuación se explicará su funcionamiento.

```
private int OC=0; //contador de objetos
public void mover() {
    super.mover();
    while(posx>=siguiente) {
        GestorElementos ge=GestorElementos.singleton();
        switch(objetos[OC++]) {
            case 0:
                ge.añade(new Caca(objetos[OC++]));
                break;
            case 1:
                ge.añade(new Comecocos(objetos[OC++]));
                break;
            case 2:
                ge.añade(new Cerdo());
                break;
            (....)
        }
        siguiente=objetos[OC++]*Pantalla.TILE_SIZE-Pantalla.getAnchoTablero();
    }
}

//el formato es posicion (tile horiz), num. objeto, parametro 1, ... ,
//parametro n
private static final int objetos[] = {
    13,1,70,
    16,0,16,    20,0,32,    24,0,64,    28,0,96,    32,0,128,
    40,1,25,    43,1,50,    46,1,55,    49,0,85,    52,1,115,
    55,0,135,   58,1,80,    65,2,    70,2,    75,2,
    80,1,50,    80,1,80,    80,1,120,
    90,1,25,    90,1,55,    90,1,95,
    100,0,16,   102,0,32,   104,0,48,   106,0,64,   108,0,80,   110,0,96,
```



```
(.....)
};
```

Como se puede ver, simplemente se van leyendo del vector *objetos* el momento en el que debe aparecer el elemento del tipo *ObjetoMovil*, referente al desplazamiento horizontal del escenario, el código del objeto a leer en concreto y, si este necesita parámetros, también se leerán. Como ejemplo, se puede observar los 3 primeros números de *objetos* (13,1,70), los cuales indican que el elemento con código '1' aparecerá cuando se dibuje la columna 13 del *TiledLayer*, y que además, este aparecerá a una distancia de 70 pixels del marco superior de la pantalla.

9.7 Clase *ObjetoMovil* y derivadas

ObjetoMovil deriva de la clase *Sprite* de la API de MIDP 2.0. Un *Sprite* es un *Layer*, pero con algunas características añadidas, entre las cuales se destacan las siguientes:

- Facilita la creación de animaciones, dibujando en un sólo mapa de bits todos los movimientos posibles del objeto (ver figura 9.9), pudiendo luego seleccionar cuál de ellos debe ser dibujado mediante los métodos *setFrame*, *prevFrame* y *nextFrame*.
- Se pueden efectuar transformaciones geométricas sobre la imagen del *sprite* (rotaciones, reflejos...). Esto permite el ahorro de memoria, ya que sólo será necesario crear y guardar las imágenes del elemento mirando hacia una sola dirección (ver figura 9.10).
- Ofrece mecanismos para detectar colisiones entre *sprites*. Este tipo de colisiones pueden ser simples (cuando colisionan dos áreas rectangulares definidas por el usuario) o *pixel-by-pixel* (cuando colisionan dos pixels de la imagen). En este proyecto se ha escogido el testeo de colisiones simples, ya que las necesidades de precisión no son excesivamente elevadas y su tiempo de cálculo es enormemente menor.

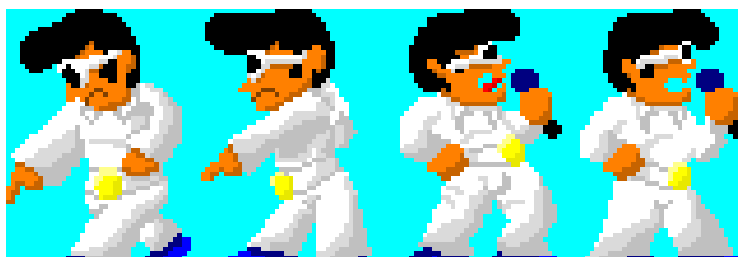


Figura 9.9: todos los movimientos de un *ObjetoMovil* se guardan en un sólo fichero de imagen

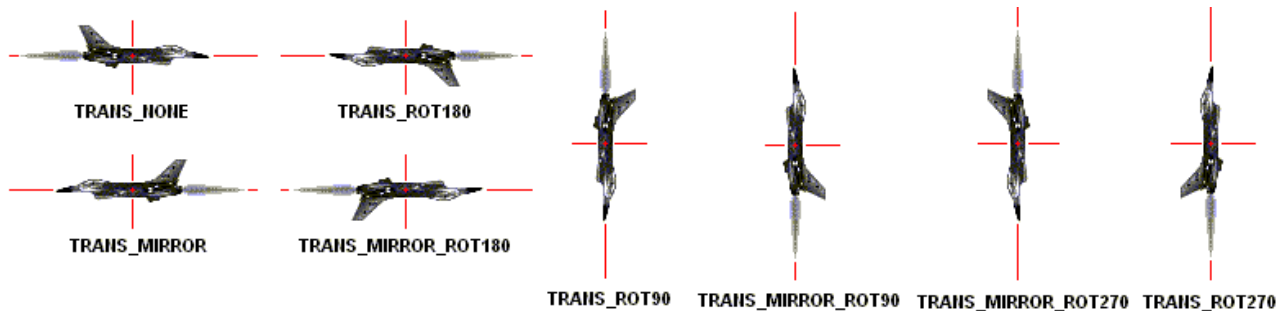


Figura 9.10: con una sola imagen se pueden obtener varias más, mediante transformaciones geométricas (fuente: Sun)

En las páginas 119-121 se encuentra la documentación en formato *JavaDoc* acerca de esta clase.

9.7.1 Características de *ObjetoMovil*

De esta clase derivan todos los elementos móviles del juego: disparos, enemigos, personaje protagonista, etc. Esta clase sólo debe implementar las operaciones comunes a todos los elementos. Estas son los métodos *mover()*, *terminar()* y *matar()*.

En el método *mover()* se debe implementar el código referente a los movimientos del elemento (por ejemplo: $x=x+1$; $y=y-1$; realizaría un movimiento en diagonal de la posición del objeto). En el método *terminar()* se implementa el código que finaliza la ejecución del *ObjetoMovil*, la función implementada por defecto tan sólo se elimina el propio objeto de la lista de objetos de *GestorElementos*. En el método *matar()* se realizan las acciones que se producen cuando el objeto muere por causa de algún disparo o otro elemento (por ejemplo, añadir a la lista de objetos una explosión y llamar a *terminar()* para eliminarse a sí mismo).

A parte de estos métodos, la clase *ObjetoMovil* contiene un atributo llamado *tipo*, en el cual se guarda el tipo de objeto al que pertenece (disparo, enemigo, objeto, protagonista, etc...). Este atributo no es necesario, ya que Java permite comprobar en tiempo de ejecución la clase a la que pertenece un objeto, pero de esta manera se acelera la ejecución, al guardar el indicador de clase en un entero.

9.7.2 Clase *Enemigo*

De esta clase, derivada de *ObjetoMovil*, derivan todos los enemigos del juego. Añade los siguientes métodos:

- *tocar(Disparo d)*, en el cual se especifican los efectos de un disparo *d* sobre el objeto *Enemigo*, tales como pérdida de energía, muerte, u otro...

- *getPuntuacion()*, que retorna la puntuación que gana el jugador cuando este enemigo muere.

9.7.3 Clase *Disparo*

De esta clase derivan todos los proyectiles lanzados por el protagonista. Esta clase agrega a la clase padre *ObjetoMovil* el método *getFuerza()*, básicamente utilizado en el método *tocar()* de la clase *Enemigo*, para obtener la potencia del disparo y así poder calcular el daño ocasionado.

9.7.4 Clase *Option*

En el juego no se ha implementado ninguna subclase de *Option*, pero se ha creado para posibles ampliaciones del juego. De esta clase derivarán los objetos que el jugador pueda recolectar durante la partida, tales como armas nuevas, puntos extra, etc...

Añade la función *actua()*, donde se especifican las acciones a realizar cuando el jugador ha recolectado ese objeto.

9.8 La gestión del sonido

A pesar que MMAPI (MultiMedia API) proporciona elementos para gestionar de manera sencilla el sonido de la aplicación, se ha decidido añadir una fina capa de software que centralice las operaciones a utilizar en el programa.

9.8.1 La clase *MediaPlayer*

Esta clase implementa la interfaz *Player*, de MMAPI, con el motivo de facilitar la carga y reproducción de los sonidos.

El único digno de ser comentado en esta clase es su constructor:

```
MediaPlayer(String filename, String contenttype, boolean repeat)
```

En él, se carga el archivo con nombre *filename* y de tipo *contenttype* (donde se especifica el tipo MIME, como “audio/midi”, “audio/wav”, etc...). Si el parámetro *repeat* es *true*, significa que al acabar de ser reproducido, el archivo deberá volver a tocarse desde el principio. Esto se consigue añadiendo un *PlayerListener*, que esté pendiente de la reproducción del archivo y, al acabar ésta (evento *END_OF_MEDIA*), vuelva a llamar al método *start()* del reproductor. A continuación se muestra el código que implementa el código que permite la repetición de los sonidos:

```
thePlayer.addPlayerListener(
```

```

new PlayerListener() {
    public void playerUpdate(Player player, String s, Object o) {
        if(s.equals(PlayerListener.END_OF_MEDIA)) {
            player.start();
        }
    }
}
);

```

9.8.2 La clase *GestorSonidos*

Esta función almacena los *MediaPlayer* a utilizar. Se encarga también de tocarlos, y eliminarlos.

La importancia de esta clase es que almacena todos los sonidos al principio de la partida y, cada vez que se quiera reproducir uno, ya tenerlo preparado de antemano para su reproducción. Esto se debe al ciclo de vida que puede tener un objeto del tipo *Player* (ver figura 9.11)

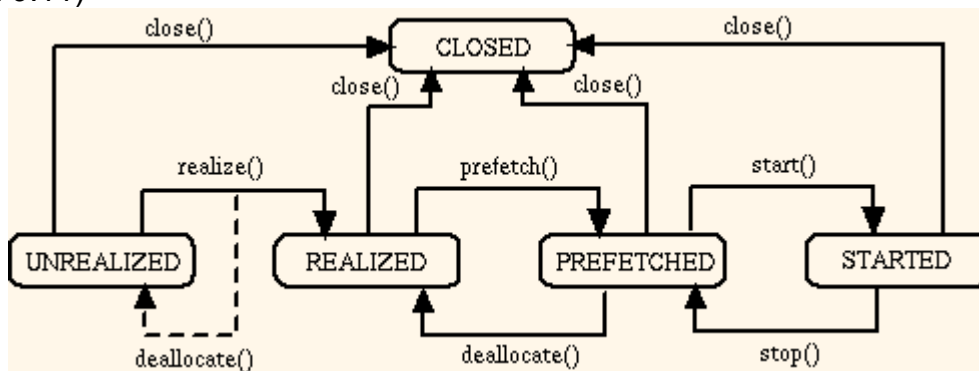


Figura 9.11: ciclo de vida de un objeto *Player* (fuente: Sun)

Un objeto del tipo *Player* empieza en el estado *UNREALIZED*, donde no ha obtenido los suficientes datos y recursos para iniciar su reproducción. El siguiente estado en el que está es *REALIZED*, cuando ya ha recolectado la suficiente información para adquirir los recursos. El paso de *UNREALIZED* a *REALIZED* es muy lento, por lo que si se hace en medio de la partida, el juego se parará algunos segundos, por eso es importante efectuarlo al principio del juego, y almacenarlo como *REALIZED* en la clase *GestorSonidos*. Hecho esto, para cambiar de sonido durante la partida, sólo hay que pasar el estado a *PREFETCHED* y luego a *STARTED*, operación que consume poco tiempo.

Si se observa el juego, al iniciar la partida el teléfono resta parado 3 o 4 segundos. Este parón es debido a que se están pasando todas las músicas al estado *REALIZED*.

Capítulo 10. Creación de una librería para la programación de videojuegos

A la hora de crear un juego, sería interesante no tener que crearlo desde 0, sino a partir de una librería que ya implementara ciertos elementos básicos, comunes a cualquier videojuego, con el propósito de automatizar lo máximo posible el proceso de creación. En este capítulo, la idea será aprovechar el esqueleto del videojuego del capítulo anterior (ver figura 10.1) y hacerle algunas modificaciones necesarias para crear una librería que sirva para crear videojuegos del tipo *shoot'em-up* (mata-marcianos) de manera rápida, e intentando proporcionar la mayor flexibilidad posible para que el programador no se vea limitado por las características de la librería.

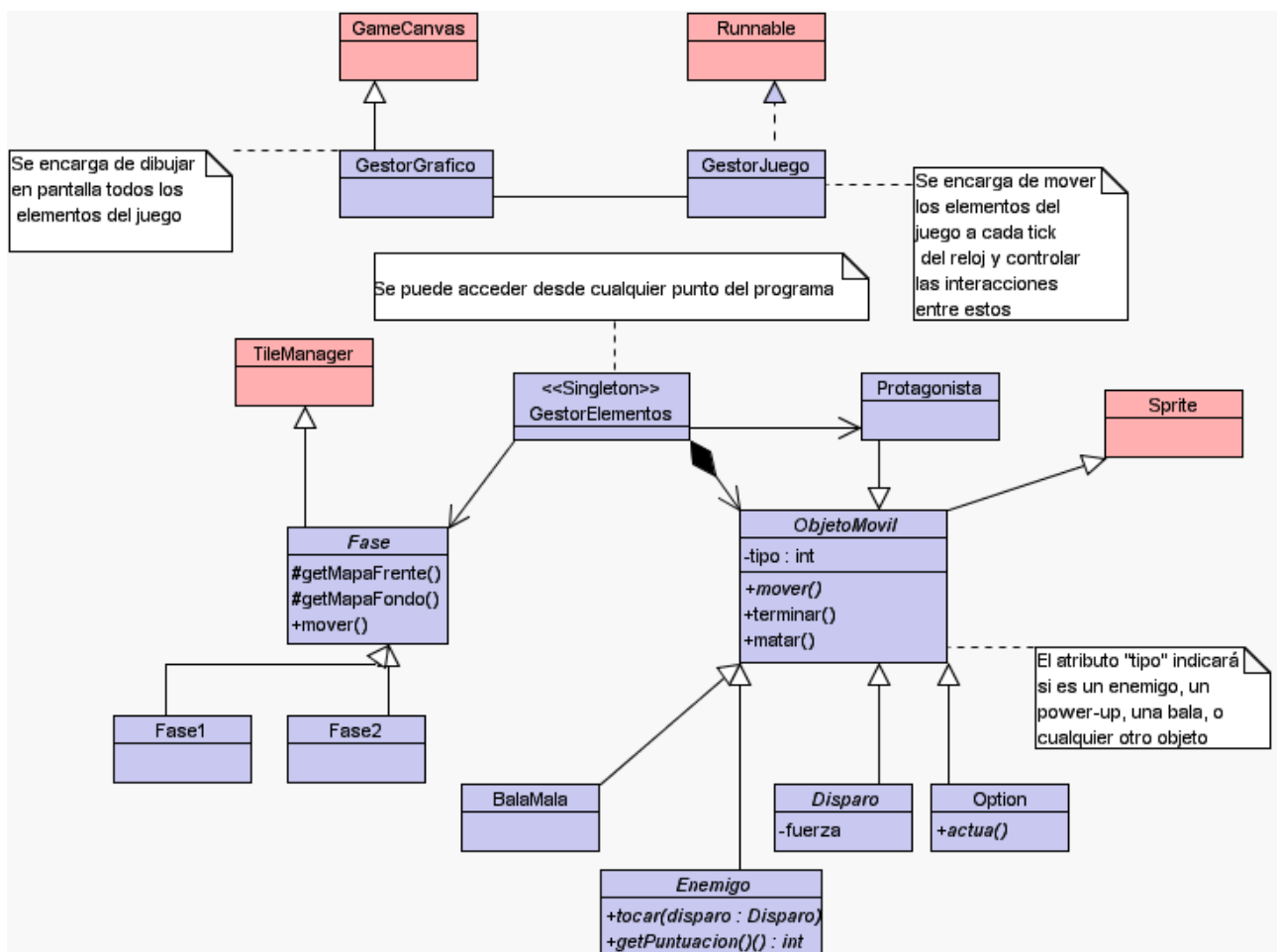


Figura 10.1: diagrama de clases de la librería

Para conocer con más detalle la utilización de las clases que componen la librería a la hora de crear videojuegos, en el Apéndice A se incluye la documentación de ésta en

formato *JavaDoc*.

10.1 Cambios realizados

En este apartado se enumerarán los cambios realizados para facilitar y flexibilizar la creación de videojuegos a partir de la librería creada.

10.1.1 Clase *GestorJuego*

Se han buscado independizar la clase *GestorJuego* del *MIDlet* que lo crea. Para ello, se han realizado los siguientes cambios:

- Se ha añadido el método *visualiza*, al que se le pasa como parámetro el objeto *Display* donde pintar la escena. Antes se le pasaba el *MIDlet* como parámetro en el constructor y *GestorJuego* se encargaba de obtener el *Display* del *MIDlet*.
- Los estados de la partida (jugando, pausa, fin de juego...) antes se gestionaban desde el *MIDlet*. Ahora los gestiona la propia clase *GestorJuego*.

10.1.2 Clase *GestorElementos*

Antes era la propia clase *GestorElementos* la que decidía el orden de las fases a ejecutar a medida que se le pedía que se fueran cargando. Ahora se le pasan en orden desde fuera, mediante un vector, en el método *setFases* y *GestorElementos* se encargará de ir cargándolas en orden, a medida que se vayan terminando las anteriores.

10.1.3 Clase *GestorSonidos*

Ahora se pueden eliminar los sonidos cargados de manera independiente, indicándole el número de sonido a eliminar. De esta manera no es necesario descargar y cargar un mismo sonido.

10.1.4 Clase *Protagonista*

Se ha hecho como una clase abstracta, donde se implementan las funcionalidades básicas de ésta, y dejando las funciones específicas para una clase derivada de ésta, creada por el programador del juego, que será la que realmente se instancie en el juego.

10.1.5 Clase *Fase*

Se han realizado cambios para dotar a ésta de mayor flexibilidad. Asimismo, se han añadido funciones para que el desarrollo de las clases derivadas sea más rápido y automatizable.

- Se ha independizado el uso de los diferentes planos del escenario (instancias de *TiledLayer*), pudiendo especificar separadamente la estructura, velocidad, visualización, etc...
- Los datos referentes a los objetos, así como sus parámetros se han integrado en la clase *Fase*. Ahora la clase derivada obtendrá los parámetros de éstos mediante la función *getParametros*, sin poder acceder directamente al *array* donde éstos están guardados.

