UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA - LEÓN
FACULTAD DE CIENCIAS
DEPARTAMENTO DE COMPUTACIÓN

"Desarrollo de software para móviles con Java 2 Microedition"

MONOGRAFIA PARA OPTAR AL TÍTULO DE LICENCIADO EN COMPUTACIÓN.

Presentado por:

Br. Jorge Antonio Molina Poveda.

Tutor:

Lic. Eduardo Santiago Molina Poveda.

León, Diciembre del 2005.

Este trabajo está dedicado:

Primeramente a **Dios**, Ser perfecto creador de todo cuanto existe, por haberme permitido llegar hasta este punto.

A mi padre, por demostrarme que siempre es posible lograr las cosas; solo se requiere un poco de esfuerzo.

A mi madre por apoyarme y tolerarme siempre.

A ambos: gracias por no dejarme hacer mi gusto, sino lo correcto.

A mis hermanos: Rommel, Omayra y Santiago por su apoyo.

No puedo olvidar a mis abuelos, Ramón y Elba, quienes se marcharon de este mundo poco tiempo atrás, a ellos ¡Gracias! por sus consejos y su apoyo. Los extraño.

Agradezco a todas aquellas personas que han tenido que ver con este trabajo. A mis amigos y compañeros, a aquellos que me cedieron su tiempo, computadora y quizás su conexión a Internet☺; pero sobre todo su paciencia.

Índice

Introducción

Es increíble la forma en que los dispositivos inalámbricos, particularmente los teléfonos han transformado la forma de vida de muchos individuos alrededor del mundo. Pocos años atrás tener un celular que pudiera conectarse a Internet era toda una maravilla. Hoy en día es común encontrarse con que además de eso pueden servir como videoconsolas, reproductores de audio y video, cámaras digitales, etc.

Todas estas nuevas funcionalidades han permitido el nacimiento de una industria enfocada en satisfacer las necesidades de los usuarios, proporcionando las aplicaciones y los servicios necesarios para explotar al máximo el hardware del dispositivo.

Desarrollar software y servicios para este tipo de aparatos no es tan simple. Uno de los problemas con el que se enfrenta cualquier desarrollador es la compatibilidad entre dispositivos. A diferencia de industrias como la de las computadoras de escritorio, donde el dominio está concentrado en pocos sistemas operativos y arquitecturas, la industria móvil es extremamente incompatible, debido a la gran cantidad de aparatos existentes. Aún cuando esta está dominada por pocas marcas, las incompatibilidades entre aparatos de una misma marca todavía son muy notorias. El desarrollar aplicaciones comercialmente viables significaría crear diferentes versiones de la aplicación para la mayoría de los aparatos que cumplieran con los requisitos hardware de la misma.

Lo que se necesita es una plataforma que permita desarrollar aplicaciones que se ejecuten en una gran cantidad de dispositivos y que logre además un balance entre portabilidad y rendimiento para el tipo de recursos de cada aparato. J2ME es una de esas plataformas y de hecho la de más aceptación en la actualidad.

Este documento aborda el desarrollo de software con J2ME. También se abordan conceptos sobre un campo relativamente nuevo de los servicios móviles: los servicios basados en localización; los cuales nos servirán para entender la aplicación de ejemplo desarrollada.

El documento está estructurado en tres partes:

La primera aborda conceptos sobre J2ME, su estructura y las API que soportan su funcionalidad. La segunda parte aborda conceptos sobre servicios basados en localización y el soporte que J2ME proporciona para el desarrollo de aplicaciones que los utilizan. La tercera parte describe una aplicación de ejemplo.

Objetivos

General

- Abordar el desarrollo de software para móviles utilizando la plataforma Java 2 Microedition.

Específicos

- Indagar ventajas y desventajas de desarrollar software utilizando está plataforma.
- Desarrollar un servicio simulado de consulta comercial utilizando el soporte de servicios basados en localización que J2ME proporciona.

Primera Parte

Java 2 Microedition.

1.1 Introducción a Java

Java es tanto un lenguaje de programación como una tecnología, creada por la empresa Sun Microsystems en la década de 1990. El propósito de su creación, fue el permitir una comunicación transparente entre aparatos electrodomésticos heterogéneos. Con la explosión de Internet, el objetivo cambió, y se concentró en desarrollar una plataforma que permitiera la ejecución de software independiente del sistema operativo y del hardware subyacente, pero aplicado mayormente a computadoras personales y servidores. Ahora Java no solo está presente en este tipo de aparatos, sino que ha sido utilizado para crear aplicaciones de todo tipo y para todo tipo de hardware. Se encuentra en teléfonos móviles, juguetes, robots, sistemas caseros de monitoreo, electrodomésticos, tarjetas de identificación, etc., dado que todos estos aparatos poseen características muy distintas, Sun decidió crear diferentes plataformas que cumplieran con cada uno de los requerimientos de cada tipo de dispositivo, estas son:

- Java 2 Enterprise Edition (J2EE): diseñada para aplicaciones que se ejecutan en servidores. Entre las tecnologías disponibles están: Java Server Pages, Java Server Faces, Servlets, JDBC, etc.
- Java 2 Standard Edition (J2SE): diseñada para aplicaciones que se ejecutan en computadoras personales. Entre las tecnologías disponibles están: Swing, AWT, JDBC, Java 3D, etc.
- Java 2 Microedition (J2ME): diseñada para aplicaciones que se ejecutan en aparatos con recursos hardware limitados.

En el siguiente apartado se abordará la tecnología J2ME, aplicada al desarrollo de software para aparatos móviles.

1.2 La plataforma

Java 2 Micro Edition (J2ME) fue creada por iniciativa de Motorola y Sun en el año 1999, con la intención de desarrollar un entorno que permitiera la implementación de software en dispositivos móviles.

Es un subconjunto optimizado de la tecnología Java para escritorio (J2SE), y contiene las API (Application Programming Interface: Interfaz de programación de aplicaciones) necesarias para el desarrollo de programas que se ejecutan en dispositivos de consumo, celulares y PDA (Personal Digital Assistants: Asistentes Digitales Personales).

Una de las ventajas de utilizar J2ME como plataforma de desarrollo de aplicaciones, es el poder producir código que puede ejecutarse en múltiples dispositivos; esto es mucho más importante en aparatos móviles, ya que estos son mucho más numerosos e incompatibles de lo que son entre sí las computadoras de escritorio, esta es la razón por la que en J2ME (a diferencia del resto de plataformas Java) no existe un único entorno de ejecución (véase la Figura 1); sino que estos se construyen para familias de dispositivos a partir de la combinación de **configuraciones** y **perfiles.**

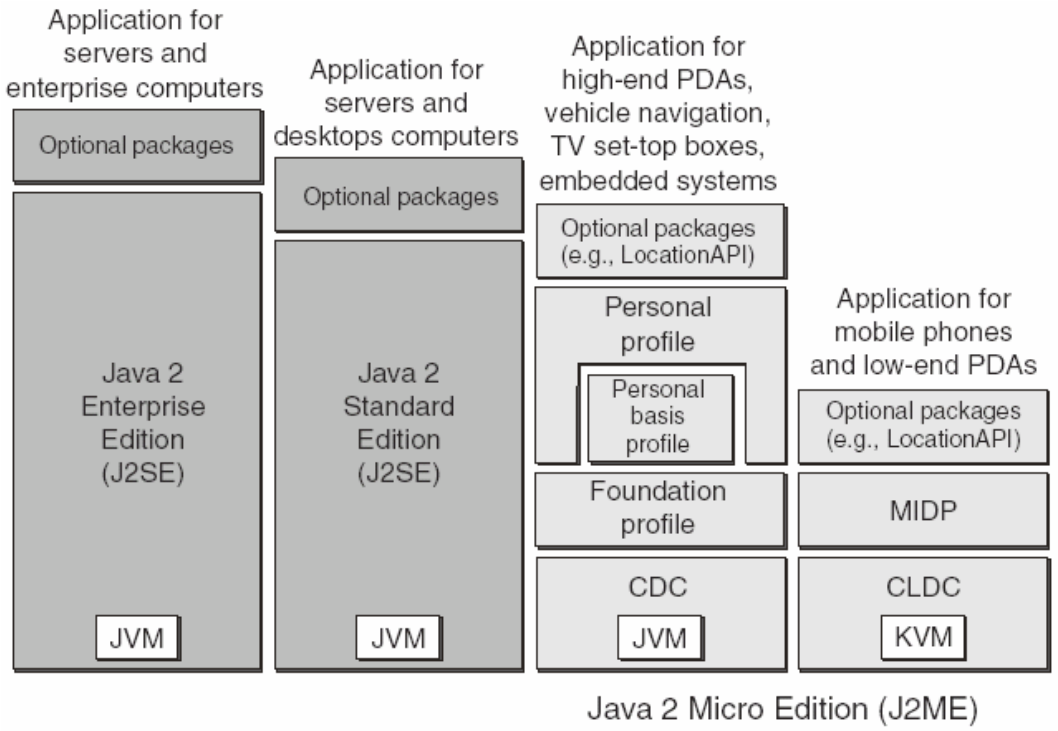


Figura 1: La plataforma Java.

1.2.1 Configuraciones

Una configuración esta compuesta de la maquina virtual y un conjunto mínimo de clases. Esta provee la funcionalidad básica para un rango particular de dispositivos que comparten características similares (cantidad de memoria, conectividad de red, etc.).

Existen dos especificaciones de configuraciones J2ME:

- **Connected Limited Device Configuration (CLDC)**: diseñada para dispositivos con procesadores lentos, poca memoria. Éstos comúnmente tienen como mínimo 128KB- 512KB disponibles para Java y las aplicaciones asociadas.