10.2 Utilización de la librería para la creación de videojuegos

Dentro del mundo Java, la mayoría de librerías de funciones son utilizadas como librerías compiladas (archivos *.jar*), a las que se accede mediante enlaces dinámicos. En este proyecto se ha decidido proporcionar la librería como código fuente a incluir en el programa. Esto se ha hecho por los dos motivos:

- Optimización de recursos: los terminales móviles tienen recursos de cálculo y de memoria limitados. Es posible que para un desarrollo en particular no se deseen utilizar todas las características de la librería, por lo que se estarán ejecutando funciones no utilizadas, con su consiguiente repercusión en la velocidad de ejecución y la memoria ocupada. Es entonces recomendable que el programador elimine del código las partes que no desea utilizar.
- Flexibilidad: aunque la librería creada pretende dar la máxima flexibilidad posible para la creación de juegos (siempre que estos sean del estilo *shoot'em-up*), es posible que el programador desee agregar algunas características extra al juego, por lo que deberá modificar ciertas partes de la librería. Es por ello que es necesario acceder al código fuente.

Capítulo 11. Conclusiones y líneas abiertas

En el siguiente capítulo, el último, se expondrán las conclusiones extraídas de la realización del proyecto. Además, se analizarán posibles ampliaciones del proyecto y se enumerarán las líneas de desarrollo abiertas como resultado del análisis.

11.1 Conclusiones

Desde un inicial desconocimiento acerca del mundo de la creación de aplicaciones para el mundo de la telefonía móvil, pero siempre sabiendo el tipo de aplicaciones que se tratarían en el proyecto, se planteó el proyecto incidiendo en dos campos:

- Indagación en las características de las variadas herramientas disponibles para el desarrollo de aplicaciones móviles y elección de una de ellas. Una vez realizado este paso, se estudiaron con mayor profundidad las herramientas escogidas (*Java 2, Mobile Edition* y *J2ME Wireless Toolkit*) y se realizaron aplicaciones de prueba, con el objetivo de familiarizarse con el entorno de desarrollo.
- Ensayo para la creación de videojuegos bajo plataformas móviles desde un punto de vista esencialmente de la ingeniería del software. Se hacen propuestas para la correcta interacción entre el usuario y la máquina. Por otra parte, se establecen técnicas para la programación de videojuegos, tales como las estructuras de clases necesarias para la creación de aplicaciones de este tipo, así como diversos algoritmos para la gestión y la optimización en tiempo real. Todas las técnicas descritas se han acabado empaquetando en una librería para agilizar el proceso de creación.

Como el trabajo realizado gira en torno a la realización de videojuegos, se ha estudiado con especial hincapié las clases de J2ME para tal labor, tales como las que sirven para trabajar con los datos multimedia (gráficos y sonidos), así como las facilidades que MIDP 2.0 brinda para la creación de juegos, tales como *sprites*, mapas de “baldosas” (clase *TiledLayer*), “lienzzos” para dibujar (clase *Canvas*), etc.

Los elementos estudiados son básicos a la hora de crear un videojuego o algún otro tipo de aplicación multimedia, pero aún no se han explorado otras herramientas que *Sun* proporciona y que pueden enriquecer enormemente la experiencia del usuario, tales como los gráficos 3D, la reproducción de vídeos, la conexión a red, etc...

Se debe recordar ahora el capítulo 2 (Sistemas de desarrollo y ejecución de aplicaciones), donde se explican diferentes metodologías para la creación de videojuegos. Referente a esto, en un principio se inició el desarrollo de los videojuegos con el requerimiento de utilizar solamente las librerías estándar de J2ME, tanto las de la versión 1.0 del MIDP como las de la 2.0. Sin embargo, en el capítulo 9, donde se proponían diversos métodos y estructuras para la creación de videojuegos y el empaquetado de estos en una librería para su posterior reutilización, casi sin darnos cuenta se creó una interfaz entre el código que hace el programador y la Máquina Virtual de Java.

¿Qué quiere decir esto? Que si se hace un juego para esta librería, que actualmente sólo soporta MIDP 2.0, se puede exportar a otros modelos de móviles, que utilicen otras API de otros fabricantes con sólo modificar la librería, siempre que no se modifiquen las especificaciones de ésta.

Modificar la librería puede suponer un considerable esfuerzo, pero la ventaja está en que una vez modificada, los juegos que se realicen a partir de ese momento podrán compilarse para diversas plataformas, reutilizando siempre las librerías ya creadas. Es un coste inicial a tomar en cuenta, pero el ahorro y las ventajas posteriores son inmensas.

11.2 Líneas abiertas

11.2.1 Creación de un entorno de desarrollo de videojuegos

Hasta ahora se ha mostrado, como líneas abiertas, aquellas cuyo objetivo es principalmente el estudio, por parte del desarrollador, de las diversas herramientas y tecnologías puestas a su disposición para el desarrollo de aplicaciones móviles.

En el siguiente apartado, se marca una línea en la que se requiere, no tanto un estudio/evaluación de herramientas, sino un esfuerzo creador, con el objetivo de ampliar y continuar el trabajo realizado en la librería para la creación de videojuegos creada en este proyecto.

Gracias a la mencionada librería, la creación de videojuegos se convierte en una tarea sin demasiada dificultad, desde el punto de vista de la programación, la cual puede ser fácilmente automatizada. Por ejemplo, para crear una fase de un juego, tan sólo hay que definir los gráficos así como la distribución de estos mediante *tiles*, y los personajes que van a aparecer, con sus características particulares. Éstos, al tratarse de simples

arrays, pueden ser creados/modificados mediante alguna herramienta visual, que libere al desarrollador de la pesada (y susceptible a fallos) tarea de crear manualmente dichos *arrays*. Asimismo, la creación de los diferentes elementos móviles, tales como personajes, disparos, y otros elementos, también puede ser definida automáticamente, mediante un editor que permita definir la trayectoria a seguir por el objeto y algunas de las características principales de éste (fuerza, velocidad, energía...).

Por tanto, teniendo en cuenta los requerimientos descritos, se propone una herramienta integrada que, a modo de editor visual, permita la edición de los diferentes elementos de un videojuego y cree el código fuente en Java que implemente el videojuego entero, aprovechando y ampliando, si fuera necesario, la librería ya creada. A continuación se describen los elementos deseables de la aplicación resultante.

Editor de fases

Esta parte del programa debería ser capaz de editar las imágenes referentes a las imágenes de fondo, tanto los mapas de bits que forman las “baldosas” como los mapas que indican dónde debe ir colocada cada baldosa.

Actualmente hay una aplicación de código abierto, *TileStudio* (disponible en <http://tilestudio.sourceforge.net>), que implementa las funciones requeridas:

- Creación manual de mapas de bits con las baldosas a dibujar
- Creación manual de los *arrays* para formar los *TiledLayer* (ver figura 11.1).
- Creación automática de mapas de bits con las baldosas y de los *TiledLayer* a partir de un mapa de bits que dibuje la escena.
- Permite indicar en qué punto del mapa pueden encontrarse objetos, enemigos, etc...

Esta aplicación, una vez editados los datos de las pantallas, exporta a diferentes lenguajes de programación códigos fuente donde se definen los *arrays* con los datos editados.

Se podría modificar el código para agregar algunas funcionalidades específicas de la librería y crear automáticamente clases Java con los datos de las diferentes fases, en el formato que acepta la librería

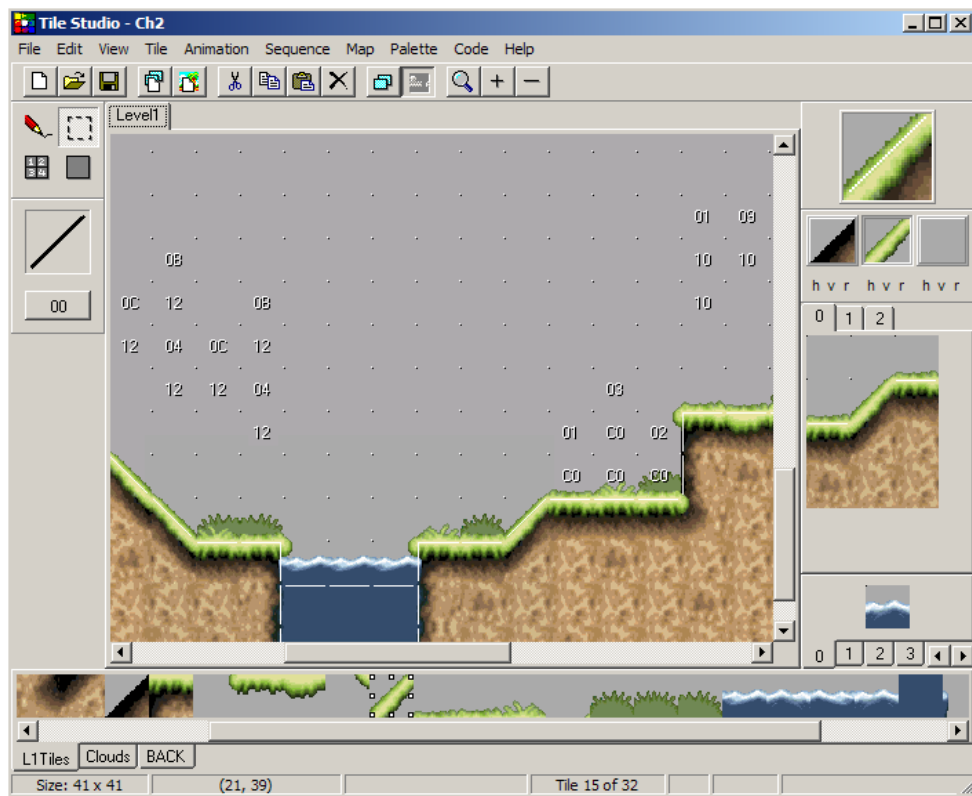


Figura 11.1: editor de pantallas de TileStudio.

Editor de objetos

Este editor debería permitir tanto enemigos, como disparos, como objetos opcionales (armas, energía extra, etc...); en definitiva, cualquier objeto que pueda derivar de la clase *ObjetoMovil* de la librería.

Las características deseables son:

- Editar los gráficos de los elementos *ObjetoMovil*, *frame a frame*, así como sus características (tamaño, número de *frames*, etc...)
- Importar los gráficos desde archivos ya creados.
- Editar las características de los elementos: energía, fuerza, velocidad, puntos que proporciona al jugador, etc...
- Editar la trayectoria de los elementos: el usuario debería definir unos puntos clave, y el código generado movería al elemento suavemente entre los diferentes puntos.
- Opcionalmente, permitir incrustar trozos de código Java para realizar funciones complejas, que no se puedan expresar con el editor. Esto daría mayor flexibilidad a

la aplicación.

Otros elementos

Sería bueno poder editar desde la aplicación algunos aspectos secundarios del juego, tales como el orden de las fases, la presentación y el final, las músicas y sonidos asociados a cada evento, etc...

11.2.2 Otras líneas abiertas

Aparte de las ampliación de la librería y creación de un entorno de trabajo, se proponen otras líneas.

A partir de la evaluación general del proyecto expuesta en el capítulo, se propondrán algunos campos en los que resultaría interesante proseguir la labor de estudio y desarrollo. Todas las *API* nombradas pertenecen a *Sun Microsystems* y pueden ser obtenidas gratuitamente desde el sitio WEB <http://java.sun.com>.

Capacidades multimedia avanzadas

MMAPI proporciona elementos para la reproducción multimedia, tales como vídeos o sonido digital. Debido al gran tamaño de los archivos de sonido (wav, mp3...) y vídeo (avi, mpg...) hoy por hoy es inviable almacenar dichos archivos en el terminal, aunque bien es cierto que en pocos años los terminales móviles incorporarán dispositivos de almacenamiento con capacidad para varios gigabytes.

Pero como es posible que todavía tarden en aparecer de manera generalizada los dispositivos móviles con capacidades de almacenamiento altas, es interesante investigar y desarrollar en aplicaciones que hagan uso del *streaming* de audio y vídeo, donde éstos se transfieran en tiempo real desde un servidor de Internet hasta el dispositivo de mano.

Gráficos 3D

Tras la reciente aparición de consolas de videojuegos portátiles con capacidades 3D muy avanzadas, es probable que en poco tiempo los móviles con soporte hardware para el 3D empiecen a ser comunes.

De hecho, *Sun Microsystems* ya tiene a disposición de los desarrolladores su *Mobile 3D Graphics API*, incluida por primera vez en el *Wireless Tool Kit 2.2*, el mismo que se utilizó para desarrollar este proyecto. Además, existe *OpenGL ES (Embedded Systems)*, una versión de las populares OpenGL para sistemas embebidos.

No hace falta ser adivino para intuir que en un futuro cercano los videojuegos 3D

para móviles sean la tendencia principal dentro del mercado, por lo que es de vital importancia estudiar este campo y desarrollar aplicaciones que lo aprovechen al máximo.

Conectividad

La telefonía móvil brinda a sus usuarios de una conectividad que, si bien es limitada en cuanto a velocidad y estabilidad, ofrece la ventaja de poder permanecer conectado prácticamente en cualquier lugar. Aprovechar estas características e implementarlas dentro de las aplicaciones de entretenimiento puede mejorar la experiencia del usuario y establecer nuevos conceptos de videojuego.

A parte de las características de conexión que proporciona MIDP, tales como conexiones HTTP o conexiones vía *socket*, Sun proporciona APIs para ampliar las capacidades de comunicación, tales como *Wireless Messaging API*, con la cual se puede manejar la mensajería SMS, o las *Java APIs for Bluetooth Wireless Technology*, con las cuales se pueden desarrollar aplicaciones para la conexión vía *bluetooth* con dispositivos cercanos.

Parte III. Apéndices

Apéndice A. JavaDoc de la librería para la creación de videojuegos

Índice por paquetes

com.ulpgc.gf.fases

Class Fase	100
------------	-----

com.ulpgc.gf.items

Class ObjetoMovil	118
-------------------	-----

com.ulpgc.gf.items.armas

Class Disparo	98
---------------	----

com.ulpgc.gf.items.options

Class Option	121
--------------	-----

com.ulpgc.gf.items.personajes

Class Enemigo	98
---------------	----

Class Protagonista	123
--------------------	-----

com.ulpgc.gf.util

Class GestorElementos	102
-----------------------	-----

Class GestorGrafico	106
---------------------	-----

Class GestorJuego	108
-------------------	-----

Class Pantalla	122
----------------	-----

com.ulpgc.gf.util.sound

Class GestorSonidos	111
---------------------	-----

Class MediaPlayer	113
-------------------	-----

com.ulpgc.gf.items.armas

Class Disparo

```
java.lang.Object
├── javax.microedition.lcdui.game.Layer
│   ├── javax.microedition.lcdui.game.Sprite
│   │   ├── com.ulpgc.gf.items.ObjetoMovil
│   │   │   └── com.ulpgc.gf.items.armas.Disparo
```

```
public abstract class Disparo
extends ObjetoMovil
```

Implementa un disparo del [Protagonista](#) o de algún enemigo.

Constructor Detail

Disparo

```
public Disparo(javax.microedition.lcdui.Image image)
```

Construye el disparo a partir de una imagen

Parameters:

image - La imagen de mapa de bits del Disparo

Method Detail

getFuerza

```
public abstract int getFuerza()
```

Retorna la fuerza del disparo. A más fuerza, menos disparos serán necesarios para acabar con un enemigo.

Returns:

La fuerza del disparo

com.ulpgc.gf.items.personajes

Class Enemigo

```
java.lang.Object
├── javax.microedition.lcdui.game.Layer
│   ├── javax.microedition.lcdui.game.Sprite
│   │   ├── com.ulpgc.gf.items.ObjetoMovil
│   │   │   └── com.ulpgc.gf.items.personajes.Enemigo
```

```
public abstract class Enemigo
extends ObjetoMovil
```

Implementa los enemigos del juego.

Constructor Detail

Enemigo

```
public Enemigo(javax.microedition.lcdui.Image image)
```

Construye el objeto `Enemigo` a partir de una imagen, sin animación.

Parameters:

`image` - La imagen de mapa de bits que representa el objeto

Enemigo

```
public Enemigo(javax.microedition.lcdui.Image image,
               int ancho,
               int alto)
```

Construye el objeto `Enemigo` a partir de una imagen, animada

Parameters:

`image` - La imagen que representa todos los *frames* del objeto animado

`ancho` - Tamaño horizontal de los frames, en píxels

`alto` - Tamaño vertical de los frames, en píxels

Method Detail

matar

```
public void matar()
```

Implementa la muerte del enemigo

Overrides:

[matar](#) in class [ObjetoMovil](#)

tocar

```
public abstract void tocar(Disparo disparo)
```

Implementa la acción a realizar cuando el enemigo es tocado por un disparo

Parameters:

`disparo` - El objeto del tipo [Disparo](#) que colisiona con el enemigo

getPuntuacion

```
public abstract int getPuntuacion()
```

Devuelve el número de puntos a sumar cuando muere el enemigo

Returns:

Número de puntos a sumar

com.ulpgc.gf.fases

Class Fase

java.lang.Object

↳ javax.microedition.lcdui.game.LayerManager

↳ com.ulpgc.gf.fases.Fase

```
public abstract class Fase
```

```
extends javax.microedition.lcdui.game.LayerManager
```

Implementa la superclase de todas las pantallas del juego.

Los gráficos de la pantalla serán guardados en una imagen, en forma de baldosas del tamaño que se especifica en el campo [Pantalla.TILE_SIZE](#) de la clase [Pantalla](#). El gráfico de la pantalla se reconstruirá a partir de el array bidimensional de `byte` que se le pasa al constructor como parámetro.

Las instancias de la clase [ObjetoMovil](#) que se vayan creando también se pasarán mediante un array de `int` donde a cada [ObjetoMovil](#) le pertenecerá una ristra de `int` según el siguiente formato:

num. objeto, parametro_1, ..., parametro_n

La correspondencia entre el número de objeto y la clase del objeto, así como el número de parámetros que cogerá del array para su construcción vendrán especificados sobrecargando la función [añadeEnemigo\(int\)](#) en la clase derivada.

Field Detail

vscrollfrente

```
protected int vscrollfrente
```

Indica la velocidad del scroll del Layer frontal

vscrollfondo

protected int **vscrollfondo**

Indica la velocidad del scroll del Layer de fondo

Constructor Detail

Fase

```
public Fase(byte[][] mapaFrente,
            java.lang.String imgMapaFrente,
            byte[][] mapaFondo,
            java.lang.String imgMapaFondo,
            int[] objetos,
            java.lang.String[] archivosMusicas)
```

Construye una nueva instancia de un objeto [Fase](#).

Parameters:

mapaFrente - Mapa de *Tiles* o "baldosas" para el plano frontal de la pantalla
imgMapaFrente - Imagen con las baldosas para el plano frontal
mapaFondo - Mapa de *Tiles* o "baldosas" para el plano fondo de la pantalla
imgMapaFondo - Imagen con las baldosas para el plano de fondo
objetos - Mapa de los objetos a crear durante el transcurso de la fase
archivosMusicas - Array de *String* con los archivos de las músicas a tocar durante el transcurso de la fase.

Method Detail

paint

```
public void paint(javax.microedition.lcdui.Graphics g)
```

Dibuja los planos frontal y de fondo en sus posiciones correctas.

Parameters:

g - Contexto gráfico

setAcabada

```
public void setAcabada(boolean acabada)
```

Marca la acción de la Fase como acabada. Generalmente se utiliza cuando se llega al final de ésta y se mata al monstruo final.

Parameters:

acabada - Si true, marca la fase como acabada.

isAcabada

```
public boolean isAcabada()
```

Comprueba si la fase está marcada como *acabada*.

Returns:

true, si la fase está acabada. false, en caso contrario.

mover

```
public void mover()
```

Se encarga de la gestión del movimiento de los planos de frontal y de fondo, así como de comprobar cuándo es necesario crear un nuevo

añadeEnemigo

```
protected abstract void añadeEnemigo(int numObjeto)
```

Relaciona el [ObjetoMovil](#) a instanciar y el número de argumentos necesario, con el número de objeto que se le pasa como parámetro.

Un ejemplo del contenido de la función sería:

```
switch (numObjeto) {  
    case 1:  
        int param[]=pideParametros(3);  
        GestorElementos.singleton().añade(new  
        FooObjeto(param[0],param[1],param[2]));  
        break;  
}
```

finalizar

```
public void finalizar()
```

Finaliza la ejecución de la pantalla. Y libera los recursos acumulados por ésta

com.ulpgc.gf.util

Class GestorElementos

java.lang.Object

 com.ulpgc.gf.util.GestorElementos

```
public class GestorElementos  
extends java.lang.Object
```

Gestiona los recursos de memoria asociados a las instancias de [ObjetoMovil](#) y de [Fase](#).

Es una clase del tipo *singleton*, lo cual quiere decir que sólo hay una instancia de ella en toda la aplicación, además, es accesible desde cualquier punto de ésta.

Constructor Detail

GestorElementos

```
public GestorElementos()
```

Method Detail

getFaseSize

```
public int getFaseSize()
```

Obtiene el tamaño horizontal en *Tiles* de la Fase activa

Returns:

El tamaño horizontal en *Tiles* de la Fase activa.

iniciaDatos

```
public void iniciaDatos()
```

Inicializa los datos referentes al gestor

acumulaPuntuacion

```
public void acumulaPuntuacion(int puntos)
```

Acumula la puntuación a sumar, como resultado de la muerte de los enemigos u otros eventos.

Parameters:

`puntos` - Los puntos a acumular

desacumulaPuntuacion

```
public int desacumulaPuntuacion()
```

Desacumula la puntuación a sumar.

Returns:

puntos Puntuación acumulada con anterioridad.

setProtagonista

```
public void setProtagonista(Protagonista prota)
```

Establece el objeto [Protagonista](#) activo durante el juego

getProtagonista

```
public Protagonista getProtagonista()
```

Retorna el objeto [Protagonista](#) activo durante el juego

Returns:

Un objeto de la clase [Protagonista](#)

singleton

```
public static GestorElementos singleton()
```

Retorna la única instancia de la clase `GestorElementos`

Returns:

Un objeto `GestorElementos`

destroy

```
public void destroy()
```

Finaliza la ejecución de todos los objetos contenidos en la clase `GestorElementos` y libera los recursos de memoria.

añade

```
public void añade(ObjetoMovil objetoMovil)
```

Añade instancias de [ObjetoMovil](#) para su posterior tratamiento.

Parameters:

`objetoMovil` - El [ObjetoMovil](#) a añadir.

reinicia

```
public boolean reinicia(int n)
```

Inicia el contador de elementos al primer elemento de la cola

Parameters:

n - Número de contador (0 o 1)

Returns:

false si no hay elementos en la cola. true en caso contrario

siguiente

```
public boolean siguiente(int n)
```

Incrementa el contador de elementos hasta el siguiente de la lista.

Parameters:

n - Número de contador de elementos (0 o 1)

Returns:

false si se ha llegado al final de la lista. true en caso contrario

getObjetoMovil

```
public ObjetoMovil getObjetoMovil(int n)
```

Obtiene la instancia de [ObjetoMovil](#) referenciada por el contador de elementos.

Parameters:

n - Número de contador de elementos (0 o 1)

Returns:

El [ObjetoMovil](#) referenciado. null en caso de que no haya ninguno referenciado.

elimina

```
public void elimina(ObjetoMovil objetoMovil)
```

Elimina de la lista el [ObjetoMovil](#) deseado.

Parameters:

objetoMovil - La referencia al [ObjetoMovil](#) que se desea eliminar.

setFases

```
public void setFases(java.util.Vector fases)
```

Establece las fases (clase [Fase](#)) con las que contará el juego

Parameters:

`fases` - Un vector con las fases en orden de ejecución.

cargaPrimeraFase

```
public void cargaPrimeraFase()
```

Establece la primera [Fase](#) como la que va a ser ejecutada.

cargaSiguienteFase

```
public boolean cargaSiguienteFase()
```

Establece la siguiente [Fase](#) de la lista como la que va a ser ejecutada.

Returns:

`false` si ya se han ejecutado todas las fases. `true` en caso contrario.

getFase

```
public Fase getFase()
```

Retorna la fase en ejecución

Returns:

Un objeto de la clase [Fase](#)

liberaFase

```
public void liberaFase()
```

Establece como que no se está ejecutando ninguna [Fase](#)

com.ulpgc.gf.util

Class GestorGrafico

java.lang.Object

□ javax.microedition.lcdui.Displayable

□ javax.microedition.lcdui.Canvas

□ javax.microedition.lcdui.game.GameCanvas

 `com.ulpgc.gf.util.GestorGrafico`

```
public class GestorGrafico
extends javax.microedition.lcdui.game.GameCanvas
```

Se encarga de la gestión de los gráficos en pantalla.

Constructor Detail

GestorGrafico

```
public GestorGrafico(GestorJuego gestorJuego)
```

Crea una instancia de `GestorGrafico`, y se le asigna la clase [GestorJuego](#) que la utilizará.

Method Detail

paint

```
public void paint(javax.microedition.lcdui.Graphics graphics)
```

Pinta todos los elementos del juego en pantalla.

Parameters:

`graphics` - Objeto `Graphics` a utilizar.

keyPressed

```
protected void keyPressed(int i)
```

Manejador de eventos que se dispara al pulsarse una tecla del dispositivo.

Parameters:

`i` - Código de tecla pulsada

keyReleased

```
protected void keyReleased(int i)
```

Manejador de eventos que se dispara al soltarse una tecla del dispositivo, previamente pulsada.

Parameters:

`i` - Código de tecla pulsada



com.ulpgc.gf.util

Class GestorJuego

java.lang.Object

com.ulpgc.gf.util.GestorJuego

All Implemented Interfaces:

javax.microedition.lcdui.CommandListener, java.lang.Runnable

public class **GestorJuego**

extends java.lang.Object

implements java.lang.Runnable, javax.microedition.lcdui.CommandListener

Implementa el bucle principal de la ejecución del juego.

Se encarga de mover todos los elementos (Fases, Objetos, Protagonista...), así como de controlar todas las interacciones entre estos. También se encarga de llamar al dibujado en pantalla de todos los elementos.

Implementa la clase `Runnable` para ser ejecutado en un `Thread` aparte.

Field Detail

ESTADO_JUEGO

public static final int **ESTADO_JUEGO**

Indica que se está jugando una partida al juego.

ESTADO_GAMEOVER

public static final int **ESTADO_GAMEOVER**

Indica que la partida acaba de terminar (aparece el letrero GAME OVER)

ESTADO_PAUSA

public static final int **ESTADO_PAUSA**

Indica que la acción del juego se ha pausado

ESTADO_INDEFINIDO

public static final int **ESTADO_INDEFINIDO**

Indica un estado no definido, auxiliar para acciones especiales que el programador desee.

ESTADO_ACABAR

```
public static final int ESTADO_ACABAR
```

El juego terminó definitivamente (se vuelve a la presentación).

Constructor Detail

GestorJuego

```
public GestorJuego()
```

Crea una instancia de GestorJuego

Method Detail

visualiza

```
public void visualiza(javax.microedition.lcdui.Display display)
```

Indica el contexto del dispositivo a usar para la visualización en pantalla de los elementos.

Parameters:

display - Dispositivo usado para la visualización.

getEstado

```
public int getEstado()
```

Retorna el estado del GestorJuego ([ESTADO_ACABAR](#), [ESTADO_GAMEOVER](#), [ESTADO_INDEFINIDO](#), [ESTADO_JUEGO](#) o [ESTADO_PAUSA](#))

Returns:

El número de estado del gestor

setEstado

```
public void setEstado(int estado_actual)
```

Define el estado del GestorJuego ([ESTADO_ACABAR](#), [ESTADO_GAMEOVER](#), [ESTADO_INDEFINIDO](#), [ESTADO_JUEGO](#) o [ESTADO_PAUSA](#))

Parameters:

`estado_actual` - El número de estado

run

```
public void run()
```

Bucle principal del juego. Se mueven los objetos, se controlan las interacciones entre estos, y se dibujan en pantalla.

Specified by:

`run` in interface `java.lang.Runnable`

iniciaPartida

```
public void iniciaPartida()
```

Inicia los datos para empezar una partida nueva

commandAction

```
public void commandAction(javax.microedition.lcdui.Command command,  
                           javax.microedition.lcdui.Displayable displayable)
```

Manejador de eventos para cuando se pulsan los comandos de "Pausar" y "Continuar partida"

Specified by:

`commandAction` in interface
`javax.microedition.lcdui.CommandListener`

pausar

```
public void pausar(boolean pausa)
```

Implementa la pausa del juego.

Parameters:

`pausa` - Si `true`, el juego será pausado. Si `false`, la ejecución del juego continuará.

getPuntos

```
public int getPuntos()
```

Obtiene la puntuación de la partida

Returns:

El número de puntos acumulados durante la partida.

setVidas

```
public void setVidas(int vidas)
```

Indica el número de "vidas" del protagonista.

Parameters:

vidas - El número de "vidas"

getVidas

```
public int getVidas()
```

Obtiene el número de "vidas" del protagonista

Returns:

El número de "vidas"



com.ulpgc.gf.util.sound

Class GestorSonidos

java.lang.Object

□ com.ulpgc.gf.util.sound.GestorSonidos

```
public class GestorSonidos  
extends java.lang.Object
```

Se encarga de la gestión de memoria relacionada con la gestión de los recursos de sonido. También se encarga de hacerlos sonar y pararlos.

Es una clase del tipo *singleton*, lo cual quiere decir que sólo hay una instancia de ella en toda la aplicación, además, es accesible desde cualquier punto de ésta.

Method Detail

singleton

```
public static GestorSonidos singleton()
```

Obtiene la única instancia de la clase `GestorSonidos` de toda la aplicación.

Returns:

La instancia de `GestorSonidos`.

eliminaSonidos

```
public void eliminaSonidos()
```

Elimina de la memoria todos los sonidos cargados.

eliminaSonido

```
public void eliminaSonido(int index)
```

Elimina un sonido de memoria.

Parameters:

`index` - El índice del sonido a eliminar.

cargaSonido

```
public int cargaSonido(java.lang.String filename,  
                        java.lang.String type,  
                        boolean repeat)
```

Carga en memoria un sonido.

Parameters:

`filename` - El nombre del fichero que contiene el sonido

`type` - El tipo MIME del sonido ("audio/midi", audio/mp3, etc...)

`repeat` - Si true, al acabar de tocarse el sonido completo, automáticamente se empezará a tocar de nuevo desde el principio.

Returns:

El índice del sonido.

tocaSonido

```
public void tocaSonido(int index)
```

Comienza a tocar un sonido.

Parameters:

`index` - El índice del sonido a tocar.

getIndexSonidoActual

```
public java.lang.Integer getIndexSonidoActual()
```

Obtiene el índice del sonido que está tocando.

Returns:

El índice del sonido que está tocando. `null` en caso de que no se esté tocando ninguno.

paraSonido

```
public void paraSonido()
```

Termina el sonido que está tocando.

getMediaPlayer

```
public MediaPlayer getMediaPlayer(int index)
```

Obtiene el objeto [MediaPlayer](#) del sonido que se desee.

Parameters:

`index` - Índice del sonido.

Returns:

La instancia de [MediaPlayer](#) del sonido deseado.



`com.ulpgc.gf.util.sound`

Class MediaPlayer

```
java.lang.Object
```

```
└─ com.ulpgc.gf.util.sound.MediaPlayer
```

All Implemented Interfaces:

`javax.microedition.media.Controllable`, `javax.microedition.media.Player`

```
public class MediaPlayer
```

```
extends java.lang.Object
```

```
implements javax.microedition.media.Player
```

Se encarga de tocar y parar los sonidos cargados en memoria. La idea de ésta clase es pre-cachear los sonidos en memoria para tenerlos disponibles al instante en que se quiera comenzar su ejecución.

Se debe tener cuidado, ya que si se cachean demasiados sonidos, la memoria se llenará

muy fácilmente.

Constructor Detail

MediaPlayer

```
public MediaPlayer(java.lang.String filename,  
                   java.lang.String contenttype,  
                   boolean repeat)
```

Construye un objeto `MediaPlayer` para un determinado sonido.

Parameters:

`filename` - El nombre del fichero que contiene el sonido

`contenttype` - El tipo MIME del sonido ("audio/midi", audio/mp3, etc...)

`repeat` - Si `true`, al acabar de tocarse el sonido completo, automáticamente se empezará a tocar de nuevo desde el principio.

Method Detail

start

```
public void start()
```

Comienza a tocar el sonido

Specified by:

`start` in interface `javax.microedition.media.Player`

getControls

```
public javax.microedition.media.Control[] getControls()
```

Ver documentación de `javax.microedition.media.Controllable`

Specified by:

`getControls` in interface `javax.microedition.media.Controllable`

getControl

```
public javax.microedition.media.Control getControl(java.lang.String s)
```

Ver documentación de `javax.microedition.media.Controllable`

Specified by:

`getControl` in interface `javax.microedition.media.Controllable`

realize

```
public void realize()  
    throws javax.microedition.media.MediaException
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`realize` in interface `javax.microedition.media.Player`

Throws:

`javax.microedition.media.MediaException`

prefetch

```
public void prefetch()  
    throws javax.microedition.media.MediaException
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`prefetch` in interface `javax.microedition.media.Player`

Throws:

`javax.microedition.media.MediaException`

stop

```
public void stop()
```

Para el sonido

Specified by:

`stop` in interface `javax.microedition.media.Player`

deallocate

```
public void deallocate()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`deallocate` in interface `javax.microedition.media.Player`

close

```
public void close()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`close` in interface `javax.microedition.media.Player`

setTimeBase

```
public void setTimeBase(javax.microedition.media.TimeBase timeBase)  
    throws javax.microedition.media.MediaException
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`setTimeBase` in interface `javax.microedition.media.Player`

Throws:

`javax.microedition.media.MediaException`

getTimeBase

```
public javax.microedition.media.TimeBase getTimeBase()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`getTimeBase` in interface `javax.microedition.media.Player`

setMediaTime

```
public long setMediaTime(long l)  
    throws javax.microedition.media.MediaException
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`setMediaTime` in interface `javax.microedition.media.Player`

Throws:

`javax.microedition.media.MediaException`

getMediaTime

```
public long getMediaTime()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`getMediaTime` in interface `javax.microedition.media.Player`

getState

```
public int getState()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`getState` in interface `javax.microedition.media.Player`

getDuration

```
public long getDuration()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`getDuration` in interface `javax.microedition.media.Player`

getContentType

```
public java.lang.String getContentType()
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`getContentType` in interface `javax.microedition.media.Player`

setLoopCount

```
public void setLoopCount(int i)
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`setLoopCount` in interface `javax.microedition.media.Player`

addPlayerListener

```
public void  
addPlayerListener(javax.microedition.media.PlayerListener playerListener)
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`addPlayerListener` in interface `javax.microedition.media.Player`

removePlayerListener

```
public void  
removePlayerListener(javax.microedition.media.PlayerListener playerListener)
```

Ver documentación de `javax.microedition.media.Player`

Specified by:

`removePlayerListener` in interface
`javax.microedition.media.Player`

`com.ulpgc.gf.items`

Class ObjetoMovil

```
java.lang.Object  
└─ javax.microedition.lcdui.game.Layer  
    └─ javax.microedition.lcdui.game.Sprite  
        └─ com.ulpgc.gf.items.ObjetoMovil
```

Direct Known Subclasses:

[Disparo](#), [Enemigo](#), [Option](#), [Protagonista](#)

```
public abstract class ObjetoMovil  
extends javax.microedition.lcdui.game.Sprite
```

Superclase de todos los objetos activos en la pantalla, tales como personajes, disparos, y demás objetos móviles.

Author:

Mario Macias Lloret

Field Detail

TIPO_DISPARO

```
public static final int TIPO_DISPARO
```

Indica que la instancia `ObjetoMovil` es un disparo del protagonista

TIPO_ENEMIGO

```
public static final int TIPO_ENEMIGO
```

Indica que la instancia `ObjetoMovil` es un enemigo.

TIPO_OPTION

```
public static final int TIPO_OPTION
```

Indica que la instancia `ObjetoMovil` es un objeto a coger por el protagonista (puntos extra, vidas, etc...)

TIPO_BALAMALA

```
public static final int TIPO_BALAMALA
```

Indica que la instancia `ObjetoMovil` es un disparo de los enemigos

TIPO_OTRO