- **CDC (Connected Device Configuration):** diseñada para dispositivos que tienen mayores recursos (más memoria, procesador más rápido, etc.). CDC incluye una maquina virtual Java similar a J2SE y un subconjunto mayor de clases que las proporcionadas por CLDC. Como resultado la mayor parte de dispositivos tienen CPU de 32 bits y como mínimo 2MB de memoria disponible para Java y sus aplicaciones asociadas. Este tipo de aparatos incluye: cajas decodificadoras de señales de TV (settop boxes) y PDA de ultima generación.

Aunque una configuración proporciona el soporte básico al entorno, no proporciona la funcionalidad necesaria para la ejecución de las aplicaciones (application lifecycle: ciclo de vida de las aplicaciones). Es ahí donde entra en juego el perfil.

1.2.2 Perfiles

Un perfil extiende la funcionalidad de una configuración y define librerías más avanzadas y especificas del dispositivo, entre estas se encuentran: manejo del ciclo de vida de la aplicación, librerías de interfaz gráfica de usuario, redes, almacenamiento persistente, etc. Los perfiles son implementados en dispositivos similares. Las aplicaciones escritas para un determinado perfil pueden ser portadas a otros dispositivos que lo implementen. Los perfiles disponibles en J2ME son:

- Mobile Information Device Profile (MIDP): Ofrece la funcionalidad requerida por las aplicaciones para móviles: interfaces de usuario, conectividad de red, salvaguarda de la información y el manejo del ciclo de vida de la aplicación.
- Foundation Profile**:** Diseñada para el desarrollo de aplicaciones incrustadas (embeded) que no requieran de interfaz de usuario.
- Personal Basis Profile (PBP): diseñada para dispositivos conectados a redes, que soportan un nivel básico de presentación gráfica.
- Personal Profile (PP): diseñada para dispositivos que requieren entornos gráficos completos o soporte para applets de Internet. Incluye completo soporte para AWT y las herramientas necesarias para la ejecución de applets diseñados para entornos de escritorio.
- PDA Profile: Es un perfil utilizado con CLDC que proporciona funcionalidad más específica para aparatos de PDA de las que otros perfiles basados en CLDC pueden proporcionar.

Todas las plataformas Java pueden ser extendidas por paquetes opcionales, que no son más que clases que proporcionan funcionalidad específica para una tecnología o industria.

1.3 Connected Limited Device Configuration

Esta configuración está diseñada para dispositivos con pocos recursos hardware. Esta compuesta por una maquina virtual y todas las librerías asociadas que dan el soporte básico a todo el entorno.

Actualmente existen dos versiones de esta especificación: CLDC1.0 y CLDC1.1.

La API CLDC es básicamente una versión optimizada del paquete java.lang y java.io de J2SE. A continuación se enumeran algunas características que posee esta configuración y que están relacionadas con la interfaz de programación proporcionada así como la seguridad y estabilidad del propio entorno. A menos que se especifique lo contrario, estas características son aplicables para ambas versiones de la configuración:

- Eliminación de datos en coma flotante, debido a que muchos procesadores móviles no soportan operaciones con este tipo de datos. Con los nuevos aparatos esta limitación no existe, por lo que CLDC1.1 sí los soporta.
- Muchas de las clases eliminan métodos inútiles y /o inseguros. P. Ej: Thread no posee suspend(), resume(), setDaemon(). No existe el método Object.finalize().
- No existen cargadores de clases personalizados (ClassLoaders), toda la carga de código es efectuado por el sistema, esto evita ejecutar código maligno.
- No hay JNI (Java Native Interface), es decir no se puede acceder a métodos nativos del aparato por otras interfaces que no sean las proporcionadas por el entorno.
- No hay soporte para la reflexión de objetos, lo que elimina la seriación (almacenamiento persistente en disco) de objetos.

Desde el punto de vista de la maquina virtual el diseño también ha cambiado para CLDC:

- Utilización de objetos compactos, es decir que mucha de la información utilizada para manejar referencias a objetos en J2SE ha sido eliminada o simplificada.
- Manejo de recursos unificados: No existen áreas separadas en memoria para manejar referencias a variables, todas los datos estáticos (p. Ej.: variables declaradas dentro de métodos) y datos dinámicos (objetos creados por medio de new()), así como código de la propia maquina virtual son almacenados en la heap.
- La interpretación es optimizada.
- Recolector de basura optimizado.
- Compilador adaptativo, el cual compila a código nativo los métodos más utilizados.
- Sincronización de hilos más rápida.

Estas características proporcionadas por CLDC se conocen como características a nivel de maquina virtual.

1.4 Mobile Information Device Profile

El Mobile Information Device Profile (Perfil de Dispositivo de Información Móvil) extiende CLDC para ofrecer un entorno de ejecución Java completo para equipos electrónicos de consumo masivo (celulares, PDA y algunos electrodomésticos).

Las aplicaciones diseñadas que utilizan este perfil son conocidas como MIDlets.

Existen dos versiones de este perfil: MIDP1.0 y MIDP2.0.

1.4.1 MIDP1.0

Entre las características de esta versión están:

- Un modelo de control del ciclo de vida de las aplicaciones.
- Modelo consistente de interfaz de usuario.
- Aportar funcionalidades estándar de almacenamiento persistente.
- Soporte para la comunicación en Internet por medio de HTTP1.1.
- No proporciona seguridad adicional a la que CLDC aporta.

1.4.2 MIPD2.0

Esta versión agrega lo siguiente a su predecesora:

- Seguridad de extremo a extremo mediante el uso de estándares existentes como Secure Socket Layer (SSL) o Wireless Transport Layer Security (WTLS) que permiten la transmisión de datos cifrados.
- Multimedia incorporada.
- Mejora en la API de interfaz de usuario (nuevas clases y la posibilidad de crear componentes de interfaz de usuario propios).
- Seguridad a nivel de aplicaciones.

1.4.3 Seguridad en MIDP

Como todas las plataformas Java, J2ME limita la ejecución de las aplicaciones al contexto de una maquina virtual donde cada operación efectuada por la aplicación es supervisada, esto para evitar operaciones que dañen o degraden los recursos del sistema. Adicional a este nivel, MIDP proporciona otro en el que se restringen los servicios que una determinada aplicación puede obtener del dispositivo aún dentro del contexto de la propia maquina virtual.

La especificación MIDP1.0 limita la ejecución de cada midlet en un contexto donde el acceso a funcionalidad sensible del aparato es controlado; este concepto es conocido como “caja de arena” o “sandbox”. En este caso todos los MIDlets que deseen acceder a funcionalidad limitada, deben obtener permiso explicito del usuario.

MIDP 2.0 introduce el concepto de aplicaciones de confianza (trusted applications); las cuales pueden utilizar automáticamente API consideradas sensibles y que por lo tanto están restringidas. Una vez que se ha comprobado que el midlet es de confianza se puede garantizar el acceso a estas API sin necesidad de notificar al usuario.

Midlets no confiables

Un midlet no confiable (untrusted midlet) es una aplicación en la que no se puede determinar su origen y la integridad del archivo jar contenedor no puede ser

comprobada por el dispositivo. Los midlets no confiables “deben” ejecutarse (siempre que no ocurra un error en su verificación) en un contexto restringido donde el acceso a API sensibles no es permitido o es autorizado únicamente con permiso expreso del usuario.

Entre las API que pueden ser utilizadas por midlets no confiables sin necesidad de intervención del usuario están:

- API de almacenamiento persistente.
- La API de manejo del ciclo de vida de los midlets.
- API de interfaz de usuario.
- API de juegos.
- API multimedia.

Funcionalidad considerada sensible, como aquella proporcionada por la API de acceso a protocolos de red, sólo puede ser utilizada por MIDlets si el usuario lo autoriza.

Seguridad en midlets confiables

Un midlet confiable es aquel en que la autenticidad e integridad del archivo de aplicación ha sido verificada.

La seguridad para este tipo de midlets está basada en dominios de protección. Cada dominio define los permisos que le pueden ser garantizados a ese midlet en el dominio. Un permiso es un identificador definido por una API para prevenir su uso sin autorización.

El creador del dominio especifica la forma en que el dispositivo identifica y verifica que un midlet es confiable.

Modelo de Autorización

La autorización básica para un midlet es establecido por la relación entre los siguientes elementos:

- Un dominio de protección que consiste en un conjunto de permisos “Garantizado” y “Usuario” (Granted y User).
- Un conjunto de permisos solicitados por el midlet en los atributos MIDlet-Permissions y MIDlet-Permissions-Opt de su archivo descriptor de aplicación.
- Un conjunto de permisos para cada API protegida o función en el dispositivo, el cual es una unión de todos los permisos definidos por cada API en el dispositivo para funciones protegidas.
- El usuario, al que puede pedírsele autorización.

Permisos

Un permiso es el medio por el cual se establece un control de una API o función que requiere autorización explicita antes de ser invocada.

El nombre de los permisos tiene una organización similar a los paquetes Java, diferencian entre mayúsculas y minúsculas. Todos los permisos para una API utilizan como prefijo el nombre del paquete donde se encuentra la API, si es para una función dentro de una clase, el permiso contendrá además el nombre de la clase.

Un midlet que requiera acceso para API protegidas debe pedir los correspondientes permisos, esto se hace a través del atributo MIDlet-Permissions. Los permisos solicitados en este atributo son críticos, si el acceso no es concedido el midlet no podrá ejecutarse.

Un midlet que requiera permisos, pero que aún sin estos pueda funcionar correctamente debe solicitarlos por el atributo MIDlet-Permissions-Opt.

Los atributos MIDlet-Permissions y MIDlet-Permissions-Opt contienen una lista de nombres de permisos separados por comas.

Todos estos atributos están almacenados en el Java application descriptor de cada aplicación.

Dominios de protección

Un dominio de protección define los permisos y sus modos de interacción. Está compuesto por:

- Los permisos que deberían ser autorizados (“Allowed”).
- Los permisos que deberían ser autorizados por el usuario (”User”) y sus modos de interacción correspondientes.

Un modo de interacción de usuario se define para permitir al usuario negar o admitir el acceso a un API, puede tener los siguientes valores:

- **blanket:** se pregunta sólo la primera vez que se invoca a la API y permanece válido para cada invocación hasta que el midlet sea desinstalado o el permiso sea cambiado por el usuario.
- **session:** válida para cada invocación del midlet, en este modo el usuario es consultado la primera vez que se accede al API protegida por cada ejecución del midlet.
- **oneshot**: pregunta al usuario cada vez que se intenta acceder al API protegida.

La autorización para midlets confiables está basada en la información proporcionada por: el dominio de protección, permisos en el dispositivo y los permisos solicitados por el midlet. Los permisos en el dominio de protección son “Allowed” (Garantizados) o “User” (Usuario). Los permisos solicitados por la aplicación son críticos o no- críticos.

Para establecer los permisos que se autorizan a un midlet al invocarse, las siguientes condiciones deben cumplirse:

- El midlet ha sido puesto a un dominio de protección.
- Los permisos solicitados por la aplicación se obtienen de los atributos MIDlet-Permissions y MIDlet-Permissions-Opt. Si aparecen en el descriptor de la aplicación deben coincidir con los atributos correspondientes en el manifiesto; si no coinciden el midlet no debe ser invocado.
- Si alguno de los permisos no críticos solicitados es desconocido por el dispositivo, se elimina de los permisos solicitados.
- Si alguno de los permisos críticos solicitados es desconocido el midlet no es invocado.
- Si alguno de los permisos críticos no se encuentran en el dominio de protección el midlet no es invocado.

- Durante la ejecución se debe revisar el permiso de uso para cada API protegida, si no se logra comprobar; se lanza una excepción del tipo SecurityException.

La siguiente tabla proporciona un ejemplo de como está estructurada una política de dominio:

Sintaxis	Significado
domain: O=”Moonsoft Corporation”, C=NI	O (Organization), C (Country)
allow: javax.microedition.io.HttpConnection	Acceso autorizado
oneshot: javax.microedition.io.SocketConnection	Se pregunta al usuario en cada acceso.

A continuación describiremos la API MIDP.

1.4.4 API MIDP

Este apartado contiene una breve descripción de algunas de las clases proporcionadas por la API MIDP. Excepto donde se indique lo contrario, la descripción proporcionada es aplicable en ambas versiones de la especificación.

1.4.4.1 MIDlets

Todas las aplicaciones MIDP están representadas y contienen al menos una clase derivada de la clase MIDlet, la cual esta contenida en el paquete javax.microedition.midlet. Esta clase define el ciclo de vida de todas las aplicaciones y los servicios que estas pueden obtener de su entorno.

La clase MIDlet

Como se ha mencionado, esta clase define todo el ciclo de vida de las aplicaciones MIDP. Durante su ejecución, toda aplicación MIDP transita en diferentes estados. Las transiciones hacia y desde estos estados son efectuados por un modulo del entorno conocido como Application Management Software (software de manejo de las aplicaciones) que realiza llamadas a métodos de esta clase.

Un midlet puede estar en tres estados diferentes: pausado, activo y destruido.

Un midlet pasa al estado:

- **Pausado**
    - Inmediatamente después que ha sido creado utilizando su constructor sin parámetros. Si ocurre una excepción, el midlet pasa inmediatamente al estado destruido.
    - Desde el estado activo, inmediatamente después que el AMS ha llamado al método MIDlet.pauseApp() y este retorna sin provocar excepciones.
    - La aplicación se encuentra activa y se ha llamado a MIDlet.notifyPaused().
    - Desde el estado activo y el método MIDlet.startApp() provoca una MIDletStateChangeException.

- **Activo**

  - Antes de que el AMS llame al método MIDlet.startApp().

- **Destruido:** El midlet ha liberado todos sus recursos. La transición a este estado se da una sola vez cuando:

  - El AMS ha llamado al método MIDlet.destroyApp() y retorna sin problemas excepto cuando el argumento de este método es establecido a false y una MIDletStateChangeException ha sido lanzada.
  - Cuando la llamada al método MIDlet.notifyDestroyed() hecha por la aplicación ha retornado. La aplicación es responsable por llamar a MIDlet.destroyApp() antes de llamar a MIDlet.notifyDestroyed().

La siguiente figura clarifica mejor lo dicho:

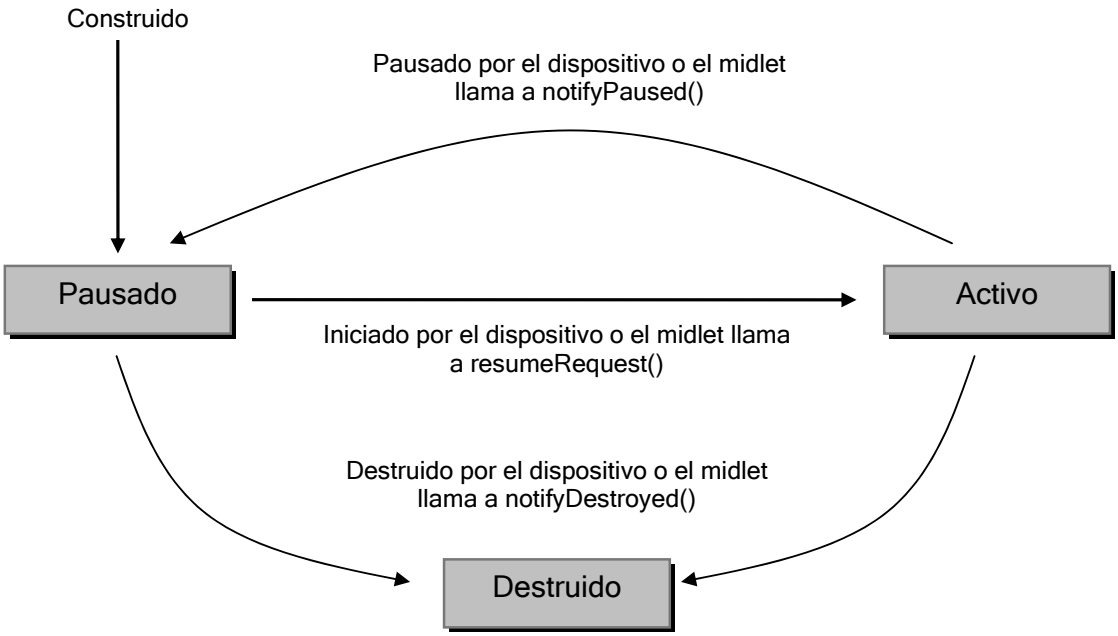


Figura 2: Ciclo de vida de las aplicaciones MIDP

Los métodos de la clase que permiten las transiciones a los estados mencionados son:

protected abstract void pauseApp()

Este método es llamado cuando el midlet debe pausar su ejecución. Ocurre inmediatamente después que se carga la aplicación. También es invocado cuando ocurre algún evento externo (por ejemplo llega una llamada telefónica). En este caso el midlet debe liberar recursos temporales y suspender su actividad.

protected abstract void startApp() throws MIDletStateChangeException

Este método es llamado cuando el midlet debe reanudar su ejecución. Si el midlet no puede iniciar en este momento por alguna razón (no fatal) pero puede iniciar posteriormente, el método debe lanzar una MIDletStateChangeException, si no puede iniciar en ningún momento posterior, este método debería llamar a notifyDestroyed().Si startApp() lanza otro tipo de excepción (de forma deliberada o porque un método al que invoca la provoca) el midlet es destruido.

protected abstract void destroyApp(boolean condicional) throws MIDletStateChangeException

Este método es llamado para indicar al midlet que debe liberar cualquier recurso obtenido. Si el midlet no desea entrar en el estado destruido debe lanzar un MIDletStateChangeException, únicamente si el parámetro condicional es establecido a false. Si el parámetro condicional es true, el midlet entra al estado destruido sin importar como termine el método.

public final void notifyPaused()
public final void notifyDestroyed()
public final void resumeRequest()

Estos métodos son utilizados por el midlet para indicar que se encuentra en el estado pausado, destruido o que desea entrar en el estado activo, respectivamente.

Servicios proporcionados por el entorno

También se mencionó que un MIDlet puede comunicarse con su entorno, los siguientes métodos son utilizados para este propósito:

public final boolean platformRequest(String URL) throws ConnectionNotFoundException

Este método permite utilizar servicios del dispositivo, por ejemplo mostrar la URL especificada.

Si la plataforma tiene capacidades para satisfacer la petición, este método activa la aplicación correspondiente para que el usuario pueda interactuar con ella. Sin embargo el midlet permanece ejecutándose.

Este método no maneja múltiples peticiones, lo que significa que si la aplicación efectúa múltiples llamadas, la plataforma sólo cumplirá la última especificada.

Si el parámetro URL es de la forma: tel:<número>, la plataforma interpreta esto como una petición de inicio de llamada telefónica.

Este método retorna true si la aplicación debe finalizar antes de cumplir con la petición. Si la plataforma no puede manejar la petición se provoca un ConnectionNotFoundException.

public final int checkPermission(String permiso)

Este método permite obtener el estado de un permiso. El parámetro identifica el nombre de la API por la que se pregunta. Si el estado del permiso es negado, es decir, que la API definida por el permiso no se puede utilizar, el método retorna 0. Si la API definida por el parámetro se puede utilizar el método retorna 1. El método retorna –1 si el estado del permiso es desconocido, por ejemplo por que requiere interacción del

usuario. Para obtener información sobre lo que es un permiso y la seguridad a nivel de midlets, véase la sección sobre seguridad en MIDP.

public final String getAppProperty(String propiedad)

Proporciona un mecanismo que permite obtener propiedades con nombre proporcionadas por el AMS. Los valores obtenidos son tomados de una combinación del archivo de descripción de aplicaciones y el archivo manifiesto que acompañan a la aplicación. El parámetro propiedad especifica el nombre de la propiedad. Se retorna null si la propiedad especificada no existe. Se provoca NullPointerException si el parámetro referencia null.