```
public static final int TIPO_OTRO
```

Indica que la instancia `ObjetoMovil` es cualquier otro tipo de objeto móvil

TIPO_PROTA

```
public static final int TIPO_PROTA
```

Indica que la instancia `ObjetoMovil` es el protagonista del videojuego

Constructor Detail

ObjetoMovil

```
public ObjetoMovil(javax.microedition.lcdui.Image image)
```

Construye un `ObjetoMovil` a partir de una imagen, sin animación.

Parameters:

`image` - La imagen de mapa de bits que representa el objeto

ObjetoMovil

```
public ObjetoMovil(javax.microedition.lcdui.Image image,  
                    int framewidth,  
                    int frameheight)
```

Construye un ObjetoMovil a partir de una imagen, animada

Parameters:

image - La imagen que representa todos los *frames* del objeto animado

framewidth - Tamaño horizontal de los frames, en píxels

frameheight - Tamaño vertical de los frames, en píxels

Method Detail

getTipo

```
public int getTipo()
```

Retorna el tipo del objeto ([TIPO_BALAMALA](#), [TIPO_DISPARO](#), [TIPO_ENEMIGO](#), [TIPO_OPTION](#), [TIPO_OTRO](#) o [TIPO_PROTA](#)).

Returns:

el tipo del ObjetoMovil

setTipo

```
public void setTipo(int tipo)
```

Establece el tipo del objeto ([TIPO_BALAMALA](#), [TIPO_DISPARO](#), [TIPO_ENEMIGO](#), [TIPO_OPTION](#), [TIPO_OTRO](#) o [TIPO_PROTA](#)).

terminar

```
public void terminar()
```

Elimina la ejecución del objeto

matar

```
public void matar()
```

Implementa la muerte del objeto

mover

```
public abstract void mover()
```

Implementa el movimiento del objeto

cargaImagen

```
protected static javax.microedition.lcdui.Image
cargaImagen(java.lang.String imagename)
```

Carga desde el dispositivo de almacenamiento la imagen del ObjetoMovil

Parameters:

imagename - Nombre del archivo que contiene la imagen

Returns:

Objeto Image con la imagen del ObjetoMovil



com.ulpgc.gf.items.options

Class Option

```
java.lang.Object
├── javax.microedition.lcdui.game.Layer
│   ├── javax.microedition.lcdui.game.Sprite
│   │   ├── com.ulpgc.gf.items.ObjetoMovil
│   │   │   └── com.ulpgc.gf.items.options.Option
```

```
public abstract class Option
extends ObjetoMovil
```

Implementa un [ObjetoMovil](#) a coger por el protagonista (puntos extra, vidas, etc...)

Constructor Detail

Option

```
public Option(javax.microedition.lcdui.Image image)
```

Construye el objeto `Option` a partir de una imagen, sin animación.

Parameters:

image - La imagen de mapa de bits que representa el objeto

Option

```
public Option(javax.microedition.lcdui.Image image,
              int framewidth,
              int frameheight)
```

Construye el objeto `Option` a partir de una imagen, animada

Parameters:

`image` - La imagen que representa todos los *frames* del objeto animado

`framewidth` - Tamaño horizontal de los frames, en píxels

`frameheight` - Tamaño vertical de los frames, en píxels

Method Detail

actua


```
public abstract void actua()
```

Implementa la acción a realizar cuando el [Protagonista](#) lo coge.

`com.ulpgc.gf.util`

Class Pantalla

`java.lang.Object`

 `com.ulpgc.gf.util.Pantalla`

```
public class Pantalla  
extends java.lang.Object
```

La función de esta clase es "pre-cachear" algunos calculos referentes a la pantalla y poder obtenerlos con mayor rapidez y facilidad.

Field Detail

TILE_SIZE

```
public static final int TILE_SIZE
```

Constante que indica el tamaño del lado, en píxels, de los *Tiles* cuadrados que conforman el escenario.

Constructor Detail

Pantalla

```
public Pantalla()
```

Method Detail

cachearDatos

```
public static void cachearDatos()
```

Cachea los datos referentes al tamaño de la pantalla y del tablero de juego.

getAnchoPantalla

```
public static int getAnchoPantalla()
```

Returns:

El ancho, en píxels, del tamaño de la pantalla.

getAltoPantalla

```
public static int getAltoPantalla()
```

Returns:

La altura, en píxels, del tamaño de la pantalla.

getAnchoTablero

```
public static int getAnchoTablero()
```

Returns:

La anchura, en píxels, del tamaño del tablero de juego.

getAltoTablero

```
public static int getAltoTablero()
```

Returns:

La altura, en píxels, del tamaño del tablero de juego.



com.ulpgc.gf.items.personajes

Class Protagonista

```
java.lang.Object
```

```
└─ javax.microedition.lcdui.game.Layer
```

```
    └─ javax.microedition.lcdui.game.Sprite
```

```
        └─ com.ulpgc.gf.items.ObjetoMovil
```

```
            └─ com.ulpgc.gf.items.personajes.Protagonista
```

```
public abstract class Protagonista
```

```
extends ObjetoMovil
```

Field Detail

STOP

```
public static final int STOP
```

El protagonista no se mueve

UP

```
public static final int UP
```

El protagonista se mueve hacia arriba

DOWN

```
public static final int DOWN
```

El protagonista se mueve hacia abajo

LEFT

```
public static final int LEFT
```

El protagonista se mueve hacia la izquierda

RIGHT

```
public static final int RIGHT
```

El protagonista se mueve hacia la derecha

Constructor Detail

Protagonista

```
public Protagonista(javax.microedition.lcdui.Image image)
```

Construye el objeto `Protagonista` a partir de una imagen, sin animación.

Parameters:

`image` - La imagen de mapa de bits que representa el objeto

Protagonista

```
public Protagonista(javax.microedition.lcdui.Image image,
                    int framewidth,
                    int frameheight)
```

Construye un objeto `Protagonista` a partir de una imagen, animada

Parameters:

`image` - La imagen que representa todos los *frames* del objeto animado
`framewidth` - Tamaño horizontal de los frames, en píxels
`frameheight` - Tamaño vertical de los frames, en píxels

Method Detail

getPuedeMorir

```
public boolean getPuedeMorir()
```

Indica si el protagonista puede morir al colisionar con un enemigo o un disparo

Returns:

`true` si puede morir. `false` en caso contrario.

mover

```
public abstract void mover()
```

Implementa el movimiento del protagonista

Specified by:

[mover](#) in class [ObjetoMovil](#)

dispara

```
public abstract void dispara()
```

Implementa la acción a realizar cuando el protagonista dispara

addDireccion

```
public void addDireccion(int direccion)
```

Añade una dirección en la cual el objeto se mueve ([STOP](#), [UP](#), [DOWN](#), [LEFT](#) o [RIGHT](#)).

Parameters:

`direccion` - La dirección en la cual se mueve

deleteDireccion

```
public void deleteDireccion(int direccion)
```

Indica que el objeto ha dejado de moverse en una dirección ([STOP](#), [UP](#), [DOWN](#), [LEFT](#) o [RIGHT](#)).

Parameters:

`direccion` - La dirección en la cual ha dejado de moverse

seDirige

```
public boolean seDirige(int direccion)
```

Consulta si el objeto se mueve en una dirección concreta ([STOP](#), [UP](#), [DOWN](#), [LEFT](#) o [RIGHT](#)).

Parameters:

`direccion` - La dirección en la cual nos interesa saber si el objeto se mueve

Returns:

`true` si el objeto se mueve en la dirección consultada. `false` en caso contrario.

Apéndice B. Código fuente del videojuego “Escabechina”

C:\program\WTK22\apps\Escabechina\src\Escabechina.java

```
1  import javax.microedition.lcdui.Command;
2  import javax.microedition.lcdui.CommandListener;
3  import javax.microedition.lcdui.Display;
4  import javax.microedition.lcdui.Displayable;
5  import javax.microedition.midlet.MIDlet;
6  import javax.microedition.midlet.MIDletStateChangeException;
7  import java.util.Timer;
8
9  /* *****
10     ** Escabechina *****
11     *****
12     * Juego creado por Mario Macías Lloret para el proyecto de fin *
13     * de carrera en ingeniería informática: "Creación de juegos para J2ME" *
14     * Universidad de Las Palmas de Gran Canaria *
15     * Curso 2004-2005 *
16     *****
17 */
18 public class Escabechina extends MIDlet implements CommandListener{
19     private Timer timer=null;
20     final static public int ESTADO_PRESENTACION=0;
21     final static public int ESTADO_JUGANDO=1;
22     final static public int ESTADO_PAUSA=2;
23     final static public int ESTADO_GAMEOVER=3;
24
25     final public static int DELAY_MS = 50;
26
27     Display display=Display.getDisplay(this);
28     private int estado=ESTADO_PRESENTACION;
29
30     TareaJuego tareaJuego=null;
31     Juego juego=null;
32
33     protected void startApp() throws MIDletStateChangeException {
34         switch(getEstado()) {
35             case ESTADO_PAUSA:
36             case ESTADO_JUGANDO:
37                 setEstado(ESTADO_PAUSA);
38                 break;
39             default:
40                 setEstado(ESTADO_PRESENTACION);
41                 break;
42         }
43     }
44
45     public void setEstado(int estado) {
46         this.estado=estado;
47         switch(estado) {
48             case ESTADO_PRESENTACION:
49                 PresentacionCanvas canvas=new PresentacionCanvas();
50                 canvas.setCommandListener(this);
51                 display.setCurrent(canvas);
52                 canvas.repaint();
53                 if(timer!=null) {
54                     timer.cancel();
55                     timer=null;
56                 }
57                 tareaJuego=null;
58                 juego=null;
59                 break;
60             case ESTADO_PAUSA:
61                 PausaCanvas pausa=new PausaCanvas();
62                 pausa.setCommandListener(this);
63                 display.setCurrent(pausa);
64                 break;
65             case ESTADO_JUGANDO:
66                 if(tareaJuego==null) {
```

Creación de videojuegos con J2ME

```
67         tareaJuego=new TareaJuego(this);
68     }
69     if(juego==null) {
70         juego=new Juego(tareaJuego);
71     }
72     juego.setCommandListener(this);
73     if(timer==null) {
74         timer=new Timer();
75         timer.schedule(tareaJuego,DELAY_MS,DELAY_MS);
76     }
77     display.setCurrent(juego);
78     break;
79     case ESTADO_GAMEOVER:
80         juego.removeCommand(juego.comandoPausa);
81         juego.addCommand(new Command("Salir",Command.STOP,1));
82         return;
83     }
84     System.gc(); //Garbage collection manual
85 }
86
87 public int getEstado() {
88     return estado;
89 }
90
91 protected void pauseApp() {
92     if(estado==ESTADO_JUGANDO) {
93         setEstado(ESTADO_PAUSA);
94     }
95 }
96
97 protected void destroyApp(boolean b) throws MIDletStateChangeException {
98 }
99
100 public void commandAction(Command command, Displayable displayable) {
101     try {
102         if(estado==ESTADO_PRESENTACION) {
103             switch(command.getCommandType()) {
104                 case Command.OK:
105                     setEstado(ESTADO_JUGANDO);
106                     break;
107                 case Command.EXIT:
108                     this.notifyDestroyed();
109                     break;
110             }
111         } else if(estado==ESTADO_JUGANDO) {
112             if(command.getCommandType()==Command.STOP) {
113                 setEstado(ESTADO_PAUSA);
114             }
115         } else if(estado==ESTADO_GAMEOVER) {
116             if(command.getCommandType()==Command.STOP) {
117                 setEstado(ESTADO_PRESENTACION);
118             }
119         } else if(estado==ESTADO_PAUSA) {
120             switch(command.getCommandType()) {
121                 case Command.OK:
122                     setEstado(ESTADO_JUGANDO);
123                     break;
124                 case Command.EXIT:
125                     setEstado(ESTADO_PRESENTACION);
126                     break;
127             }
128         }
129     }
130
131     catch(Exception e) {
132         e.printStackTrace();
133     }
134 }
135
136 }
137
```


C:\program\WTK22\apps\Escabechina\src\Juego.java

```

1
2 import javax.microedition.lcdui.Canvas;
3 import javax.microedition.lcdui.Command;
4 import javax.microedition.lcdui.Graphics;
5 import javax.microedition.lcdui.Image;
6
7 /* *****
8  ** Escabechina *****
9  *****
10  * Juego creado por Mario Macías Lloret para el proyecto de fin *
11  * de carrera en ingeniería informática: "Creación de juegos para J2ME" *
12  * Universidad de Las Palmas de Gran Canaria *
13  * Curso 2004-2005 *
14  *****
15 */
16 public class Juego extends Canvas {
17     private static Image pers[]; //Imágenes de los personajes
18     private static Image persout[]; //Imágenes de los personajes cuando les han pegado
19     private static Image hoyoback=null;
20     private static Image hoyofront=null;
21     private static Image numeros[];
22     private static Image guante=null;
23     private static Image estrella=null;
24     private static Image fin=null;
25
26
27     public final static int CELDA_WIDTH=26;
28     public final static int CELDA_HEIGHT=18;
29
30     private static Image pagina=null;
31
32     TareaJuego tarea=null;
33     public Command comandoPausa=null;
34     public Juego(TareaJuego tarea) {
35         try {
36             this.tarea=tarea;
37             estrella=Image.createImage("/estrella.png");
38
39             fin=Image.createImage("/fin.png");
40
41             pers=new Image[4];
42             pers[0]=Image.createImage("/cerdo.png");
43             pers[1]=Image.createImage("/tio.png");
44             pers[2]=Image.createImage("/pingu.png");
45             pers[3]=Image.createImage("/perro.png");
46
47             persout=new Image[4];
48             persout[0]=Image.createImage("/cerdopegao.png");
49             persout[1]=Image.createImage("/tiopegao.png");
50             persout[2]=Image.createImage("/pingupegao.png");
51             persout[3]=Image.createImage("/perropegao.png");
52
53             numeros=new Image[10];
54             for(int i=0;i<10;i++) {
55                 numeros[i]=Image.createImage("/numero_"+i+".png");
56             }
57
58             hoyoback=Image.createImage("/bujerofondo.png");
59             hoyofront=Image.createImage("/bujerofrente.png");
60
61             guante=Image.createImage("/guante.png");
62
63             pagina=Image.createImage(CELDA_WIDTH*3+18,CELDA_HEIGHT*3);
64
65             this.addCommand(comandoPausa=new Command("Pausar",Command.STOP,1));
66         } catch(Exception e) {
67             e.printStackTrace();
68         }
69     }
70
71     protected void paint(Graphics graphics) {

```

Creación de videojuegos con J2ME

```
73     Graphics pg=pagina.getGraphics();
74     int hoyo=0;
75     for(int y=0;y<3;y++) {
76         for(int x=0;x<3;x++) {
77             pg.drawImage(hoyoback,x*CELDA_WIDTH,y*CELDA_HEIGHT,0);
78             if(tarea.ocupado[hoyo]) {
79                 Image tmpimg=null;
80                 if(tarea.herido[hoyo]) {
81                     tmpimg=persout[tarea.personaje[hoyo]];
82                 } else {
83                     tmpimg=pers[tarea.personaje[hoyo]];
84                 }
85
86                 pg.drawImage(tmpimg,x*CELDA_WIDTH,y*CELDA_HEIGHT+tarea.altura[hoyo],0);
87             }
88             pg.drawImage(hoyofront,x*CELDA_WIDTH,y*CELDA_HEIGHT,0);
89
90             if(tarea.posicionguante==(hoyo+1) &&
tarea.programa.getEstado()==Escabechina.ESTADO_JUGANDO ) {
91                 pg.drawImage(guante,x*CELDA_WIDTH+4-tarea.framesguante*2,y*CELDA_HEIGHT+4-
tarea.framesguante*2,0);
92             } else if( tarea.programa.getEstado()==Escabechina.ESTADO_GAMEOVER ) {
93                 pg.drawImage(fin,14,10,0);
94             }
95
96             hoyo++;
97         }
98     }
99 }
100
101 //dibujar estrellas
102 for(int i=0;i<4;i++) {
103     if(tarea.xstar[i]>0 && tarea.xstar[i]<(26*3) && tarea.ystar[i]<18*3) {
104         pg.drawImage(estrella,tarea.xstar[i],tarea.ystar[i],0);
105     }
106 }
107
108 //dibujar el marcador
109 int pt=tarea.puntos;
110 for(int i=0;i<3;i++) {
111     pg.drawImage(numeros[pt%10],26*3,18*(2-i),0);
112     pt=pt/10;
113 }
114
115 graphics.drawImage(pagina,0,0,0);
116
117 }
118
119 protected void keyPressed(int i) {
120     super.keyPressed(i);
121     switch(i) {
122         /* para cuando se juega con el teclado del ordena (emulador) */
123
124         case Canvas.KEY_NUM1:
125             tarea.posicionguante=7;
126             tarea.framesguante=0;
127             break;
128         case Canvas.KEY_NUM2:
129             tarea.posicionguante=8;
130             tarea.framesguante=0;
131             break;
132         case Canvas.KEY_NUM3:
133             tarea.posicionguante=9;
134             tarea.framesguante=0;
135             break;
136         case Canvas.KEY_NUM4:
137             tarea.posicionguante=4;
138             tarea.framesguante=0;
139             break;
140         case Canvas.KEY_NUM5:
141             tarea.posicionguante=5;
142             tarea.framesguante=0;
143             break;
144         case Canvas.KEY_NUM6:
145             tarea.posicionguante=6;
146             tarea.framesguante=0;
```

```

147         break;
148     case Canvas.KEY_NUM7:
149         tarea.posicionguante=1;
150         tarea.framesguante=0;
151         break;
152     case Canvas.KEY_NUM8:
153         tarea.posicionguante=2;
154         tarea.framesguante=0;
155         break;
156     case Canvas.KEY_NUM9:
157         tarea.posicionguante=3;
158         tarea.framesguante=0;
159         break;
160
161     }
162 }
163 }
164

```

C:\program\WTK22\apps\Escabechina\src\PausaCanvas.java

```

1
2 import javax.microedition.lcdui.Image;
3 import javax.microedition.lcdui.Command;
4 import javax.microedition.lcdui.Graphics;
5 import javax.microedition.lcdui.Canvas;
6
7 /* *****
8  ** Escabechina *****
9  *****
10  * Juego creado por Mario Macías Lloret para el proyecto de fin *
11  * de carrera en ingeniería informática: "Creación de juegos para J2ME" *
12  * Universidad de Las Palmas de Gran Canaria *
13  * Curso 2004-2005 *
14  *****
15 */
16
17 public class PausaCanvas extends Canvas {
18     private Image pausa = null;
19     public PausaCanvas() {
20         try {
21             pausa=Image.createImage("/pausa.png");
22         }
23         catch(Exception e) {
24             e.printStackTrace();
25         }
26         this.addCommand(new Command("Continuar", Command.OK, 1));
27         this.addCommand(new Command("Salir", Command.EXIT, 0));
28     }
29
30
31
32     protected void paint(Graphics graphics) {
33         graphics.setColor(0,0,0);
34         graphics.fillRect(0,0,this.getWidth(),this.getHeight());
35         graphics.drawImage(pausa,0,0,0);
36     }
37
38 }
39

```

C:\program\WTK22\apps\Escabechina\src\PresentacionCanvas.java

```

1
2 import javax.microedition.lcdui.Canvas;
3 import javax.microedition.lcdui.Command;
4 import javax.microedition.lcdui.Graphics;

```

Creación de videojuegos con J2ME

```
5  import javax.microedition.lcdui.Image;
6
7  /* *****
8      ** Escabechina *****
9      *****
10     * Juego creado por Mario Macías Lloret para el proyecto de fin *
11     * de carrera en ingeniería informática: "Creación de juegos para J2ME" *
12     * Universidad de Las Palmas de Gran Canaria *
13     * Curso 2004-2005 *
14     *****
15 */
16 public class PresentacionCanvas extends Canvas {
17     private Image presentacion = null;
18     public PresentacionCanvas() {
19         try {
20             presentacion=Image.createImage("/presentacion.png");
21         }
22         catch(Exception e) {
23             e.printStackTrace();
24         }
25         this.addCommand(new Command("Jugar",Command.OK,1));
26         this.addCommand(new Command("Salir",Command.EXIT,0));
27     }
28
29
30
31     protected void paint(Graphics graphics) {
32         graphics.setColor(0,0,0);
33         graphics.fillRect(0,0,this.getWidth(),this.getHeight());
34         graphics.drawImage(presentacion,0,0,0);
35     }
36 }
37
```

C:\program\WTK22\apps\Escabechina\src\TareaJuego.java

```
1  import java.util.TimerTask;
2  import java.util.Random;
3  /* *****
4      ** Escabechina *****
5      *****
6      * Juego creado por Mario Macías Lloret para el proyecto de fin *
7      * de carrera en ingeniería informática: "Creación de juegos para J2ME" *
8      * Universidad de Las Palmas de Gran Canaria *
9      * Curso 2004-2005 *
10     *****
11 */
12
13 public class TareaJuego extends TimerTask {
14     private boolean pausado=false;
15     Escabechina programa=null;
16     public final int ESTADO_SUBE = 0;
17     public final int ESTADO_PARADO = 1;
18     public final int ESTADO_BAJA = 2;
19
20     static Random rnd=new Random(System.currentTimeMillis());
21
22     int vidas=3;
23     int golpesseguidos=0;
24
25     int nivel=0;
26     int puntos=0;
27     boolean ocupado[]=new boolean[9]; //indica si un agujero esta ocupado por algun personaje
28     int estado[]=new int[9]; //indica qué hace el muñeco (sube, baja, esta quieto...)
29     boolean herido[]=new boolean[9]; //si true, al personaje le han dado
30
31     int maxtmparriba=1300; //inicialmente el muñeco esta 1 segundo parado
32     int tmparriba[]=new int[9]; //indica cuanto tiempo lleva un muñeco arriba
33
34     int velocidad=1; //velocidad a la que suben y bajan los muñecos
35     int altura[]=new int[9]; //altura a la que estan los personajes
36     int personaje[]=new int[9]; //cual de los 4 personajes es (0..3)

```

```

37
38     public final int margen= 8; //Indica a qué distancia en pixels de su maxima altura se puede
golpear a un muñeco
39
40     int intervalo=3000; //El intervalo en ms entre que aparece un muñeco o otro
41     int tactual=2300; //cuenta el tiempo que lleva, cuando tactual>intervalo --> tactual=0; e
intervalo--;
42
43     int posicionguante=0; //Nº de agujero donde esta golpeando el guante (0=no golpea)
44     int maxframesguante=3; //Nº de frames en que el guante esta visible
45     int framesguante=0;
46
47     //datos para visualizar las estrellitas cuando se golpea
48     int xstar[]=new int[4]; //posicion x
49     int vxstar[]=new int[4]; //velocidad x (-4..4)
50     int ystar[]=new int[4]; //posicion y
51     int vystar[]=new int[4]; //velocidad y (-4..infinito) <-- usaremos aceleracion de la
gravedad
52
53     public TareaJuego(Escabechina programa) {
54         pausado=false;
55         this.programa=programa;
56         for(int i=0;i<9;i++) {
57             ocupado[i]=false;
58         }
59     }
60     public void run() {
61         //Esta parte del IF hace el Game Over
62         if(programa.getEstado()==Escabechina.ESTADO_GAMEOVER) {
63             for(int i=0;i<9;i++) {
64                 if(!ocupado[i]) {
65                     ocupado[i]=true;
66                     personaje[i]=i%4;
67                     estado[i]=ESTADO_SUBE;
68                     herido[i]=false;
69                     altura[i]=18;
70                     break;
71                 } else {
72                     if(estado[i]==ESTADO_SUBE) {
73                         if(altura[i]>0) {
74                             altura[i]--;
75                         } else {
76                             estado[i]=ESTADO_BAJA;
77                         }
78                     } else {
79                         if(altura[i]<2) {
80                             altura[i]++;
81                         } else {
82                             estado[i]=ESTADO_SUBE;
83                         }
84                     }
85                 }
86             }
87         } else if(programa.getEstado()==Escabechina.ESTADO_JUGANDO) {
88             //Movimiento de las estrellas
89             for(int i=0;i<4;i++) {
90                 if(xstar[i]>0 && xstar[i]<(26*3) && ystar[i]<18*3) {
91                     xstar[i]+=vxstar[i];
92                     ystar[i]+=vystar[i];
93                     vystar[i]+=1;
94                 }
95             }
96             //control del guante
97             if(posicionguante>0) {
98                 framesguante++;
99                 if((altura[posicionguante-1]-margen)<0 && ocupado[posicionguante-1] && !
herido[posicionguante-1]) {
100                 //preparamos los datos de las estrellas
101                 for(int i=0;i<4;i++) {
102                     vxstar[i]=rnd.nextInt()%4;
103                     vystar[i]=rnd.nextInt()%6;
104                     xstar[i]=(posicionguante-1)%3)*26+13;
105                     ystar[i]=(posicionguante-1)/3)*18+7;
106                 }
107

```

Creación de videojuegos con J2ME

```
108         herido[posicionguante-1]=true;
109
110         golpesseguidos++;
111         if(golpesseguidos==20) {
112             vidas=vidas==10?10:vidas+1;
113             golpesseguidos=0;
114         }
115         //estado[posicionguante-1]=ESTADO_BAJA;
116         puntos++;
117     }
118     if(framesguante>maxframesguante) {
119         framesguante=0;
120         posicionguante=0;
121     }
122 }
123
124 tactual+=Escabechina.DELAY_MS;
125 //Mirar si hay que sacar algun muñeco nuevo
126 if(tactual>intervalo) {
127     tactual=10;
128
129     if(intervalo<800) {
130         intervalo-=45;
131         if(intervalo<(400-nivel)) {
132             //cada vez que sobrepase el intervalo, subimos el nivel un poco
133             intervalo=700-nivel;
134             if(nivel<105) {
135                 nivel+=15;
136             }
137         }
138     } else {
139         intervalo-=intervalo/7;
140     }
141     velocidad=(3-(intervalo/700));
142     if(velocidad<1) velocidad=1;
143
144     maxtmparriba-=10;
145     if(maxtmparriba<150) {
146         maxtmparriba=150;
147     }
148     int hoyo=Math.abs(rnd.nextInt()%9);
149     int hoyosocupados=0;
150     while(ocupado[hoyo] && hoyosocupados<9) {
151         hoyo=(hoyo+1)%9;
152         hoyosocupados=(hoyosocupados+1)%9;
153     }
154     //cuando vaya muy rapido, todos los hoyos pueden estar ocupados
155     //de esta manera, esperamos al siguiente frame, hasta que haya alguno libre
156     if(hoyosocupados<9) {
157         tactual=0;
158         estado[hoyo]=ESTADO_SUBE;
159         herido[hoyo]=false;
160         ocupado[hoyo]=true;
161         altura[hoyo]=18;
162         personaje[hoyo]=Math.abs(rnd.nextInt()%4);
163         tmparriba[hoyo]=0;
164     }
165 }
166
167 //Controlar los que ya hay moviendose
168 for(int hoyo=0;hoyo<9;hoyo++) {
169     if(ocupado[hoyo]) {
170         switch(estado[hoyo]) {
171             case ESTADO_SUBE:
172                 altura[hoyo]-=velocidad;
173                 if(altura[hoyo]<0) {
174                     altura[hoyo]=0;
175                     estado[hoyo]=ESTADO_PARADO;
176                 }
177                 break;
178             case ESTADO_PARADO:
179                 tmparriba[hoyo]+=Escabechina.DELAY_MS;
180                 if(tmparriba[hoyo]>maxtmparriba) {
181                     estado[hoyo]=ESTADO_BAJA;
182                 }
183                 break;
```

```
184         case ESTADO_BAJA:
185             altura[hoyo]+=velocidad;
186             if(altura[hoyo]>18) {
187                 altura[hoyo]=18;
188                 ocupado[hoyo]=false;
189                 if(!herido[hoyo]) {
190                     vidas--;
191                     golpesseguidos=0;
192                     if(vidas==0) {
193                         programa.setEstado(Escabechina.ESTADO_GAMEOVER);
194                     }
195                 }
196             }
197             break;
198         }
199     }
200 }
201 }
202 programa.juego.repaint();
203 }
204
205 }
206
```


Apéndice C. Código fuente del videojuego de demostración

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\DemoFinal.java

```
package com.ulpgc.mmacias;

import com.ulpgc.mmacias.util.Utils;
import com.ulpgc.mmacias.util.sound.GestorSonidos;
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Font;
import java.io.IOException;

public class DemoFinal implements CommandListener {

    private DemoJuego midlet;

    private Pantalla pantalla;

    private Image imagen;

    public DemoFinal(DemoJuego midlet) {
        this.midlet=midlet;
        try {
            imagen=Image.createImage("/final.png");
        } catch(IOException e) {
            System.out.println("Error cargando imagen final.png: "+e.getMessage());
        }

        Letreros letreros=new Letreros();
        pantalla=letreros;
        Display.getDisplay(midlet).setCurrent(letreros);
        letreros.setCommandListener(this);
        letreros.addCommand(new Command("Finalizar",Command.SCREEN,1));
        int index=GestorSonidos.singleton().cargaSonido("/final.mid", "audio/midi", false);
        GestorSonidos.singleton().tocaSonido(index);
        Thread thread=new Thread(letreros);
        thread.run();
    }

    private abstract class Pantalla extends Canvas {
        protected boolean salir;

        public Pantalla() {
            salir=false;
        }

        public void salir() {
            salir=true;
        }
    }

    private class Letreros extends Pantalla implements Runnable {
        private final int numero=30;
        private int color[],x[],y[],incx[],incy[];

        private int colfondo,inccolfondo;
        public Letreros() {
            super();
            color=new int[numero];
            x=new int[numero];
            y=new int[numero];
            incx=new int[numero];
            incy=new int[numero];

            //inicializar datos de los graficos
            colfondo=0;inccolfondo=5;
        }
    }
}
```

Creación de videojuegos con J2ME

```
        for(int i=0;i<numero;i++) {
            color[i]=Utils.random(255*255*255);
            x[i]=Utils.random(getWidth());
            y[i]=Utils.random(getHeight());
            incx[i]=Utils.random(7)-4;
            incy[i]=Utils.random(7)-4;
        }
    }

    protected void paint(Graphics graphics) {
        graphics.setColor(128,255-colfondo,colfondo);
        graphics.fillRect(0,0,getWidth(),getHeight());
        graphics.setFont(Font.getFont(Font.FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_MEDIUM));
        for(int i=0;i<numero;i++) {
            graphics.setColor(color[i]);
            graphics.drawString("Fin",x[i],y[i],Graphics.HCENTER|Graphics.TOP);
        }

        graphics.setColor(196-colfondo/2,64+colfondo/2,128);
        graphics.setFont(Font.getFont(Font.FACE_SYSTEM,Font.STYLE_BOLD,Font.SIZE_LARGE));
        graphics.drawString("¡Felicidades!",getWidth()/2,10,Graphics.HCENTER|Graphics.TOP);

        graphics.drawImage(imagen,getWidth()/2,getHeight()/2,Graphics.HCENTER|Graphics.VCENTER);
    }

    public void run() {
        while(!salir) {
            colfondo+=inccolfondo;
            if(colfondo>=255) {
                inccolfondo=-inccolfondo;
                colfondo=255;
            } else if(colfondo<=0) {
                inccolfondo=-inccolfondo;
                colfondo=0;
            }

            for(int i=0;i<numero;i++) {
                x[i]+=incx[i];
                if(x[i]<0||x[i]>getWidth()) {
                    incx[i]=-incx[i];
                }
                y[i]+=incy[i];
                if(y[i]<0||y[i]>getHeight()) {
                    incy[i]=-incy[i];
                }
            }
            repaint();
            try {
                Thread.sleep(30);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
        midlet.setEstado(DemoJuego.ESTADO_PRESENTACION);
    }

    public void commandAction(Command command, Displayable displayable) {
        if(command.getCommandType()==Command.SCREEN) {
            GestorSonidos.singleton().eliminaSonidos();
            pantalla.salir();
        }
    }
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\DemoJuego.java

```
package com.ulpgc.mmacias;
```

```
import com.ulpgc.mmacias.util.*;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
```

```

public class DemoJuego extends MIDlet {
    public final static int ESTADO_PRESENTACION = 0;
    public final static int ESTADO_PAUSA = 1;
    public final static int ESTADO_JUEGO = 2;
    public final static int ESTADO_SALIENDO = 3;
    public final static int ESTADO_GAMEOVER = 4;
    public final static int ESTADO_FIN = 5;

    private int estado=ESTADO_PRESENTACION;
    private GestorJuego gj=null;
    private Presentacion presentacion =null;

    protected void startApp() throws MIDletStateChangeException {
        switch(estado) {
            case ESTADO_JUEGO:
            case ESTADO_PAUSA:
                setEstado(ESTADO_PAUSA);
                break;
            case ESTADO_PRESENTACION:
                Pantalla.cachearDatos();
                setEstado(ESTADO_PRESENTACION);
                break;
        }
    }

    protected void pauseApp() {
        if(estado==ESTADO_JUEGO) {
            setEstado(ESTADO_PAUSA);
        }
    }

    protected void destroyApp(boolean b) throws MIDletStateChangeException {
        setEstado(ESTADO_SALIENDO);
        GestorElementos.Singleton().destroy();
    }

    public int getEstado() {
        return estado;
    }

    public void setEstado(int estado) {
        this.estado = estado;
        switch(estado) {
            case ESTADO_PRESENTACION:
                presentacion=new Presentacion(this);
                presentacion.setEstado(Presentacion.ESTADO_IMAGEN);
                break;
            case ESTADO_FIN:
                new DemoFinal(this);
                break;
            case ESTADO_JUEGO:
                gj.pausar(false);
                break;
            case ESTADO_PAUSA:
                gj.pausar(true);
                break;
        }
    }

    public void iniciaPartida() {
        presentacion=null;
        GestorElementos.Singleton().destroy();
        gj=new GestorJuego(this);
        gj.iniciaPartida();
    }
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\Presentacion.java

```
package com.ulpgc.mmacias;
```

```
import com.ulpgc.mmacias.util.Configuracion;
```

Creación de videojuegos con J2ME

```
import javax.microedition.lcdui.*;
import java.io.IOException;

public class Presentacion implements CommandListener {

    public final static int ESTADO_IMAGEN=0;
    public final static int ESTADO_PRINCIPAL=1;
    public final static int ESTADO OPCIONES=2;
    public final static int ESTADO_AYUDA=3;

    private int estado;
    private DemoJuego midlet=null;

    public final static String OPCION_JUGAR="Jugar";
    public final static String OPCION OPCIONES="Opciones";

    private String[] opcionesMenu={OPCION_JUGAR,OPCION OPCIONES};
    List principal=null;

    private Presentacion() {
        principal=new List("Menú principal",List.IMPLICIT,opcionesMenu,null);
        principal.setCommandListener(this);
    }
    public Presentacion(DemoJuego midlet) {
        this();
        this.midlet=midlet;
    }

    public void setEstado(int nuevoestado) {
        estado=nuevoestado;
        switch(estado) {
            case ESTADO_IMAGEN:
                Display.getDisplay(midlet).setCurrent(new ImagenPresentacion());
                break;
            case ESTADO_PRINCIPAL:
                Display.getDisplay(midlet).setCurrent(principal);
                break;
            case ESTADO OPCIONES:
                Display.getDisplay(midlet).setCurrent(new FormularioOpciones());
                break;
        }
    }

    public void commandAction(Command command, Displayable displayable) {
        if(displayable==principal) {
            if(command==List.SELECT_COMMAND) {
                String label=principal.getString(principal.getSelectedIndex());
                if(label.equals(OPCION_JUGAR)) {
                    midlet.iniciaPartida();
                } else if(label.equals(OPCION OPCIONES)) {
                    setEstado(ESTADO OPCIONES);
                }
            }
        }
    }

    private class FormularioOpciones extends Form implements ItemStateListener , CommandListener {
        private ChoiceGroup sonido;
        private ChoiceGroup vidas;

        public FormularioOpciones() {
            super("Formulario de opciones");
            Configuracion config=Configuracion.singleton();

            sonido=new ChoiceGroup("Sonido",Choice.POPUP,new String[]{"On", "Off"},null);
            sonido.setSelectedIndex(0,config.isMusic());
            sonido.setSelectedIndex(1,!config.isMusic());
            this.append(sonido);

            vidas=new ChoiceGroup("Numero de vidas",Choice.POPUP,new String[]{"3", "5", "99"},null);
            switch(config.getVidasIniciales()) {
                case 3:
                    vidas.setSelectedIndex(0,true);
                    break;
                case 5:

```