Puede obtener información sobre todas las propiedades disponibles consultando la documentación oficial.

A continuación se abordará uno de los aspectos más importantes de las aplicaciones modernas: las interfaces de usuario, y como MIDP da soporte a estas.

1.4.4.2 Interfaces de usuario en MIDP

Las aplicaciones MIDP se diseñan para ser portátiles entre dispositivos que poseen características de entrada y salida diferentes, que van desde aparatos con pantallas monocromáticas pequeñas, con teclados simples hasta aquellos con pantallas a color, de mayores dimensiones y dispositivos de entrada sofisticados (teclados más completos y dispositivos apuntadores). Crear interfaces de usuario para este tipo de equipos no es una tarea sencilla, una solución inicial sería utilizar Abstract Window Toolkit (AWT) o Swing de J2SE para proporcionar los elementos de creación de esas interfaces graficas; no obstante ninguno representa una respuesta viable debido a su complejidad y a los recursos que requerirían (memoria, dimensiones de pantalla, etc.). Es por esto, que los diseñadores de MIDP introducen un conjunto de elementos de alto nivel mucho más simple y un modelo de programación basado en pantallas. Con este modelo, el programador se enfoca en la lógica de la aplicación más que en los detalles de la interfaz de usuario en sí. El resultado es una API mucho más pequeña, sencilla de usar y que requiere muchos menos recursos que AWT o Swing.

Este apartado está dedicado a describir el modelo de interfaces de usuario utilizado por MIDP y parte de las clases que lo componen.

El modelo de interfaz de usuario

El modelo de interfaces de usuario utilizado en MIDP está basado en pantallas. Las aplicaciones GUI J2SE a menudo están compuestas por ventanas que se muestran simultáneamente y con las que el usuario puede interactuar con un dispositivo apuntador. Un dispositivo MIDP en contraste, solo necesita mostrar una “ventana” a la vez y la habilidad para moverse entre “ventanas” depende de que el programador haya incluido los componentes de interfaz necesarios para hacerlo. Por lo tanto, si hay más de un midlet ejecutándose en un momento determinado, sólo una aplicación tendrá acceso a la pantalla del aparato, y el dispositivo puede o no proporcionar el mecanismo para dejar que el usuario seleccione que midlet tiene acceso a la pantalla en esa situación.

La librería de interfaz de usuario de MIPD, que está implementada en el paquete javax.microedition.lcdui incluye clases que representan la pantalla del dispositivo y proporciona los componentes necesarios para crear aplicaciones basadas en el modelo antes mencionado.

Las clases Display y Displayable

La clase Display representa una pantalla lógica del dispositivo en la que el midlet puede dibujar su interfaz de usuario. Cada midlet tiene acceso a una única instancia de esta clase; para obtener una referencia a esa instancia se utiliza el método estático getDisplay():

```
public static Display getDisplay(MIDlet midlet)
```

Usualmente este método es invocado en el método startApp() de la clase MIDlet, que luego utiliza el objeto retornado para mostrar su primera interfaz de usuario. getDisplay() siempre retorna el mismo objeto.

Cada pantalla que un midlet necesita mostrar, está compuesta de elementos conocidos como items y que son utilizados en objetos de clases derivadas de la clase Displayable (que se detallará más adelante). Un objeto Displayable no es visible hasta ser asociado al objeto Display del midlet usando el método setCurrent():

```
public void setCurrent(Displayable d)

public void setCurrent(Alert alerta, Displayable proximaPantalla)
```

La primera sobrecarga de este método solicita que el objeto Displayable sea mostrado, la segunda versión del método es utilizada para trabajar con la clase Alert (detalles sobre esta más adelante) y permite mostrar el objeto Alert; una vez que este objeto desaparece el objeto proximaPantalla es mostrado. El segundo método lanza NullPointeException si cualquiera de los dos argumentos es null e IllegalArgumentException si proximaPantalla es otro objeto Alert.

De forma similar el objeto actualmente asociado a Display puede ser obtenido llamando a getCurrent():

```
public Displayable getCurrent()
```

Display contiene otros métodos útiles para obtener información y utilizar capacidades del dispositivo, por ejemplo: capacidades de la pantalla del dispositivo, vibración, etc..

La clase Displayable representa un objeto que puede ser mostrado en la pantalla. Tal objeto puede contener un título, un ticker (el equivalente a un marquesina en HTML), cero o más comandos y tener asociado un “escuchador” de eventos para esos comandos. El contenido mostrado y la interacción con el usuario están definidos por las subclases de esta.

Esta es la declaración completa de Displayable:

```
public abstract class Displayable extends Object
{
        public boolean isShown();
        public int getHeight()
        public int getWidth();
        public void setTitle(String titulo);
        public String getTitle();
        protected void sizeChanged(int ancho, int alto);
```

```
        public void setTicker(Ticker ticker);
        public Ticker getTicker();
        public void addCommand(Command c);
        public void removeCommand(Command c);
        public void addCommandListener(CommandListener l);
}
```

El método isShown() retorna true sólo cuando el objeto Displayable puede ser visto por el usuario, es decir cuando es el objeto actual del Display asociado al midlet. Los métodos getHeight() y getWidth() retornan la altura y anchura respectivamente del área visualmente disponible (sin incluir el área ocupada por el título, ticker o comandos) para el objeto Displayable. Los métodos setTitle() y getTitle() permiten establecer y obtener el título del objeto. setTicker() y getTicker() permiten establecer y obtener el texto que se desplaza en el objeto Display; este texto está representado por un objeto de la clase Ticker.

Los otros tres métodos de la clase están relacionados con el uso de comandos, los que se abordarán a continuación.

Comandos

La clase Displayable puede contener comandos. Un comando es un elemento de la interfaz gráfica que desencadena una acción al ser activado, sería el equivalente (visualmente hablando) al item de un menú o a un botón en una aplicación Swing o AWT. En MIDP los comandos están representados por la clase Command, la cual está declarada así:

```
public class Command extends Object {…}
```

Un objeto de está clase contiene información semántica sobre la acción desencadenada, no la acción en sí. Está información es utilizada para representar el comando en la interfaz de usuario. Tal representación es dependiente de la implementación.

La información que un comando contiene está compuesta de cuatro partes: una etiqueta, una etiqueta larga opcional, un tipo y una prioridad.

La etiqueta es una cadena de texto (usualmente una palabra) utilizada para describir al usuario el propósito del comando. La etiqueta larga (unas pocas palabras) proporciona una descripción mayor, la representación de una de estas etiquetas como un elemento visual es una decisión de la implementación basada en el contexto y el espacio visual disponible.

Un comando tiene asociado un tipo, el cual es un valor entero que determina el propósito de este e influye también en su representación visual. Los tipos de comandos definidos son:

- BACK: un comando que retorna al usuario a una pantalla lógica previa.
- CANCEL: un comando que representa una respuesta negativa a un dialogo.
- EXIT: un comando utilizado para salir de la aplicación.
- HELP: un comando que representa la solicitud de ayuda.
- ITEM: un comando asociado a un Item de un objeto Screen.
- OK: un comando que representa una respuesta positiva a un dialogo.
- SCREEN: un comando que pertenece a la pantalla actual.

- STOP: un comando que detiene una operación en ejecución.

Todos los comandos no SCREEN, son sujetos al reemplazo de sus etiquetas por la implementación.

La prioridad es utilizada para indicar la importancia de un comando respecto a otros comandos en la misma pantalla. La prioridad es un valor entero, valores menores indican mayor importancia.

Típicamente la implementación selecciona el lugar de posicionamiento de un comando basado en su tipo y luego agrupa comandos con tipos similares basado en sus prioridades. Esto significa que comandos con prioridades mayores son posicionados en lugares donde el usuario puede activarlos directamente, y aquellos con prioridades menores son puestos en un menú. No es un error que comandos con las mismas prioridades y tipos existan en una misma pantalla, en este caso queda a discreción de la implementación el orden en que estos son presentados.

Para crear un comando disponemos de dos constructores:

public Command(String etiqueta, int tipo, int prioridad)

public Command(String etiqueta, String etiquetaLarga, int tipo, int prioridad)

Ambos métodos lanzan NullPointerException si etiqueta vale null o si el tipo de comando es inválido.

El siguiente fragmento de código crearía dos comandos de tipo ITEM:

```
import javax.microedition.lcdui.*;
…
        Command c1,c2;
        c1=new Command(''Comprar'',Command.ITEM,1);
        c2=new Command("Reproducir","Reproducir sonido", Command.ITEM,1);
…
```

La clase Command contiene otros métodos que permiten obtener información sobre un comando:

public int getCommandType();

public String getLabel();

public String getLongLabel();

public int getPriority();

Estos métodos retornan el tipo de comando, la etiqueta corta, la etiqueta larga o null si el comando no tiene etiqueta larga y la prioridad respectivamente.

Como ya se había mencionado, la clase Displayable contiene métodos para trabajar con comandos:

public void Displayable.addCommand(Command c)

Este método permite añadir un comando referenciado por la variable c, provoca un NullPointerException si c hace referencia a null. Si c hace referencia a un comando ya añadido, el método no hace nada.

Para quitar un comando utilizamos el método removeCommand() de Displayable:

public void Displayable.removeCommand(Command c)

Si el comando no se encuentra en el objeto Displayable o c vale null, este método no hace nada.

Para indicar que deseamos “escuchar” eventos generados por un comando debemos implementar la interfaz CommandListener y “activar” la clase que implementa esa interfaz con el método setCommandListener():

public void setCommandListener(CommandListener l)

Para indicar que ya no queremos escuchar eventos llamamos a este método con null como parámetro.

La interfaz CommandListener contiene un solo método:

public void commandAction(Command c, Displayable d)

Donde c es el comando que genero el evento y d representa la pantalla en la que el evento fue generado.