```

        vidas.setSelectedIndex(1,true);
        break;
    default:
        vidas.setSelectedIndex(2,true);
        break;
    }
    this.append(vidas);

    this.addCommand(new Command("Volver",Command.SCREEN,1));
    this.setItemStateListener(this);
    this.setCommandListener(this);
}

public void itemStateChanged(Item item) {
    Configuracion config=Configuracion.singleton();
    if(item.getLabel().equals("Sonido")) {
        ChoiceGroup cg=(ChoiceGroup) item;
        config.setMusic(cg.getSelectedIndex()==0);
    } else if(item.getLabel().equals("Numero de vidas")) {
        ChoiceGroup cg=(ChoiceGroup) item;
        config.setVidasIniciales(Integer.parseInt(cg.getString(cg.getSelectedIndex())));
    }
}

public void commandAction(Command command, Displayable displayable) {
    if(command.getLabel().equals("Volver")) {
        setEstado(ESTADO_PRINCIPAL);
    }
}
}

//Clases auxiliares para la presentación
private class ImagenPresentacion extends Canvas {
    private Image imagen_presentacion=null;
    public ImagenPresentacion() {
        try {
            imagen_presentacion=Image.createImage("/presentacion.png");
        } catch (IOException e) {
            System.err.println("Error cargando imagen_presentacion 'presentacion.png':
"+e.getMessage());
        }
    }
    protected void paint(Graphics graphics) {
        graphics.setColor(210,180,255);
        graphics.fillRect(0,0,getWidth(),getHeight());
        graphics.drawImage(imagen_presentacion,getWidth()/2,getHeight()/2,Graphics.VCENTER|
Graphics.HCENTER);
    }
    protected void keyPressed(int i) {
        super.keyPressed(i);
        if(estado==ESTADO_IMAGEN) {
            setEstado(ESTADO_PRINCIPAL);
        }
    }
}
}
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\fases\Fase.java

```

package com.ulpgc.mmacias.fases;

import com.ulpgc.mmacias.util.GestorElementos;
import com.ulpgc.mmacias.util.Pantalla;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.LayerManager;
import javax.microedition.lcdui.game.TiledLayer;
import java.io.IOException;

public abstract class Fase extends LayerManager {

    private boolean acabada;
    protected int posx=0;

```

```

protected TiledLayer frente=null;
protected TiledLayer fondo=null;

public Fase(String imgMapaFrente,String imgMapaFondo) {
    super();
    GestorElementos ge=GestorElementos.singleton();
    ge.iniciaDatos();
    this.append(ge.getProtagonista());
    acabada=false;
    setViewWindow(0,0,Pantalla.getAnchoPantalla(),Pantalla.getAltoTablero());

    byte mapaFrente[][]=getMapaFrente();
    try {
        frente=new
TiledLayer(mapaFrente[0].length,mapaFrente.length,Image.createImage(imgMapaFrente),Pantalla.TILE_SIZ
E,Pantalla.TILE_SIZE);
    } catch(IOException e) {
        System.out.println("Error cargando imagen "+imgMapaFrente+": "+e.getMessage());
    }
    rellenarTiles(frente,mapaFrente);
    frente.setPosition(0,16*5);
    frente.setVisible(true);
    this.append(frente);

    byte mapaFondo[][]=getMapaFondo();
    try {
        fondo=new
TiledLayer(mapaFondo[0].length,mapaFondo.length,Image.createImage(imgMapaFondo),Pantalla.TILE_SIZE,P
antalla.TILE_SIZE);
    } catch(IOException e) {
        System.out.println("Error cargando imagen "+imgMapaFondo+": "+e.getMessage());
    }
    rellenarTiles(fondo,mapaFondo);
    fondo.setPosition(0,0);
    fondo.setVisible(true);
    this.append(fondo);

    setViewWindow(0,0,Pantalla.getAnchoTablero(),Pantalla.getAltoTablero());
}

protected void rellenarTiles(TiledLayer tl,byte[][] tilemap) {
    for(int rows=0;rows<tilemap.length;rows++) {
        for(int cols=0;cols<tilemap[rows].length;cols++) {
            tl.setCell(cols,rows,tilemap[rows][cols]);
        }
    }
}

public void paint(Graphics g) {
    try {
        if(g!=null) {
            paint(g,0,0);
        }
    } catch(NullPointerException e) {
        System.out.println(e.getMessage());
    }
}

public void setAcabada(boolean acabada) {
    this.acabada=acabada;
}

public boolean isAcabada() {
    return acabada;
}

public void mover() {
    //Si no se ha llegado al final de la pantalla
    if((posx+Pantalla.getAnchoTablero())<frente.getColumns()*Pantalla.TILE_SIZE-5) {
        posx+=3;
        //Semaforo para no cambiar la posicion de la pantalla mientras se esta dibujando
        //asi se evitan efectos indeseables
        frente.setPosition(-posx,Pantalla.TILE_SIZE*5);
        fondo.setPosition(-posx/4,0);
    }
};

```

```

abstract public void finalizar();

abstract protected byte[][] getMapaFrente();
abstract protected byte[][] getMapaFondo();
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\fases\Fase1.java

```

package com.ulpgc.mmacias.fases;

import com.ulpgc.mmacias.items.personajes.*;
import com.ulpgc.mmacias.util.GestorElementos;
import com.ulpgc.mmacias.util.Pantalla;
import com.ulpgc.mmacias.util.sound.GestorSonidos;

public class Fase1 extends Fase {
    //siguiente pixel donde aparecera algun elemento del array "objetos[]"
    private int siguiente;

    private int musicaFase1, musicaElvis;

    public Fase1() {
        super("/fase1.png", "/fase1fondo.png");

        siguiente=objetos[OC++] * Pantalla.TILE_SIZE - Pantalla.getAnchoTablero();

        //pre-cachear los sonidos (cuidado! puede agotar toda la memoria)

        musicaFase1=GestorSonidos.singleton().cargaSonido("/fase1.mid", "audio/midi", true);
        musicaElvis=GestorSonidos.singleton().cargaSonido("/elvis.mid", "audio/midi", true);
        GestorSonidos.singleton().tocaSonido(musicaFase1);
    }

    public void finalizar() {
        GestorSonidos.singleton().paraSonido();
        GestorSonidos.singleton().eliminaSonidos();
    }

    private int OC=0; //contador de objetos
    public void mover() {
        super.mover();
        while(posx>=siguiente) {
            GestorElementos ge=GestorElementos.singleton();
            switch(objetos[OC++]) {
                case 0:
                    ge.añade(new Caca(objetos[OC++]));
                    break;
                case 1:
                    ge.añade(new Comecocos(objetos[OC++]));
                    break;
                case 2:
                    ge.añade(new Cerdo());
                    break;
                case 3:
                    ge.añade(new NotaMusical());
                    break;
                case 4:
                    ge.añade(new Elvis(this));
                    GestorSonidos.singleton().tocaSonido(musicaElvis);
                    break;
            }
            siguiente=objetos[OC++] * Pantalla.TILE_SIZE - Pantalla.getAnchoTablero();
        }
    }

    protected byte[][] getMapaFrente() {
        return mapa;
    }

    protected byte[][] getMapaFondo() {
        return mapafondo;
    }
}

```

- 144 -


```

23,24,23,24,23,24,27,28,22,21,22,21,22,21,22,21,22,21,22,21,22,21,22,21,22,
21,22,21,22,21,25,26,25,26,25,26,25,26,25,26,25,26,22,21,22,21,22,21,
22,21,22,21,22,21,22,21,22,21,22,21,25,26,22,21,27,28,22,21,22,21,25,26,22,
21,27,28,22,21,22}};

```

```

private static final byte mapafondo[][]= //75x10
{{ 1, 1, 1, 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 4, 5, 1, 1, 1, 1, 2, 3, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 1, 1,
 1},
{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
{ 1, 6, 7, 8, 1, 9,10,11,12, 1, 1, 6, 7, 8, 1, 1, 6, 7, 8, 1, 1, 6, 7, 8, 1,
 1, 4, 5, 9,10,11,12, 1, 1, 6, 7, 8, 1, 1, 1, 1, 6, 7, 8, 1, 1, 6, 7, 8,
 1, 4, 5,10,11,12, 1, 1, 6, 7, 8, 4, 5, 6, 7, 8, 1, 1, 6, 7, 8, 1, 9,10,11},
{ 9,13,14,15,16, 9,13,14,15,16, 9,13,14,15,16, 9,13,14,15,16, 9,13,14,15,16,
 1, 1, 1, 9,13,14,15,16, 9,13,14,15,16, 1, 1, 1,17,18,14,15,16, 9,13,14,15,
16, 1, 9,13,14,15,16,17,18,14,15,16, 9,13,14,15,16,17,18,14,15,16, 9,13,14},
{19,20,21,22,23,19,24,25,26,23,19,20,21,22,23,19,20,21,22,23,19,20,21,22,23,
 1, 1, 1,19,24,25,26,23,19,20,21,22,23,27,28, 1,29,30,21,22,23,19,20,21,22,
23, 1,19,24,25,26,23,29,30,21,22,23,19,20,21,22,23,29,30,21,22,23,19,24,25},
{31,32,33,34,35,31,32,33,34,35,31,32,33,34,36,37,32,33,34,36,37,32,33,34,35,
 1, 1, 1,31,32,33,34,36,37,32,33,34,35,38,39, 1,31,32,33,34,40,37,32,33,34,
35, 1,31,32,33,34,40,37,32,33,34,36,37,32,33,34,36,37,32,33,34,35,31,32,33},
{41,42,43,44,45,41,42,43,44,45,41,42,43,46,47,48,49,43,46,47,48,49,43,44,45,
 1, 1, 1,41,42,43,46,47,48,49,43,44,45, 1, 1, 1,41,42,43,46,50,48,49,43,44,
45, 1,41,42,43,46,50,48,49,43,46,47,48,49,43,46,47,48,49,43,44,45,41,42,43},
{51,52,53,54,55,51,52,53,54,55,51,52,53,56,57,58,59,53,56,57,58,59,53,54,55,
 1, 1, 1,51,52,53,56,57,58,59,53,54,55, 1, 4, 5,51,52,53,56,57,58,59,53,54,
55, 1,51,52,53,56,57,58,59,53,56,57,58,59,53,56,57,58,59,53,54,55,51,52,53},
{60,61,62,63,64,60,61,62,63,64,60,61,62,65,66,67,68,62,65,66,67,68,62,63,64,
 1, 1, 1,60,61,62,65,66,67,68,62,63,64, 1, 1, 1,60,61,62,65,66,67,68,62,63,
64, 1,60,61,62,65,66,67,68,62,65,66,67,68,62,65,66,67,68,62,63,64,60,61,62},
{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}};

```

```

}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\ObjetoMovil.java

```

package com.ulpgc.mmacias.items;

import com.ulpgc.mmacias.util.GestorElementos;

import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.Sprite;
import java.io.IOException;

public abstract class ObjetoMovil extends Sprite {
    public static final int TIPO_DISPARO = 0;
    public static final int TIPO_ENEMIGO = 1;
    public static final int TIPO_OPTION = 2;
    public static final int TIPO_BALAMALA = 3;
    public static final int TIPO_OTRO = 4;
    public static final int TIPO_PROTA = 5;

    protected int tipo;

    public ObjetoMovil(Image image) {
        super(image);
    }

    public ObjetoMovil(Image image, int framewidth, int frameheight) {
        super(image, framewidth, frameheight);
    }

    public int getTipo() {
        return tipo;
    }
}

```

Creación de videojuegos con J2ME

```
public void setTipo(int tipo) {
    this.tipo = tipo;
}

public void terminar() {
    GestorElementos.singleton().elimina(this);
}

public void matar() {
    terminar();
}

abstract public void mover();

protected static Image cargaImagen(String imagename) {
    Image image=null;
    try {
        image=Image.createImage(imagename);
    } catch(IOException e) {
        System.err.println("Error cargando imagen "+imagename+": "+e.getMessage());
    }
    return image;
}
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\armas\Disparo.java

```
package com.ulpgc.mmacias.items.armas;

import com.ulpgc.mmacias.items.ObjetoMovil;

import javax.microedition.lcdui.Image;

public abstract class Disparo extends ObjetoMovil {

    public Disparo(Image image) {
        super(image);
        this.tipo=TIPO_DISPARO;
    }
    abstract public int getFuerza();
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\armas\DisparoSimple.java

```
package com.ulpgc.mmacias.items.armas;

import com.ulpgc.mmacias.util.Pantalla;

import javax.microedition.lcdui.Image;

public class DisparoSimple extends Disparo {
    private static Image image=null;
    private static final int ANCHO=8;
    private static final int ALTO=8;

    static {
        image=cargaImagen("/disparo.png");
    }
    public DisparoSimple(int x, int y) {
        super(image);
        this.setPosition(x,y);
        defineReferencePixel(ANCHO/2,ALTO/2);
    }
    public void mover() {
        if(getX()>Pantalla.getAnchoTablero()) {
            terminar();
        }
        move(15,0);
    }
}
```

```

    }

    public int getFuerza() {
        return 1;
    }
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\options\Option.java

```

package com.ulpgc.mmacias.items.options;

import com.ulpgc.mmacias.items.ObjetoMovil;

import javax.microedition.lcdui.Image;

public abstract class Option extends ObjetoMovil {
    public Option(Image image) {
        super(image);
    }

    public Option(Image image, int framewidth, int frameheight) {
        super(image, framewidth, frameheight);
    }

    //hace lo que quiera cuando el protagonista lo obtiene
    public abstract void actua();
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\otros\BalaMala.java

```

package com.ulpgc.mmacias.items.otros;

import com.ulpgc.mmacias.items.ObjetoMovil;
import com.ulpgc.mmacias.util.Pantalla;

import javax.microedition.lcdui.Image;

public class BalaMala extends ObjetoMovil {
    private static final int VELOCIDAD=5;
    private static final int ANCHO=8;
    private static final int ALTO=8;

    private static Image image=null;
    static {
        if(image==null) {
            image=cargaImagen("/balamala.png");
        }
    }

    //haremos que la bala se dirija al punto mediante el algoritmo de bresenham.
    double x,y,incx,incy;

    public BalaMala(int fromx, int fromy, int tox, int toy) {
        super(image);
        this.tipo=TIPO_BALAMALA;
        this.setPosition(fromx,fromy);
        defineReferencePixel(ANCHO/2,ALTO/2);

        x=fromx; y=fromy;

        //Aproximacion rapida y rastrea de sqrt(lado_largo+lado_corto) --> lado_largo
        +5/16*lado_corto
        //Normalizaremos los vectores y los multiplicamos por la velocidad para conocer el
        incremento que deben
        //tener las balas en todo momento
        //Se hace así por que java no tiene ni SQRT, SIN, COS, etc... para hacerlo mediante otros
        métodos más precisos
        int L1=Math.abs(tox-fromx);
        int L2=Math.abs(toy-fromy);
    }
}

```

Creación de videojuegos con J2ME

```
        int max,min;
        if(Math.abs(L1)>Math.abs(L2)) {
            max=L1; min=L2;
        } else {
            max=L2; min=L1;
        }
        double modulo=Math.abs(16*max+5*min)/16;
        incx=L1/modulo*VELOCIDAD;
        incy=L2/modulo*VELOCIDAD;
        if(fromx>tox) incx=-incx;
        if(fromy>toy) incy=-incy;
    }
    public void mover() {
        if(getX()>Pantalla.getAnchoTablero() || getX()<=-ANCHO || getY()<=-ALTO ||
        getY()>Pantalla.getAltoTablero()) {
            terminar();
        }
        x+=incx; y+=incy;
        setPosition((int)x,(int)y);
    }
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\otros\Explosion.java

```
package com.ulpgc.mmacias.items.otros;

import com.ulpgc.mmacias.items.ObjetoMovil;
import javax.microedition.lcdui.Image;

public class Explosion extends ObjetoMovil{
    private static final int WIDTH=16;
    private static final int HEIGHT=16;

    private static Image image=null;

    private int time=0;

    static {
        image=cargaImagen("/explosion.png");
    }

    public Explosion(int x,int y) {
        super(image,WIDTH,HEIGHT);
        tipo=TIPO_OTRO;
        setPosition(x,y);
    }

    public void mover() {
        if(++time>4) {
            terminar();
        }
        nextFrame();
    }
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\Caca.java

```
package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.otros.Explosion;
import com.ulpgc.mmacias.util.Pantalla;
import com.ulpgc.mmacias.util.Utils;
import com.ulpgc.mmacias.util.GestorElementos;

import javax.microedition.lcdui.Image;
```

```

public class Caca extends Enemigo {
    private static final int WIDTH=16;
    private static final int HEIGHT=16;

    private static Image image=null;

    private int estado=0;
    private int time=0;

    private int disparo=Utils.random(Pantalla.getAnchoTablero());
    private boolean disparado=false;

    static {
        image=cargaImagen("/caca.png");
    }

    private Caca() {
        super(image,WIDTH,HEIGHT);
        defineReferencePixel(WIDTH/2,HEIGHT/2);
    }

    public Caca(int altura) {
        this();
        setPosition(Pantalla.getAnchoTablero()+WIDTH,altura);
    }

    public void mover() {
        if(time++%2==0) {
            this.nextFrame();
        }

        switch(estado) {
            case 0:
                move(-6,0);
                int x=getX();
                if(x<disparo && !disparado) {
                    dispara();
                    disparado=true;
                }
                if(getX()<10) {
                    if(getY()<Pantalla.getAltoTablero()/2) {
                        estado=1;
                    } else {
                        estado=2;
                    }
                }
                break;
            case 1:
                move(4,2);
                if(getX()>Pantalla.getAnchoTablero() || getY()<-WIDTH) {
                    terminar();
                }
                break;
            case 2:
                move(4,-2);
                if(getX()>Pantalla.getAnchoTablero() || getY()>Pantalla.getAltoTablero()) {
                    terminar();
                }
                break;
        }
    }

    public void matar() {
        super.matar();
        GestorElementos.singleton().añade(new Explosion(getX(),getY()));
    }

    public void tocar(Disparo disparo) {
        matar();
    }

    public int getPuntuacion() //devuelve el numero de puntos a sumar cuando muere el enemigo
    {
        return 5;
    }
}