El siguiente ejemplo muestra como podrían utilizarse estos métodos para activar y procesar eventos generados por comandos:

```
import javax.microedition.lcdui.*;

public class MiDisplayable extends Displayable implements CommandListener
{
        Command c1,c2;
        public MiDisplayable(){
                c1=new Command(“Reproducir”,Command.ITEM,1)
                c2=new Command(“Salir”,Command.EXIT,1);
                addCommand(c1);
                addCommand(c2);
                setCommandListener(this);
        }
        public void commandAction(Command c, Displayable d){
                if(d==this){
                }
                else if(c==c1){
                        //ir a http://www.moonsoft.com
                }
                else if(c==c2){/*salirse de la aplicación*/}
        }
}
```

Los comandos aplicados a Displayable no son muy útiles porque esta clase al ser abstracta no proporciona la suficiente funcionalidad para ser utilizada en interfaces de usuario. Sin embargo existe un conjunto de clases derivadas de esta, que sí sirven como base para construir las interfaces de usuario necesarias para las aplicaciones. Esta jerarquía se muestra en la siguiente figura:

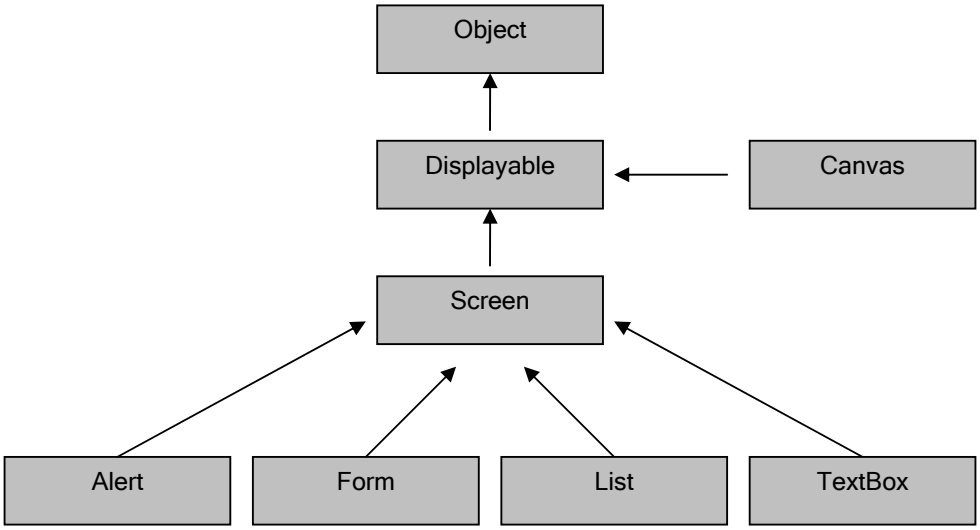


Figura 3: Jerarquía de clases Displayable

Como puede observarse hay dos clases derivadas directamente de Displayable, ambas abstractas. Estas dos clases constituyen las dos formas de programar interfaces gráficas en MIDP, la primera representada por la clase Screen se conoce como la API de interfaces de alto nivel donde el programador no tiene mucho control sobre la apariencia o "look and feel" de la interfaz, a cambio no obstante recibe sencillez de uso. La segunda forma representada por la clase Canvas es conocida como la API de bajo nivel y se utiliza principalmente en la programación de aplicaciones multimedia, en este caso el programador tiene control total sobre la apariencia de la interfaz (por supuesto debe escribir el código para lograr esa apariencia). En este caso solamente nos ocuparemos de Screen y sus clases derivadas.

La clase Screen

La clase Screen es la clase base para todas las "ventanas" de la interfaz de alto nivel. No tiene métodos propios. La subclase más comúnmente utilizada es Form, que permite desarrollar un interfaz gráfica completa añadiéndole componentes estándar. Las clases List, TextBox y Alert (el equivalente a un cuadros de mensajes) también son subclases de Screen.

La clase TextBox

La clase TextBox es una pantalla que permite al usuario editar texto. Contiene un tamaño máximo o capacidad que indica el número de caracteres que pueden ser almacenados en un objeto de este tipo. Esta capacidad máxima puede ser limitada por la implementación y no necesariamente coincide con la capacidad solicitada por la aplicación.

El texto que un objeto TextBox contiene puede ser más grande que el que se puede mostrar en un momento dado, en ese caso la clase proporciona los servicios de desplazamiento o scrolling de forma automática.

Un objeto de esta clase conoce siempre el tipo de texto que contiene o puede aceptar, esto mediante restricciones de entrada o "input constraints". Estas restricciones son compartidas con la clase TextField (de la que hablaremos más adelante):

Restricción	Significado
TextField.ANY	Se permite introducir cualquier texto.
TextField.DECIMAL (MIDP2.0)	El texto a introducir representa un número decimal.
TextField.EMAILADDR	El texto a introducir es una dirección de correo electrónico.
TextField.NUMERIC	El texto a introducir representa un valor entero.
TextField.PHONENUMBER	El texto a introducir representa un número telefónico, el texto puede ser puesto en un formato especial de cara a su visualización.
TextField.URL	El texto a introducir hace referencia a una Universal Resource Locator.

Además, existen modificadores que pueden ser combinados con los limitantes antes mencionados, estos son:

Modificador	Significado
TextField.PASSWORD	El texto introducido es sensible y debe ser enmascarado.
TextField.UNEDITABLE	El texto no puede ser modificado por el usuario.
TextField.SENSITIVE	El texto es sensible y no debe ser almacenado en diccionarios para luego ser utilizado en esquemas de entrada acelerada de texto.
TextField.NON_PREDICTIVE	Indica que en la entrada de texto, no deben usarse esquemas de predicción de texto.
TextField.INITIAL_CAPS_WORD	Indica que la primera letra de cada palabra debe ser puesta en mayúscula.
TextField.INITIAL_CAPS_SENTENCE	Indica que la letra de cada oración debe ser puesta en mayúscula.

Un valor especial que no corresponde a ninguna de las categorías anteriores es TextField.CONSTRAINT_MASK, es utilizado para comprobar qué limitantes o modificadores están activos.

Para crear una instancia TextBox se dispone del siguiente método:

```
public TextBox(String titulo, String texto, int tamañoMaximo, int restricciones)
```

Este constructor inicializa la instancia TextBox con un título, su contenido referenciado por texto, un tamaño máximo y restricciones en la edición. El tamaño máximo solicitado puede no ser concedido, en ese caso el texto es truncado.

Se provoca un IllegalArgumentException si una de las siguientes condiciones ocurre:

- El parámetro tamañoMaximo es menor o igual a cero.
- Las restricciones no son validas.
- Si el texto especificado en texto es inválido de acuerdo a las restricciones establecidas.
- Si la longitud del texto excede el tamaño máximo solicitado.

La siguiente línea construye un objeto TextBox sin título, sin contenido, con 30 caracteres como máximo y en el que se puede introducir cualquier cadena:

```
TextBox tb=new TextBox(null,null,30,TextField.ANY);
```

Para comprobar cuál es el tamaño máximo concedido a la instancia utilizamos el método getMaxSize():

public int getMaxSize()

Para establecer un nuevo máximo en el número de caracteres utilizamos el método setMaxSize():

public int setMaxSize(int nuevoMaximo)

Este método retorna el número de caracteres máximo concedido, el cual puede ser menor que el solicitado. Provoca un IllegalArgumentException si el parámetro nuevoMaximo es menor o igual a cero o si al truncar el texto que contiene el objeto resulta no válido para las restricciones establecidas.

Para cambiar el texto del TextBox se pueden utilizar cualquiera de los siguientes métodos:

public void setString(String str)

public void setChars(char[] datos, int desde, int longitud)

public void insert(String str, int posicion)

public void insert(char[] datos, int desde, int longitud, int posicion)

El método setString() establece el texto a str; lanza un IllegalArgumentException si la cadena no cumple con las restricciones establecidas o si la longitud del nuevo texto excede la capacidad máxima.

El método setChars() establece el texto a los caracteres comprendidos en el rango [desde ,desde + longitud]. Provoca un ArrayIndexOutOfBoundsException si el rango es inválido e IllegalArgumentException si los caracteres son inválidos para las limitantes o si la longitud del texto excede el límite de capacidad máxima.

La primera versión de insert() añade el texto str desde la posición antes del parámetro posicion en el contenido del objeto TextBox. Si posicion es menor o igual a cero la inserción ocurre al principio, si es mayor o igual a la longitud del texto ocurre al final. Este método provoca IllegalArgumentException si el texto resultante es inválido para las limitantes establecidas, esta excepción también es provocada si el resultado excede la capacidad máxima. NullPointerException se provoca si el str referencia null.

La segunda versión de insert() añade los caracteres en el rango [desde, desde + longitud] del arreglo datos, partiendo la posición posición-1. Se provoca ArrayIndexOutOfBoundsException si el rango no es válido, IllegalArgumentException si el texto resultante es inválido para las limitantes establecidas, esta excepción también es provocada si el resultado excede la capacidad máxima. NullPointerException se provoca si el arreglo datos referencia null.
Para obtener el texto contenido en la instancia TextBox se utiliza getString() y getChars():

public String getString()

public int getChars(char[] datos)

El método getString() retorna un objeto String que contiene el texto del TextBox o null si no hay ninguno. El método getChars() copia en el arreglo de caracteres datos el texto del objeto; si datos hace referencia a null se lanza NullPointerException, si el arreglo no es lo suficientemente grande para almacenar el texto se provoca ArrayIndexOutOfBoundsException.

El método setConstraints() se utiliza para establecer las restricciones del contenido:

public void setConstraints(int restricciones)

Este método provoca un IllegalArgumentException si restricciones es no válido. Si el contenido del objeto no coincide con las restricciones establecidas, es borrado.