```

}

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\Cerdo.java

```
package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.otros.Expllosion;
import com.ulpgc.mmacias.util.GestorElementos;
import com.ulpgc.mmacias.util.Pantalla;
import com.ulpgc.mmacias.util.Utills;

import javax.microedition.lcdui.Image;

public class Cerdo extends Enemigo {
    private int energia=3;
    private static final int ANCHO=32;
    private static final int ALTO=24;
    private static Image image=null;
    private static final int VELOCIDAD=-4;
    private int REBOTE=-13-Utills.random(6);

    private int accel=0;

    static {
        image=cargaImagen("/cerdo.png");
    }

    public Cerdo() {
        super(image, ANCHO, ALTO);
        defineReferencePixel (ANCHO/2, ALTO/2);
        setPosition (Pantalla.getAnchoTablero()+ANCHO, Utills.random(Pantalla.getAltoTablero()-ANCHO));
    }

    public void matar() {
        super.matar();
        GestorElementos ge=GestorElementos.singleton();
        int x=getX(), y=getY();
        ge.añade(new Expllosion(x-ANCHO/4,y));
        ge.añade(new Expllosion(x-3*ANCHO/4,y));
        terminar();
    }

    public void tocar(Disparo disparo) {
        energia-=disparo.getFuerza();
        if(energia<=0) {
            matar();
        }
    }

    public int getPuntuacion() //devuelve el numero de puntos a sumar cuando muere el enemigo
    {
        return 12;
    }

    public void mover() {
        accel+=2;
        move(VELOCIDAD, accel);
        if(accel==0) {
            dispara();
        }
        if(getY()>Pantalla.getAltoTablero()-ANCHO) {
            accel=REBOTE;
        }
        if(getX()<-ANCHO) {
            terminar();
        }
        if(accel>0) {
            setFrame(0);
        } else {
            setFrame(1);
        }
    }
}
```

```

    }
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\Comeccos.java

```

package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.otros.Expllosion;
import com.ulpgc.mmacias.util.Pantalla;
import com.ulpgc.mmacias.util.Utills;
import com.ulpgc.mmacias.util.GestorElementos;

import javax.microedition.lcdui.Image;

public class Comeccos extends Enemigo {
    private static final int ANCHO=16;
    private static final int ALTO=16;

    private static Image image=null;

    private int dirx=-7,diry=0;
    boolean sube=true;
    private int time=0;
    private int siguienteCambio=10+Utills.random(20);

    static {
        image=cargaImagen("/comeccos.png");
    }

    private Comeccos() {
        super(image,ANCHO,ALTO);
        defineReferencePixel(ANCHO/2,ALTO/2);
    }

    public Comeccos(int altura) {
        this();
        setPosition(Pantalla.getAnchoTablero()+ANCHO,altura);
    }

    public void mover() {
        int x=getX(), y=getY();
        if(time++%2==0) {
            this.nextFrame();
        }
        if(time>siguienteCambio) {
            cambiaDireccion();
            siguienteCambio+=5+Utills.random(5);
        }

        if(y<=0) {
            setPosition(x,y=0);
            cambiaDireccion();
        } else if(y>Pantalla.getAltoTablero()-ALTO) {
            setPosition(x,Pantalla.getAltoTablero()-ANCHO);
            cambiaDireccion();
        }

        if(getX()<=-ANCHO) {
            terminar();
        }
        move(dirx,diry);
    }

    private void cambiaDireccion() {
        if(dirx==0) {
            dirx=-5;
            diry=0;
        } else {
            dirx=0;
            diry=Utills.random(2)==0?-5:5;
        }
    }
}

```

Creación de videojuegos con J2ME

```
    }

    if(dirx!=0) {
        setTransform(TRANS_NONE);
    } else {
        if(diry<0) {
            setTransform(TRANS_ROT90);
        } else {
            setTransform(TRANS_ROT270);
        }
    }
}

public void tocar(Disparo disparo) {
    matar();
}

public void matar() {
    super.matar();
    GestorElementos.singleton().añade(new Explosion(getX(),getY()));
}

public int getPuntuacion() //devuelve el numero de puntos a sumar cuando muere el enemigo
{
    return 7;
}

}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\Elvis.java
a

```
package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.ObjetoMovil;
import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.otros.BalaMala;
import com.ulpgc.mmacias.util.GestorElementos;
import com.ulpgc.mmacias.util.Pantalla;
import com.ulpgc.mmacias.fases.Fase;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.Sprite;

public class Elvis extends Enemigo {
    public static final int ANCHO = 40;
    public static final int ALTO = 56;
    public static final Image image=ObjetoMovil.cargaImagen("/elvis.png");

    private int estado=0;
    private int energia=30;
    private int tiempo=0;
    private int direccion=15;

    private Fase fase;

    public Elvis(Fase laFase) {
        super(image,ANCHO,ALTO);
        this.fase=laFase;
        tipo=TIPO_ENEMIGO;
        defineReferencePixel(ANCHO/2,ALTO/2);
        setPosition(Pantalla.getAnchoTablero(),Pantalla.getAltoTablero()-Pantalla.TILE_SIZE/2-ALTO);
    }

    public void mover() {
        tiempo++;
        switch(estado) {
            case 0:
                move(-3,0);
                if(tiempo%4==0) {
                    if(getFrame()==0) {
                        setFrame(1);
                    }
                }
            }
        }
    }
}
```



```

        } else {
            setFrame(0);
        }
    }
    if(tiempo%20==0) {
        GestorElementos ge=GestorElementos.singleton();
        Protagonista prota=ge.getProtagonista();
        ge.añade(new BalaMala(getX()+ANCHO/2,getY()+15,prota.getX(),prota.getY()));
    }
    if(getX()<Pantalla.getAnchoTablero()/2) {
        estado=1;
        tiempo=0;
    }
    break;
case 1:
    if(tiempo%2==0) {
        if(getFrame()==2) {
            setFrame(3);
        } else {
            setFrame(2);
        }
    }
    if(tiempo%15==0) {
        GestorElementos ge=GestorElementos.singleton();
        ge.añade(new NotaMusical());
    }
    if(tiempo>90) {
        estado=2;
        tiempo=0;
    }
    break;
case 2:
    if(tiempo%2==0) {
        if(getFrame()==0) {
            setFrame(1);
        } else {
            setFrame(0);
        }
    }
    move(direccion,0);
    if(getX()<0 || getX()>Pantalla.getAnchoTablero()-ANCHO) {
        direccion=-direccion;
        if(direccion>0) {
            setTransform(Sprite.TRANS_MIRROR);
        } else {
            setTransform(Sprite.TRANS_NONE);
        }
    }
    if(tiempo%15==0) {
        GestorElementos ge=GestorElementos.singleton();
        Protagonista prota=ge.getProtagonista();
        ge.añade(new BalaMala(getX(),getY()-ALTO/4,prota.getX(),prota.getY()));
    }
    if(tiempo>85) {
        estado=1;
        tiempo=0;
    }
    break;
case 3:
    move(5,direccion++);
    switch((tiempo++)%4) {
        case 0:
            setTransform(TRANS_NONE);
            break;
        case 1:
            setTransform(TRANS_ROT270);
            break;
        case 3:
            setTransform(TRANS_ROT180);
            break;
        case 4:
            setTransform(TRANS_ROT90);
            break;
    }
    if(getX()>Pantalla.getAnchoTablero()) {
        super.matar();
    }

```

```

        fase.setAcabada(true);
    }
    break;
}

}

public void tocar(Disparo disparo) {
    energia-=disparo.getFuerza();
    if(energia<0) {
        matar();
    }
}

public void matar() {
    tipo=TIPO_OTRO;
    estado=3;
    direccion=-12;
    tiempo=0;
}

public int getPuntuacion() //devuelve el numero de puntos a sumar cuando muere el enemigo
{
    return 60;
}
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\Enemigo.java

```

package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.ObjetoMovil;
import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.otros.BalaMala;
import com.ulpgc.mmacias.util.GestorElementos;
import javax.microedition.lcdui.Image;

public abstract class Enemigo extends ObjetoMovil {
    public Enemigo(Image image) {
        super(image);
        this.tipo=TIPO_ENEMIGO;
    }

    public Enemigo(Image image, int ancho, int alto) {
        super(image, ancho, alto);
        this.tipo=TIPO_ENEMIGO;
    }

    public void matar() {
        GestorElementos.singleton().acumulaPuntuacion(getPuntuacion());
        terminar();
    }

    protected void dispara() {
        GestorElementos ge=GestorElementos.singleton();
        Protagonista p=ge.getProtagonista();
        ge.añade(new BalaMala(getX(),getY(),p.getX(),p.getY()));
    }

    public abstract void tocar(Disparo disparo);
    public abstract int getPuntuacion(); //devuelve el numero de puntos a sumar cuando muere el
    enemigo
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\NotaMusical.java

```

package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.otros.Expllosion;
import com.ulpgc.mmacias.util.GestorElementos;
import com.ulpgc.mmacias.util.Pantalla;
import com.ulpgc.mmacias.util.Utills;

import javax.microedition.lcdui.Image;

public class NotaMusical extends Enemigo {
    public static final int ANCHO = 16;
    public static final int ALTO = 16;

    private static Image image=null;
    static {
        image=cargaImagen("/nota.png");
    }

    private boolean atacando=false;
    private int time=0;
    private int accelx=0;
    private int accely=0;

    public NotaMusical() {
        super(image,ANCHO,ALTO);
        this.tipo=TIPO_OTRO;
        defineReferencePixel(ANCHO/2,ALTO/2);
        setPosition(Utills.random(Pantalla.getAnchoTablero()),Utills.random(Pantalla.getAltoTablero()));
    };

    }

    public void tocar(Disparo disparo) {
        matar();
    }

    public void mover() {
        time++;
        if(atacando) {
            Protagonista prota=GestorElementos.singleton().getProtagonista();
            int x=getX(), y=getY();
            if(time%5==0) {
                accelx+=x>prota.getX()?-1:+1;
                accely+=y>prota.getY()?-1:+1;
            }
            if(x<-ANCHO || y <-ALTO || x>Pantalla.getAnchoTablero()+ANCHO ||
y>Pantalla.getAltoTablero()+ALTO) {
                terminar();
            } else {
                nextFrame();
                move(accelx,accely);
            }
        } else {
            setVisible(time%2==0?true:false);
            if(time>13) {
                setVisible(true);
                atacando=true;
                tipo=TIPO_ENEMIGO;
            }
        }
    }

    }

    public int getPuntuacion() //devuelve el numero de puntos a sumar cuando muere el enemigo
    {
        return 10;
    }

    public void matar() {
        super.matar();
        GestorElementos.singleton().añade(new Expllosion(getX(),getY()));
    }

    }

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\items\personajes\Protagonista.java

```
package com.ulpgc.mmacias.items.personajes;

import com.ulpgc.mmacias.items.ObjetoMovil;
import com.ulpgc.mmacias.items.armas.DisparoSimple;
import com.ulpgc.mmacias.items.otros.Expllosion;
import com.ulpgc.mmacias.util.GestorElementos;
import com.ulpgc.mmacias.util.Pantalla;

import javax.microedition.lcdui.Image;

public class Protagonista extends ObjetoMovil {
    static Image image=null;

    private boolean muriendo,saliendo;
    private int veloc_caida=0; //para cuando esta muriendo
    private int tiempo=0;

    private static final int ANCHO=24;
    private static final int ALTO=16;

    public static final int STOP = 0;
    public static final int UP = 1;
    public static final int DOWN = 2;
    public static final int LEFT = 4;
    public static final int RIGHT = 8 ;

    private int velocidad = 45;
    private int errorveloc = 0; // como iremos dividiendo la velocidad entre 10 para dar mas precision,
                                // acumulamos el error para no perderla

    private int direccion;
    private boolean disparando;
    static {
        image=cargaImagen("/protagonista.png");
    }

    public Protagonista() {
        super(image,ANCHO,ALTO);
        tipo=TIPO_PROTA;
        defineReferencePixel(ANCHO/2,ALTO/2);
        setSaliendo();
        disparando=false;
    }

    private void setSaliendo() {
        muriendo=false;
        saliendo=true;
        setPosition(20,50);
        tiempo=0;
        setVisible(false);
    }

    public void mover() {
        if(disparando) {
            disparando=false;
            GestorElementos.singleton().añade(new DisparoSimple(getX(),getY()+ALTO/2));
        }
        if(muriendo) {
            move(0,++veloc_caida);
            if(veloc_caida<8) {
                setFrame(0);
            } else {
                setFrame(1);
            }
        }
        if(veloc_caida%3==0) {
            GestorElementos.singleton().añade(new Expllosion(getX(),getY()));
        }
        if(getY()>Pantalla.getAltoTablero()) {
            setSaliendo();
        }
    }
}
```

```

    }
} else {
    if(saliendo) {
        setVisible((tiempo%2)==0?true:false);
        if(tiempo++>24) {
            setVisible(true);
            saliendo=false;
        }
    }
    errorveloc=(errorveloc+velocidad)%10;
    int v=(errorveloc+velocidad)/10;

    if((direccion & UP)!=0) {
        setFrame(2);
        if(getY()>0) {
            move(0,-v);
        }
    } else if((direccion & DOWN)!=0) {
        setFrame(1);
        if(getY()<(Pantalla.getAltoTablero()-ALTO)) {
            move(0,v);
        }
    } else {
        setFrame(0);
    }

    if((direccion & LEFT)!=0) {
        if(getX()>0) {
            move(-v,0);
        }
    } else if((direccion & RIGHT)!=0) {
        if(getX()<Pantalla.getAnchoTablero()-30) {
            move(v,0);
        }
    }
}
}

public void matar() {
    if(puedeMorir()) {
        muriendo=true;
        veloc_caida=0;
    }
}

public boolean isMuriendo() {
    return muriendo;
}

public boolean puedeMorir() {
    return !muriendo && !saliendo;
}

public void addDireccion(int direccion) {
    this.direccion=this.direccion | direccion;
}

public void deleteDireccion(int direccion) {
    this.direccion=this.direccion & ~direccion;
}

public void dispara() {
    disparando=true;
}
}

```

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\sound\GestorSonidos.
java

```

```
package com.ulpgc.mmacias.util.sound;
```

Creación de videojuegos con J2ME

```
import com.ulpgc.mmacias.util.Configuracion;

import javax.microedition.media.MediaException;
import java.util.Vector;

public class GestorSonidos {
    private static GestorSonidos gm=null;
    private Vector sonidos=new Vector(5);

    private Integer index_tocando=null;

    private GestorSonidos() {
    }

    public static GestorSonidos singleton() {
        if(gm==null) {
            gm=new GestorSonidos();
        }
        return gm;
    }

    public void eliminaSonidos() {
        if(sonidos!=null) {
            for(int i=0;i<sonidos.size();i++) {
                MediaPlayer sonido=((MediaPlayer)sonidos.elementAt(i));
                sonido.stop();
                sonido.close();
            }
            index_tocando=null;
            sonidos=new Vector(5);
        }
    }

    //Retorna la posición del vector donde está el sonido
    public int cargaSonido(String filename, String type, boolean repeat) {
        MediaPlayer sonido=new MediaPlayer(filename,type,repeat);
        try {
            sonido.realize();
        } catch(MediaException e) {
            System.out.println("Error cargando el sonido: "+e.getMessage());
        }
        sonidos.addElement(sonido);
        return sonidos.indexOf(sonido);
    }

    public void tocaSonido(int index) {
        paraSonido();
        if(Configuracion.singleton().isMusic()) {
            ((MediaPlayer)sonidos.elementAt(index)).start();
            index_tocando=new Integer(index);
        }
    }

    public Integer getIndexSonidoActual() {
        return index_tocando;
    }

    public void paraSonido() {
        if(index_tocando!=null) {
            ((MediaPlayer)sonidos.elementAt(index_tocando.intValue())).stop();
            index_tocando=null;
        }
    }
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\sound\MediaPlayer.java

```
package com.ulpgc.mmacias.util.sound;

import javax.microedition.media.*;
import java.io.IOException;
```

```

public class MediaPlayer implements Player {
    private Player thePlayer=null;

    private long mediaTime; // La idea es pre-cachear el mediaTime y la duración al
                                // inicio de la pantalla, ya que su cálculo requiere muchísimo tiempo
    private long duration;

    public MediaPlayer(String filename, String contenttype, boolean repeat) {
        try {
            thePlayer=Manager.createPlayer(getClass().getResourceAsStream(filename),contenttype);
            mediaTime=thePlayer.getMediaTime();
            duration=thePlayer.getDuration();
            if(repeat) {
                thePlayer.addPlayerListener(
                    new PlayerListener() {
                        public void playerUpdate(Player player, String s, Object o) {
                            if(s.equals(PlayerListener.END_OF_MEDIA)) {
                                try {
                                    player.start();
                                } catch(MediaException e) {
                                    System.out.println("Error re-tocando la musiqueti:
"+e.getMessage());
                                }
                            }
                        }
                    }
                );
            }
            } catch(MediaException e) {
                System.out.println(filename+" - El Player no puede ser creado para dicho Stream y tipo:
"+e.getMessage());
            } catch(IOException e) {
                System.out.println(filename+" - Hubo un problema leyendo los datos del InputStream:
"+e.getMessage());
            }
        }

        public void start() {
            try {
                thePlayer.start();
            } catch(MediaException e) {
                System.out.println("Error tocando la musiqueti: "+e.getMessage());
            }
        }

        public Control[] getControls() {
            return thePlayer.getControls();
        }

        public Control getControl(String s) {
            return thePlayer.getControl(s);
        }

        public void realize() throws MediaException {
            thePlayer.realize();
        }

        public void prefetch() throws MediaException {
            thePlayer.prefetch();
        }

        public void stop() {
            try {
                thePlayer.stop();
            } catch(MediaException e) {
                System.out.println("Error parando la musica: "+e.getMessage());
            }
        }

        public void deallocate() {
            thePlayer.deallocate();
        }

        public void close() {
            thePlayer.close();
        }
    }

```

Creación de videojuegos con J2ME