Para obtener las restricciones que una instancia posee, se dispone de getConstraints():

public int getConstraints()

Para saber la longitud del texto almacenado, se dispone de size():

public int size()

La clase List

Esta clase representa una pantalla que contiene un grupo de opciones seleccionables. Todos los métodos de esta clase están declarados en la interfaz javax.microedition.lcdui.Choice.

La declaración de esta clase es la siguiente:

public class List extends Screen implements Choice{...}

Esta clase tiene mucha similitud con la clase ChoiceGroup ya que esta también implementa la interfaz Choice.

Un elemento de la lista está compuesto de una etiqueta, un objeto Image y un atributo fuente.

Tipos de listas:

Un objeto List tiene asociado un tipo (un valor entero) que indica la forma de selección que puede efectuarse sobre la instancia.

Están definidos los siguientes tipos:

- Choice.EXCLUSIVE: la lista tiene exactamente un solo elemento seleccionado a la vez.
- Choice.IMPLICIT: un solo elemento seleccionado a la vez, aquel que tiene el foco cuando un comando de la lista es invocado.
- Choice.MULTIPLE: la lista puede contener múltiples elementos seleccionados a la vez.
- Choice.POPUP: un solo elemento seleccionado a la vez. Ese elemento es siempre visible, el resto de elementos puede estar oculto hasta que el usuario realice una acción para mostrarlos. Este tipo no puede ser utilizado con un objeto List.

Para crear un objeto List se dispone de los siguientes constructores:

public List(String titulo, int tipoLista)

Este método crea una lista sin contenido, con el título y del tipo especificado. Se provoca un IllegalArgumentException si tipoLista no es IMPLICIT, EXCLUSIVE o MULTIPLE.

public List(String titulo, int tipoLista, String[] elementos, Image[] iconos)

Este método crea una instancia List con el título, del tipo, con los elementos y sus iconos especificados. Se provoca un NullPointerException si el arreglo elementos o alguno de sus referencias vale null; IllegalArgumentException si el arreglo iconos no es null y tiene una longitud diferente a el arreglo elementos, o si el tipo de lista no es IMPLICIT, EXCLUSIVE o MULTIPLE.

El siguiente fragmento construye un objeto List de tipo EXCLUSIVE:

```
...
String[] elementos={“Leer mensaje”,”Borrar mensaje”,”Reenviar”};
List listaAcciones=new List(“Acciones”,Choice.EXCLUSIVE,elementos,null);
Display.getDisplay(this).setCurrent(listaAcciones);
...
```

Para añadir un elemento a una instancia List se dispone de los siguientes métodos :

public int append(String etiqueta, Image icono)

Este método añade al final de la lista un nuevo elemento con la etiqueta e icono especificados. Retorna el índice asignado al nuevo elemento. Provoca NullPointerException si la etiqueta referencia null.

public void insert(int indice, String etiqueta, Image icono)

Este método añade un nuevo elemento antes del elemento especificado. El valor del indice está entre 0 y size()-1 inclusive (size() retorna el numero de elementos de la lista). Se provoca IndexOutOfBoundsException si el valor de indice es inválido, NullPointerException si la variable etiqueta referencia null.

Para reemplazar un elemento se dispone del método set():

public void set(int indice, String etiqueta, Image icono)

Este método reemplaza el contenido del elemento en la posición índice, el valor de indice está entre 0 y size()-1 inclusive. Provoca un IndexOutOfBoundsException si el valor indice es inválido y NullPointerException si etiqueta referencia null.

Para borrar elementos de la lista se dispone de los siguientes métodos:

public void delete(int indice)

Este método borra el elemento de la lista que esta en el índice especificado. El valor del indice está entre 0 y size()-1, de lo contrario se provoca IndexOutOfBoundsException.

public void deleteAll()

Este método borra todos los elementos de la lista.

Para averiguar si un elemento está seleccionado utilizamos el método isSelected():

public boolean isSelected(int indice)

Devuelve true si elemento en la posición especificada está seleccionado. El valor indice está entre 0 y size()-1 inclusive, de lo contrario se provoca IndexOutOfBoundsException.

Para obtener el indice del elemento seleccionado se utiliza el método getSelectedIndex():

public int getSelectedIndex();

Si la lista es de tipo MULTIPLE se retorna -1. Para este caso se utiliza el método getSelectedFlags():

public int getSelectedFlags(boolean[] array_de_retorno)

Devuelve el número de elementos seleccionados y copia el estado de los elementos en el array_de_retorno. Este método es valido para cualquier tipo de lista. Provoca IllegalArgumentException si el arreglo tiene menos elementos que la lista y NullPointerException si el arreglo referencia null.

Para seleccionar un elemento se utiliza setSelectedIndex():

public void setSelectedIndex(int indice, boolean seleccionado)

Para listas de tipo MÚLTIPLE, simplemente pone el elemento indicado al estado especificado.

Para Choices tipo EXCLUSIVE y POPUP e IMPLICIT, el método tiene efecto sólo si el parámetro seleccionado vale true. El elemento previamente seleccionado es deseleccionado. Si indice hace referencia a un indice ya seleccionado el método no

hace nada. Para cualquier tipo de lista se provoca un IndexOutOfBoundsException si valor de indice no está en el rango 0 a size()-1 inclusive.

Para seleccionar elementos en una lista se dispone de setSelectedFlags():

public void setSelectedFlags(boolean[] estados)

Para Choices tipo MÚLTIPLE, este método selecciona todos los elementos al estado indicado en el arreglo.

Para Choices tipo EXCLUSIVE, POPUP e IMPLICIT, únicamente un elemento del arreglo debe valer true, si ningún elemento vale true, se establece el primer elemento a true. Si dos o más elementos valen true se selecciona el primero de estos elementos. En cualquier caso se provoca un IllegalArgumentException si el arreglo no tiene la misma cantidad de elementos de la lista y NullPointerException si el arreglo referencia null.

Las clases Alert y AlertType:

La clase Alert es una clase derivada de Screen, permite mostrar información al usuario por un cierto periodo de tiempo. Puede contener solo una cadena de texto, una imagen y un medidor o Gauge. Es el equivalente a las cajas de mensaje en las aplicaciones AWT o Swing.

Las alertas son utilizadas principalmente para mostrar información sobre errores o situaciones especiales.

Esta clase está declarada así:

public class Alert extends Screen{...}

Un objeto Alert puede tener asociada un objeto de la clase AlertType, que proporciona un indicador del tipo de alerta. Los tipos de AlertType definidos son los siguientes:

- AlertType.ALARM: este tipo de alerta indica un evento del que el usuario ha pedido ser notificado.
- AlertType.CONFIRMATION: indica la confirmación de acciones efectuadas por el usuario.
- AlertType.INFO: para proporcionar una simple información al usuario.
- AlertType.WARNING: utilizada para informar de alguna operación peligrosa o que no puede ser deshecha.
- AlertType.ERROR: utilizada para indicarle al usuario la ocurrencia de un error.

Para construir un objeto de este tipo se dispone de los siguientes métodos:

public Alert(String titulo)

public Alert(String titulo, String textoInformativo, Image icono, AlertType tipo)

El primer método construye el objeto Alert con el titulo especificado y sin un tipo especifico. El segundo construye el objeto con el titulo, texto informativo, icono y tipo señalado. Los valores de los parámetros en ambos métodos pueden valer null, un valor para tipo de null indica que la alerta no tiene un tipo asociado.

Para establecer la cantidad de tiempo que una alerta es mostrada se utiliza el método setTimeout():

public void setTimeout(int milisegundos)

El valor de milisegundos es un valor entero positivo o el valor especial Alert.FOREVER. Si el valor es Alert.FOREVER la alerta es modal. Se provoca un IllegalArgumentException si el valor de milisegundos es inválido.

Si el contenido del objeto Alert es demasiado grande, de tal forma que tenga que efectuarse un desplazamiento o scrolling para mostrarlo, la alerta es modal sin importar el parámetro milisegundos.

Para saber la cantidad de tiempo que la alerta será mostrada se utiliza el método getTimeout():

public int getTimeout()

Este método retorna el valor en milisegundos o Alert.FOREVER.

Para saber la cantidad de tiempo por defecto que la alerta puede ser mostrada se utiliza getDefaultTimeOut():

public int getDefaultTimeout()

En este caso el valor retornado es la cantidad de tiempo en milisegundos utilizados por la implementación o Alert.FOREVER para mostrar una alerta y no tiene relación alguna con el valor establecido en setTimeout(). Este valor es dependiente de la plataforma.

Para establecer el texto de la alerta se utiliza setString():

public void setString(String str)

Para recuperar el texto de una alerta se utiliza el método getString():

public String getString()

Para establecer el icono de la alerta se utiliza setImage()

public void setImage(Image img)

El parámetro Image puede ser una imagen mutable o inmutable (más adelante hablaremos de Image). También puede ser null si se desea eliminar una imagen previa.

Para recuperar la imagen se utiliza getImage():

public Image getImage()

El método retorna la Imagen de la alerta, o null si no hay ninguna.

El método setType() permite cambiar el tipo de la alerta:

public void setType(AlertType tipo)

El parámetro tipo es uno de los objetos estáticos AlertType descritos anteriormente o null si la alerta no tiene un tipo especifico.

Para saber el tipo de alerta mostrada se utiliza el método getType():

public AlertType getType()

Comandos dentro de Alertas

Siendo una clase derivada de Display, una instancia Alert puede contener comandos que pueden ser procesados por un CommandListener. Todos los objetos Alert comparten un comando especial llamado DISMISS_COMMAND:

public static final Command DISMISS_COMMAND

Un objeto Alert determinado puede añadir sus propios comandos, si la aplicación añade otros comandos, automáticamente el comando DISMISS_COMMAND es quitado. Si la aplicación quita todos sus comandos de un objeto Alert, el comando por defecto es restaurado nuevamente. Tratar de añadir o quitar el comando DISMISS_COMMAND no tiene ningún efecto, lo que significa que una alerta siempre contiene al menos un comando.