```
}

public void setTimeBase(TimeBase timeBase) throws MediaException {
    thePlayer.setTimeBase(timeBase);
}

public TimeBase getTimeBase() {
    return thePlayer.getTimeBase();
}

public long setMediaTime(long l) throws MediaException {
    return thePlayer.setMediaTime(l);
}

public long getMediaTime() {
    return mediaTime;
}

public int getState() {
    return thePlayer.getState();
}

public long getDuration() {
    return duration;
}

public String getContentType() {
    return thePlayer.getContentType();
}

public void setLoopCount(int i) {
    thePlayer.setLoopCount(i);
}

public void addPlayerListener(PlayerListener playerListener) {
    thePlayer.addPlayerListener(playerListener);
}

public void removePlayerListener(PlayerListener playerListener) {
    thePlayer.removePlayerListener(playerListener);
}
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\Configuracion.java

```
package com.ulpgc.mmacias.util;
```

```
public class Configuracion {
    private static Configuracion config=null;
    private boolean music;
    private int vidas;

    private Configuracion() {
        //Configuración por defecto
        music=false;
        vidas=3;
    }

    static public Configuracion singleton() {
        if(config==null) {
            config=new Configuracion();
        }
        return config;
    }

    public boolean isMusic() {
        return music;
    }

    public void setMusic(boolean music) {
        this.music = music;
    }
}
```



```

    public int getVidasIniciales() {
        return vidas;
    }

    public void setVidasIniciales(int vidas) {
        this.vidas = vidas;
    }
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\GestorElementos.java

```

package com.ulpgc.mmacias.util;

import com.ulpgc.mmacias.fases.Fase;
import com.ulpgc.mmacias.fases.FaseI;
import com.ulpgc.mmacias.items.ObjetoMovil;
import com.ulpgc.mmacias.items.personajes.Protagonista;

public class GestorElementos {
    private class Nodo {
        public ObjetoMovil objeto=null;
        public Nodo siguiente=null;
    }

    private Nodo actual[];
    private Nodo primero=null;

    private static GestorElementos ge=null;

    private Fase fase=null;
    private Protagonista prota=null;

    private int puntuacion_acumulada=0;

    public int getSize() {
        return fase.getSize();
    }

    public void iniciaDatos() {
        prota=new Protagonista();
        puntuacion_acumulada=0;

        actual=new Nodo[2]; actual[0]=null; actual[1]=null;
    }

    // estas dos funciones retornan la puntuación a sumar, como
    // resultado de la muerte de los enemigos
    public void acumulaPuntuacion(int puntos) {
        puntuacion_acumulada+=puntos;
    }

    public int desacumulaPuntuacion() {
        int tmp=puntuacion_acumulada;
        puntuacion_acumulada=0;
        return tmp;
    }

    public Protagonista getProtagonista() {
        return prota;
    }

    public static GestorElementos singleton() {
        if(ge==null) {
            ge=new GestorElementos();
        }
        return ge;
    }

    public void destroy() {

```

Creación de videojuegos con J2ME

```
ge=null;
if(fase!=null) {
    fase.finalizar();
}
fase=null;
actual=null;
primero=null;
prota=null;
}

//Se añaden al principio de la cola para que las acciones "mover"
//de los objetos no se realicen hasta la siguiente iteración, cuando
//se llame al método "reinicia"
public void añade(ObjetoMovil objetoMovil) {
    fase.insert(objetoMovil,0);
    if(primero==null) {
        primero=new Nodo();
        primero.objeto=objetoMovil;
    } else {
        Nodo tmp=new Nodo();
        tmp.siguiente=primero;
        tmp.objeto=objetoMovil;
        primero=tmp;
    }
}

public boolean reinicia(int n) {
    if(primero==null) {
        return false;
    }
    actual[n]=primero;
    return true;
}

// en las siguientes 2 funciones, "n" indica el número de
// referencia al nodo guardado en el array "actual"
public boolean siguiente(int n) {
    if(actual[n]==null || actual[n].siguiente==null) {
        return false;
    } else {
        actual[n]=actual[n].siguiente;
        return true;
    }
}

public ObjetoMovil getObjetoMovil(int n) {
    if(actual[n]==null) return null;
    return actual[n].objeto;
}

public void elimina(ObjetoMovil objetoMovil) {
    fase.remove(objetoMovil);

    Nodo recorr=primero;
    Nodo anterior=null;
    boolean encontrado=false;

    if(primero.objeto==objetoMovil) {
        primero=primero.siguiente;
    } else {
        while(!encontrado && recorr!=null) {
            if(recorr.objeto==objetoMovil) {
                if(anterior!=null) {
                    anterior.siguiente=recorr.siguiente;
                }
                encontrado=true;
            } else {
                anterior=recorr;
                recorr=recorr.siguiente;
            }
        }
    }
}

public void cargaFase(int num) {
    switch(num) {
```

```

        case 1:
            fase=new Fase1();
            break;
    }
}

public Fase getFase() {
    return fase;
}

public void liberaFase() {
    fase=null;
}
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\GestorGrafico.java

```

package com.ulpgc.mmacias.util;

import com.ulpgc.mmacias.fases.Fase;
import com.ulpgc.mmacias.items.personajes.Protagonista;

import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.GameCanvas;

public class GestorGrafico extends GameCanvas {
    private GestorElementos ge=GestorElementos.singleton();
    private GestorJuego elGestorJuego=null;
    private Font fuente=null;
    Image marcador=null;
    private int puntosanteriores=0,vidasanteriores=0;

    private GestorGrafico() {
        super(false);
        fuente=Font.getFont(Font.FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_MEDIUM);
        marcador=Image.createImage(Pantalla.getAnchoPantalla(),Pantalla.getAltoPantalla()-
Pantalla.getAltoTablero());
    }

    public GestorGrafico(GestorJuego gestorJuego) {
        this();
        elGestorJuego=gestorJuego;
        this.setCommandListener(gestorJuego);
    }

    public void paint(Graphics graphics) {
        Fase fase=ge.getFase();
        if(fase!=null) {
            fase.paint(graphics);
        } else {
            System.out.println("Fase es null");
        }

        if(elGestorJuego!=null) {

            //Hacemos esto para redibujar el área del marcador solo cuando éste haya cambiado
            //Así ahorraremos recursos de procesador
            if(puntosanteriores!=elGestorJuego.getPuntos() || vidasanteriores!
=elGestorJuego.getVidas()) {
                puntosanteriores=elGestorJuego.getPuntos();
                vidasanteriores=elGestorJuego.getVidas();
                int ant=Pantalla.getAnchoTablero();
                int anp=Pantalla.getAnchoPantalla();
                int alt=Pantalla.getAltoTablero();
                int alp=Pantalla.getAltoPantalla();

                graphics.setColor(30,30,200);
                graphics.fillRect(0,alt,anp,alp-alt);
            }
        }
    }
}

```

Creación de videojuegos con J2ME

```
        graphics.setColor(90,90,255);
        graphics.drawRect(0,alt,anp,alp-alt);

        graphics.setFont(fuente);
        graphics.setColor(255,255,255);
        graphics.drawString("Puntos: "+puntosanteriores,4,alt+2,Graphics.TOP|Graphics.LEFT);
        graphics.drawString("Vidas: "+vidasanteriores,anp-4,alt+2,Graphics.TOP|
Graphics.RIGHT);
    }
    } else {
        System.out.println("elGestorJuego es null");
    }
}

protected void keyPressed(int i) {
    super.keyPressed(i);
    Protagonista prota=GestorElementos.singleton().getProtagonista();
    switch(getGameAction(i)) {
        case Canvas.UP:
            prota.addDireccion(Protagonista.UP);
            break;
        case Canvas.DOWN:
            prota.addDireccion(Protagonista.DOWN);
            break;
        case Canvas.LEFT:
            prota.addDireccion(Protagonista.LEFT);
            break;
        case Canvas.RIGHT:
            prota.addDireccion(Protagonista.RIGHT);
            break;
        case Canvas.FIRE:
            prota.dispara();
            break;
    }
}

protected void keyReleased(int i) {
    super.keyReleased(i);
    Protagonista prota=GestorElementos.singleton().getProtagonista();
    switch(getGameAction(i)) {
        case Canvas.UP:
            prota.deleteDireccion(Protagonista.UP);
            break;
        case Canvas.DOWN:
            prota.deleteDireccion(Protagonista.DOWN);
            break;
        case Canvas.LEFT:
            prota.deleteDireccion(Protagonista.LEFT);
            break;
        case Canvas.RIGHT:
            prota.deleteDireccion(Protagonista.RIGHT);
            break;
    }
}
}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\GestorJuego.java

```
package com.ulpgc.mmacias.util;

import com.ulpgc.mmacias.fases.Fase;
import com.ulpgc.mmacias.items.ObjetoMovil;
import com.ulpgc.mmacias.items.options.Option;
import com.ulpgc.mmacias.items.armas.Disparo;
import com.ulpgc.mmacias.items.personajes.Enemigo;
import com.ulpgc.mmacias.items.personajes.Protagonista;
import com.ulpgc.mmacias.DemoJuego;
import com.ulpgc.mmacias.util.sound.GestorSonidos;
import javax.microedition.lcdui.*;

public class GestorJuego implements Runnable, CommandListener {
    private GestorElementos ge=GestorElementos.singleton();
```

```

private GestorGrafico gg=new GestorGrafico(this);
private DemoJuego midlet=null;

private boolean letrero_puesto;
private final static int FRAMES_SEGUNDO=10;

private long tiempoactual,tiempoanterior;
private long iteraciones;

private int vidas;
private int puntos;

private int tiempo;
private int musicaGameOver;

private Integer sonido_tocando;

private Command cmdContinuar=new Command("Continuar",Command.SCREEN,1);
private Command cmdPausar=new Command("Pausar",Command.SCREEN,1);

public GestorJuego(DemoJuego midlet) {
    this.midlet=midlet;
    Display.getDisplay(midlet).setCurrent(gg);
    tiempoanterior=System.currentTimeMillis();
    iteraciones=0;
    vidas=Configuracion.singleton().getVidasIniciales();
    letrero_puesto=false;
    puntos=0;
}

public void run() {
    tiempoanterior=System.currentTimeMillis();
    iteraciones=0;
    //long frames=0;
    int estado=midlet.getEstado();
    while(estado==DemoJuego.ESTADO_JUEGO || estado==DemoJuego.ESTADO_GAMEOVER ||
estado==DemoJuego.ESTADO_PAUSA) {
        if(ge.getFase().isAcabada()) {
            ge.getFase().finalizar();
            midlet.setEstado(DemoJuego.ESTADO_FIN);
        } else {
            if(estado!=DemoJuego.ESTADO_PAUSA) {
                mover();
            }
        }
        iteraciones++;

        //implementación del frame skipping
        //queremos 15 frames por segundo, por lo tanto, se deberá llamar a la
        //función mover() cada 1000/15 ms. Si no se puede alcanzar dicha cuota,
        //se omitirá la función repaint (la mas lenta) hasta que pueda ser
        //alcanzada. Si se alcanza de sobras, haremos "dormir" al thread los segundos
        //sobrantes
        tiempoactual=System.currentTimeMillis();
        long idealiter=FRAMES_SEGUNDO*(tiempoactual-tiempoanterior)/1000;
        if(iteraciones>=idealiter){
            gg.repaint();
            //frames++;
            try {
                long tiemposiguiente=(iteraciones+1)*1000/FRAMES_SEGUNDO+tiempoanterior;
                tiempoactual=System.currentTimeMillis();
                //System.out.println("jar: "+(tiemposiguiente-tiempoactual));
                long diferencia=tiemposiguiente-tiempoactual;
                if(diferencia>0) {
                    Thread.sleep(diferencia);
                }
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
        estado=midlet.getEstado();
    }
}

public void iniciaPartida() {
    midlet.setEstado(DemoJuego.ESTADO_JUEGO);
}

```

Creación de videojuegos con J2ME

```
GestorElementos.singleton().cargaFase(1);
musicaGameOver=GestorSonidos.singleton().cargaSonido("/gameover.mid","audio/midi",false);
Thread thread=new Thread(this);
thread.setPriority(Thread.MAX_PRIORITY);
thread.start();
}

public void commandAction(Command command, Displayable displayable) {
    if(command==cmdPausar) {
        midlet.setEstado(DemoJuego.ESTADO_PAUSA);
    } else if(command==cmdContinuar) {
        midlet.setEstado(DemoJuego.ESTADO_JUEGO);
    }
}

public void pausar(boolean pausa) {
    if(gg==null) return;
    if(pausa) {
        gg.removeCommand(cmdPausar);
        gg.addCommand(cmdContinuar);
        sonido_tocando=GestorSonidos.singleton().getIndexSonidoActual();
        GestorSonidos.singleton().paraSonido();
    } else {
        gg.removeCommand(cmdContinuar);
        gg.addCommand(cmdPausar);
        if(sonido_tocando!=null) {
            GestorSonidos.singleton().tocaSonido(sonido_tocando.intValue());
        }
    }
}

private class LetreroGameOver extends ObjetoMovil {
    public LetreroGameOver() {
        super(cargaImagen("/gameover.png"));
        this.setTipo(ObjetoMovil.TIPO_OTRO);
        setPosition(Pantalla.getAnchoTablero()/2-getWidth()/2,Pantalla.getAltoTablero()/2-
getHeight()/2);
    }
    public void mover() {
    }
}

private void mover() {
    addPuntos(GestorElementos.singleton().desacumulaPuntuacion());

    Protagonista prota=ge.getProtagonista();

    int estado=midlet.getEstado();
    if(estado!=DemoJuego.ESTADO_JUEGO &&
        (estado!=DemoJuego.ESTADO_GAMEOVER || !prota.isMuriendo())) {
        if(!letrero_puesto) {
            letrero_puesto=true;
            prota.setVisible(false);
            GestorElementos.singleton().añade(new LetreroGameOver());
            GestorSonidos.singleton().tocaSonido(musicaGameOver);
            tiempo=0;
        } else {
            tiempo++;
            if(tiempo>98) {
                midlet.setEstado(DemoJuego.ESTADO_PRESENTACION);
            }
        }
    } else {
        prota.mover();
    }

    Fase fase=ge.getFase();
    fase.mover();

    if(ge.reinicia(0)) {
        do {
            //comprobar las colisiones
            //es un poco lioso... se intenta minimizar al maximo el numero de comprobaciones
            ObjetoMovil obj1=ge.getObjetoMovil(0);
```

```

        int tipobj1=obj1.getTipo();
        switch(tipobj1) {
            case ObjetoMovil.TIPO_ENEMIGO:
                if(ge.reinicia(1)) {
                    do {
                        ObjetoMovil obj2=ge.getObjetoMovil(1);
                        if(obj2.getTipo()==ObjetoMovil.TIPO_DISPARO) {
                            if(obj1.collidesWith(obj2,false)) {
                                ((Enemigo) obj1).tocar((Disparo) obj2);
                                obj2.matar();
                                break;
                            }
                        }
                    } while(ge.siguiente(1));
                }

                if(obj1.collidesWith(prota,false)) {
                    matarProtagonista();
                    break;
                }
                break;
            case ObjetoMovil.TIPO_BALAMALA:
                if(obj1.collidesWith(prota,false)) {
                    obj1.matar();
                    matarProtagonista();
                    break;
                }
                break;
            case ObjetoMovil.TIPO_OPTION:
                if(obj1.collidesWith(prota,false)) {
                    ((Option) obj1).actua();
                }
        }
        obj1.mover();
    } while(ge.siguiente(0));
}

private void matarProtagonista() {
    Protagonista prota=ge.getProtagonista();
    if(protas.puedeMorir()) {
        prota.matar();
        if(--vidas<=0) {
            gg.removeCommand(cmdPausar);
            gg.removeCommand(cmdContinuar);
            midlet.setEstado(DemoJuego.ESTADO_GAMEOVER);
        }
    }
}

public int getPuntos() {
    return puntos;
}

private void addPuntos(int puntos) {
    this.puntos+=puntos;
}

public int getVidas() {
    return vidas;
}
}

```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\Pantalla.java

```

package com.ulpgc.mmacias.util;

import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.midlet.MIDlet;

```

Creación de videojuegos con J2ME

```
// La función de esta clase es "pre-cachear" algunos calculos referentes a la pantalla y algunas
utilidades más
public class Pantalla {
    public static final int TILE_SIZE=16;
    private static int anchoPantalla;
    private static int altoPantalla;

    public static void cachearDatos() {
        //Truco chapuceril para pillar el ancho y alto de la pantalla al completo
        Displayable display=new Canvas() { protected void paint(Graphics graphics) {} };
        anchoPantalla=display.getWidth();
        altoPantalla=display.getHeight();
    }

    public static int getAnchoPantalla() {
        return anchoPantalla;
    }

    public static int getAltoPantalla() {
        return altoPantalla;
    }

    public static int getAnchoTablero() {
        return anchoPantalla;
    }

    public static int getAltoTablero() {
        return 10*TILE_SIZE;
    }
}

}
```

C:\program\WTK22\apps\DemoJuego\src\com\ulpgc\mmacias\util\Utils.java

```
package com.ulpgc.mmacias.util;

import java.util.Random;

public class Utils {
    private static Random rnd=new Random(System.currentTimeMillis());
    public static int random(int max) {
        int r=rnd.nextInt();
        if(r<0) {
            r=(-r)%max;
        } else {
            r=r%max;
        }

        return r;
    }
}

}
```


Bibliografía

Recursos electrónicos

- [García 2003] Alberto García Serrano (2003). **Programación de juegos para móviles con J2ME**. PDF disponible en: <http://www.agterrano.com/publi.html> (última consulta: Noviembre 2004)
- [Mahmoud 2003] Qusay H. Mahmoud (Junio 2003). **The J2ME Mobile Media API**. HTML disponible en: <http://developers.sun.com/techtopics/mobility/midp/articles/mmapioverview/> (última consulta: Diciembre 2004)
- J2ME Game Optimization Secrets**. HTML disponible en: http://www.developer.com/java/j2me/article.php/10934_2234631_1 (última consulta: Enero 2005)
- J2ME Datasheet**. PDF disponible en: <http://java.sun.com/j2me/j2me-ds.pdf> (última consulta: Enero 2005)
- [Sun 2004] J2ME Wireless Toolkit User's Guide. HTML disponible en: <http://java.sun.com/j2me/docs/wtk2.2/docs/UserGuide-html/index.html> (última consulta: Enero 2005)

Recursos bibliograficos

- [Hartley 1998] Stephen J. Hartley (Febrero 1998). **Concurrent Programming: The Java Programming Language**. Oxford University Press
- [Varios 2003] Varios autores (Diciembre 2003). **Programación de videojuegos para teléfonos móviles en Java con J2ME**. Ediversitas Multimedia S.L.
- [Ortiz 2001] E. Ortiz, E. Guigère (Noviembre 2001) **The Mobile Information Device Profile for Java 2 Micro Edition: Professional Developer's Guide**. John Wiley & Sons, Inc