Si hay dos o más comandos presentes en la alerta, esta se vuelve modal y el valor retornado por getTimeout() es Alert.FOREVER.

Al crear una instancia Alert, esta contiene un CommandListener implícito llamado "default listener". Este puede ser reemplazado por uno proporcionado por la aplicación con el método setCommandListener(CommandListener l). Si este método es llamado con null como parámetro el manejador implícito es restaurado. Este manejador esta activo siempre y cuando haya un solo comando en la alerta y no se haya proporcionado otro manejador.

Los métodos Display.setCurrent(Alert, Displayable) y Display.setCurrent(Displayable) (cuando este se utiliza con un objeto Alert), definen un comportamiento especial, que permite mostrar una pantalla inmediatamente después de ocultar la alerta o restaurar la pantalla que estaba visible antes de mostrar la alerta. Este comportamiento ocurre sólo cuando el manejador de eventos de comandos especial está activo, lo que significa que si se instala otro manejador este debe encargarse de mostrar la pantalla correcta.

El siguiente fragmento crea y muestra un objeto Alert informativo, sin icono, modal:

```
Alert a=new Alert("MIDP App","Registro guardado",null,AlertType.INFO);
Display.getDisplay(midlet).setCurrent(a);
```

Ticker y Gauges en alertas

Una aplicación puede añadir un objeto Ticker a una alerta, pero el que esta sea mostrada depende de la implementación. También puede contener medidores de actividad o Gauges que deben cumplir ciertos requisitos, los cuales pueden ser consultados en la documentación oficial.

La clase Form

Una de las clases más importantes es Form, derivada de Screen, este tipo de objetos pueden contener items: imágenes, texto de sólo lectura, campos de texto modificables, campos de fecha editables, gauges (medidores), campos de selección, e items personalizados. En general cualquier objeto de una subclase derivada de Item puede estar contenido en un objeto Form. La implementación se encarga del posicionamiento o layout de cada item, y también de manejar el desplazamiento o scrolling.

Manejo de los items

Los items contenidos en un objeto Form pueden ser cambiados usando los métodos append(), delete(), insert() y set(). Los items en un formulario se identifican por un índice, que son enteros consecutivos que van desde 0 hasta size()-1 (el valor de retorno de size() es la cantidad de items que contiene el formulario).

Un item puede pertenecer a más de un formulario. Sin embargo para poder pertenecer a otro formulario el item debe ser primero liberado del formulario que lo contiene y luego insertarlo en el nuevo formulario; de lo contrario se provocará un IllegalStateException.

La forma de ordenamiento de los items en un formulario está organizada en filas. Todas las filas tienen el mismo ancho, aunque este pueda cambiar en ciertas circunstancias, por ejemplo al añadir o quitar barras de desplazamiento que por lo general solamente son verticales.

Para construir un objeto Form se dispone de dos métodos:

```
public Form(String titulo)

public Form(String titulo,Item[] items)
```

El primer método crea un objeto Form vacío con el titulo especificado en el parámetro. El segundo crea el objeto Form con un titulo y los componentes contenidos en el arreglo items. El segundo método lanza un IllegalStateException si alguno de los items pertenecen a otro objeto Form y un NullPointerException si alguno de estos items referencia null.

Para añadir un componente al formulario se utiliza una de las versiones el método append():

```
public int append(Image img)

public int append(Item item)

public int append(String str)
```

Todos los métodos añaden los componentes al final del formulario. Todos añaden un objeto derivado de Item, en el primer caso el objeto que se añade es un ImageItem construido a partir del objeto Image que se referencia por img; el segundo (la versión más general) permite añadir cualquier objeto derivado de Item y el la última versión permite añadir un objeto StringItem construido a partir del String especificado. Los tres métodos lanzan NullPointerException si su argumento referencia null, el segundo

método lanza además IllegalStateExeption si su argumento pertenece a otro formulario.

Para insertar un item en una posición específica uno de los siguientes métodos debe ser utilizado:

public void insert(int indice, Item componente)

public void set(int indice, Item componente)

El método insert agrega el item especificado en la posición indice-1, el valor indice puede estar entre cero y size(), inclusive. Si el valor de índice es size() el método se comporta como append(Item).El método set añade el componente especificado en el índice especificado y borra el componente con el índice anterior. El parámetro índice debe estar entre 0 y size()-1 inclusive. Ambos métodos provocan ArrayIndexOutOfBoundsException si el valor índice es ilegal, IllegalStateException si el parámetro item pertenece a otro formulario, y se provoca NullPointerException si componente referencia null.

Para obtener un item específico utilizamos el método get():

public Item get(int indice)

El parámetro índice debe estar entre 0 y size()-1 inclusive, de lo contrario se provoca un ArrayIndexOutOfBoundsException.

Para borrar un determinado componente utilizamos el método delete():

public void delete(int indice)

De nuevo el índice debe estar entre cero y size()-1, de lo contrario un ArrayIndexOutOfBoundsException es provocado.

Para borrar todos los elementos de un formulario utilizamos el método deleteAll()

public void deleteAll();

Form contiene otro método para monitorear los cambios en el contenido de un item:

public void setItemStateListener(ItemStateListener l)

Este método establece un objeto de una clase que implementa la interfaz ItemStateListener como el manejador de eventos para los items del formulario. El procesamiento se hace en el método itemStateChanged():

public void itemStateChanged(Item i)

Este método recibe como parámetro el item en el que se efectuó el cambio.

Componentes

La clase Item es una clase abstracta que sirve como base a la mayoría de componentes de interfaz de usuario en MIDP. Un componente o item es un elemento que puede ser añadido a un objeto Screen y con el que (usualmente) el usuario puede

interactuar. Comúnmente son utilizados con Form, pero algunos como Gauge pueden ser utilizados con otros objetos Screen como Alert.

La declaración de esta clase es la siguiente:

public abstract class Item extends Object {...}

Un componente posee una etiqueta para indicar el propósito de este. Esta puede ser modificada y recuperada a través de los métodos setLabel() y getLabel(), respectivamente:

public void setLabel(String etiqueta)

public String getLabel()

Un valor de null como parámetro o de retorno para el método correspondiente indica que el item no tiene etiqueta.

Posicionamiento:

Un objeto Item tiene directivas que modifican la posición en que se ubica dentro de un objeto Form:

- Item.LAYOUT_DEFAULT
- Item.LAYOUT_RIGHT
- Item.LAYOUT_TOP
- Item.LAYOUT_VCENTER
- Item.LAYOUT_NEWLINE_AFTER
- Item.LAYOUT_VSHRINK
- Item.LAYOUT_VEXPAND
- Item.LAYOUT_LEFT
- Item.LAYOUT_CENTER
- Item.LAYOUT_BOTTOM
- Item.LAYOUT_NEWLINE_BEFORE
- Item.LAYOUT_SHRINK
- Item.LAYOUT_EXPAND
- Item.LAYOUT_2

La directiva Item.LAYOUT_DEFAULT indica que el contenedor del item utiliza una política de posicionamiento por defecto. Las directivas Item.LAYOUT_LEFT, Item.LAYOUT_CENTER e Item.LAYOUT_RIGHT indican alineación horizontal, de forma similar Item.LAYOUT_TOP, Item.LAYOUT.VCENTER e Item.LAYOUT_BOTTOM indican alineación vertical con respecto al espacio asignado por el contenedor del item. Las directivas de alineación vertical, horizontal y otras directivas pueden ser combinadas con OR a nivel de bits (|) para crear posicionamientos un poco más complejos.

El establecimiento y recuperación de las directivas de posicionamiento para un componente se realiza a través de los métodos setLayout() y getLayout():

public void setLayout(int layout)

public int getLayout()

El método setLayout() provoca IllegalStateException si el item está contenido en una instancia Alert.

Para conocer el significado del resto de directivas, puede consultarse la documentación de la clase Form.

Un item puede tener comandos asociados, los cuales pueden ser añadidos utilizando el método addCommand() o setDefaultCommand() de la clase Item:

```
public void addCommand(Command c)

public void setDefaultCommand(Command c)
```

El primer método añade un comando al item, el segundo establece un comando por defecto, es decir un comando al que puede accederse fácilmente: por ejemplo al presionar una tecla determinada, aunque esto no es un requerimiento. Ambos métodos provocan un IllegalArgumentException si el item está contenido dentro de una instancia Alert.

Los comandos añadidos con estos métodos deberían tener el tipo Command.ITEM, aunque no es un error especificar otro tipo. Los comandos asociados a un item usualmente se vuelven visibles hasta que el item tiene el foco dentro de una pantalla o si el comando es único puede ser mostrado al lado del item; sin embargo esto puede ser dependiente de la implementación.

El procesamiento de un evento para un comando que pertenece a un Item se hace a través de la interfaz ItemCommandListener y de su método commandAction():

```
public interface ItemCommandListener {
        public void commandAction(Command c, Item I);
}
```

El método commandAction(), recibe como parámetros el comando que provoco el evento y el componente para el cual se provoco. Se provoca un IllegalStateException si el componente está contenido dentro de una alerta.

Para activar la escucha de un evento para una clase que implemente esta interfaz se utiliza el método setItemCommandListener de Item:

```
public void setItemCommandListener(ItemCommandListener I)
```

Apariencia de un item:

Auque la clase Item contiene campos que describen como deberían aparecer en la interfaz, estos campos se utilizan solamente en las subclases StringItem e ImageItem, por lo que los describiremos más adelante.

Jerarquía de componentes

Todos los componentes utilizados en una interfaz de usuario MIDP son derivados de Item. La jerarquía formada se muestra en la siguiente figura:

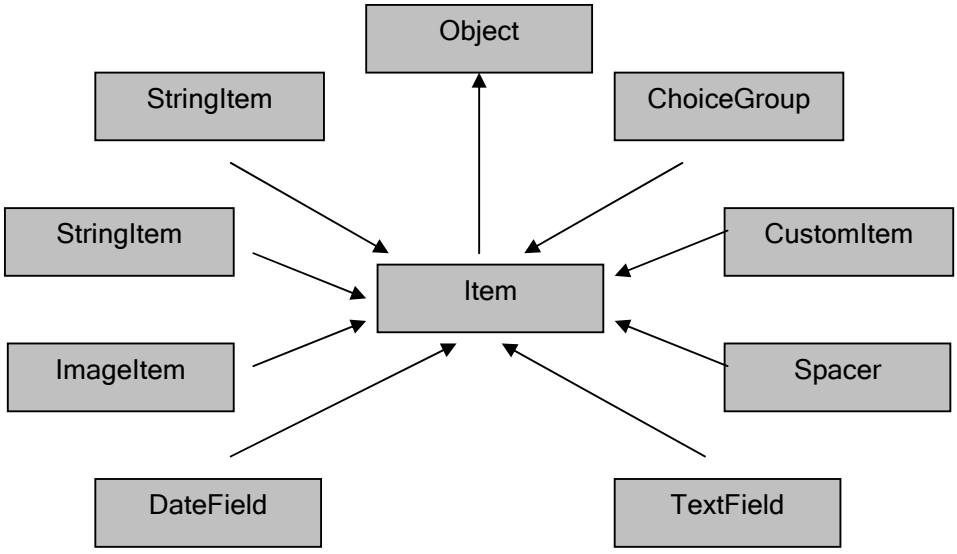


Figura 4: Jerarquía de componentes MIDP

StringItem

Un objeto de esta clase representa un componente con una etiqueta y un texto de sólo lectura para el usuario, la aplicación puede modificar ambas partes del componente. La apariencia de un componente de esta clase también puede ser establecida por la aplicación.

Esta clase está declarada de la siguiente forma:

public class StringItem extends Item{...}

Para construir un objeto de esta clase se disponen de los siguientes métodos:

public StringItem(String etiqueta, String texto)

public StringItem(String etiqueta, String texto, int apariencia)

La primera versión construye un objeto con la etiqueta y el texto especificado y una apariencia por defecto (StringItem.PLAIN).

La segunda versión construye un objeto con la etiqueta, el texto y la apariencia especificada. La apariencia es uno de los siguientes valores enteros heredados de la clase Item: BUTTON, HYPERLINK, PLAIN. Una apariencia BUTTON indica que el item se muestra como un botón, HYPERLINK indica que se muestra como un hipervínculo y PLAIN indica una apariencia por defecto (sólo el texto). Este método provoca un IllegalArgumentException si el valor de apariencia es incorrecto.

El siguiente fragmento crea dos objetos StringItem, uno con apariencia de botón y otro con la apariencia por defecto.

```
…
StringItem str1,str2;
str1=new StringItem(“Boton”,null,Item.BUTTON);
str2=new StringItem(“Etiqueta”,”Contenido”); ...
```

No hay forma de cambiar la apariencia de un objeto de este tipo una vez que ha sido creado.

Para establecer el texto de un objeto StringItem se utiliza el método setText():

public void setText(String texto)

Para obtener el texto de un objeto StringItem se utiliza el método getText():

public String getText():

Un objeto StringItem puede cambiar la fuente con que es mostrado utilizando el método setFont():

public void setFont(Font fuente)

Para obtener información de como se utilizan la clase Font en MIDP, consúltese la documentación oficial.

TextField

La clase TextField es similar a la clase TextBox en funcionalidad, sin embargo:

- TextBox es una clase derivada de Screen, por lo tanto ocupa toda la pantalla, TextField es una clase derivada de Item que ocupa espacio en un formulario.
- Textbox no tiene forma de reportar cambios en su contenido a un manejador de eventos, las modificaciones a un objeto TextField son reportadas a un ItemStateListener asociado al formulario en el que el TextField está contenido.

Para construir un objeto TextField se dispone del siguiente método:

public TextField (String etiqueta, String texto, int tamañoMaximo, int restricciones)

Este método construye un objeto TextField con la etiqueta y el contenido especificado en el parámetro texto. Con un número máximo de caracteres y las restricciones especificadas. Provoca un IllegalArgumentException si el parámetro tamañoMaximo es menor o igual a cero, si el valor de las restricciones es inválido, si el texto es inválido para las restricciones especificadas y la longitud de la cadena texto es mayor al tamaño máximo solicitado. El tamaño máximo solicitado puede o no ser concedido, si esto ocurre el contenido del TextField es truncado.

Una clase puede monitorear cambios en un objeto TextField implementando la interfaz ItemStateListener y su método itemStateChanged():

public void itemStateChanged(Item item)

Por implicaciones en el rendimiento, es posible que este método no sea llamado cada vez que ocurra un cambio en el objeto; sin embargo se garantiza que será llamado antes que el objeto pierda el foco o que el usuario active un comando en el formulario.

Salvo las variaciones descritas está clase se comporta de igual forma a TextBox. Por lo que los métodos descritos para esta última son aplicables también a TextField.

Las clases Image e ImageItem

Image

Esta clase representa una imagen, una instancia de este tipo existe de forma independiente del dispositivo de visualización, no es visualizada hasta invocar a un método de forma explicita, como paint() de la clase Canvas o al añadirlo a un objeto Form o Alert.

Esta clase se declara como sigue:

public class Image extends Object{...}

La imagen puede ser mutable o inmutable en dependencia de cómo haya sido creada. Las imágenes inmutables se crean al cargarlas de algún recurso, de archivos o desde una conexión externa y no pueden ser modificadas. Una imagen mutable es creada conteniendo sólo píxeles en blanco y la aplicación puede dibujar en ella a través de un objeto Graphics obtenido con el método getGraphics().

Un objeto Image puede estar contenido en un objeto Alert, Choice o ImageItem.

MIDP exige que las implementaciones soporten al menos el formato PNG (Portable Network Graphics- Gráficos de red portátil), sin embargo es posible que otros formatos estén soportados en una determinada plataforma.

Los métodos para crear un objeto Image son los siguientes:

static Image createImage(byte[] datos, int leerDesde, int longitud)

Este método retorna un objeto Image inmutable creado a partir de los datos contenidos en el arreglo datos y utilizando el rango que va desde leerDesde hasta leerDesde + longitud. El contenido del arreglo representa un formato de imagen decodificado y que es soportado por la implementación. Este método provoca ArrayIndexOutOfBoundsException si el rango especificado es inválido, NullPointerException si datos referencia null e IllegalArgumentException si el contenido de este arreglo no representa un formato soportado.

static Image createImage(Image fuente)

Este método retorna una imagen inmutable copia de la imagen especificada. Se provoca un IllegalArgumentException si fuente referencia null.

static Image createImage(Image fuente, int x, int y, int ancho, int alto, int transformacion)

Este método retorna una imagen inmutable utilizando como origen la región especificada del parámetro fuente aplicando la transformación seleccionada.

Los tipos de transformación están definidos por la clase javax.microedition.lcdui.Sprite (una clase que es parte de la API para videojuegos de MIDP):

- Sprite.TRANS_NONE: la región especificada se copia sin cambios.
- Sprite.TRANS_ROT90: la región especificada es rotada 90° en el sentido de las agujas del reloj.

- Sprite.TRANS_ROT180: la región especificada es rotada 180° en el sentido del las agujas del reloj.
- Sprite.TRANS_ROT270: la región especificada es rotada 270° en el sentido de las agujas del reloj.
- Sprite.TRANS_MIRROR: la región especificada es reflejada respecto a su centro vertical.
- Sprite.TRANS_MIRROR_ROT90: la región especificada se reflejada respecto a su centro vertical y luego se rota 90° en el sentido de las agujas del reloj.
- Sprite.TRANS_MIRROR_ROT180: la región especificada es reflejada respecto a su centro vertical y luego se rota 180° en el sentido de las agujas del reloj.
- Sprite.TRANS_MIRROR_ROT270: la región especificada es reflejada respecto a su centro vertical y luego se rota 270° en el sentido de las agujas del reloj.

Las dimensiones de la imagen retornada dependen de la región y la transformación aplicada. Se provoca un IllegalArgumentException si fuente referencia null, si la región especificada excede los límites de la imagen fuente, si el ancho o alto son menores o iguales a cero o si el parámetro transformacion es inválido.

static Image createImage(int ancho, int alto)

Este método retorna una imagen mutable con las dimensiones especificadas. Si cualquiera de las dimensiones es menor o igual a cero se provoca IllegalArgumentException.

static Image createImage(String nombreRecurso)

Crea una imagen inmutable con los datos obtenidos de una imagen especificada por el nombre de un recurso. Un recurso es un archivo no ejecutable que se encuentra dentro del Java archive (JAR) de la aplicación. Los datos del recurso son obtenidos a partir del método Class.getResourceAsStream(nombreRecurso). Este método provoca un NullPointerException si nombreRecurso referencia null y java.io.IOException si el recurso no existe, los datos no pueden ser leídos o la imagen no puede ser decodificada.

static Image createImage(java.io.InputStream is)

Este método es similar al anterior, retorna una imagen inmutable creada con los datos leídos del objeto InputStream, el que puede estar enlazado a un recurso o una conexión externa o un almacén de datos. Las mismas excepciones del método anterior se aplican para este.

static Image createRBGImage(int rgb[], int ancho, int alto, boolean procesarAlpha)

Este método retorna una imagen inmutable creada a partir de una secuencia de valores ARGB (Alpha, Red, Green & Blue – Alfa, Rojo, Verde y Azul) contenidos en el arreglo rgb. Cada elemento del arreglo está en el formato 0xAARRGGBB. Los datos están ordenados en el arreglo de izquierda a derecha y de arriba hacia abajo. Si el valor procesarAlpha es true el byte de mayor orden especifica la transparencia, si este parámetro es false este byte es ignorado. Este método provoca NullPointerException si el arreglo rgb es null, IllegalArgumentException si ancho o alto es menor o igual a cero, ArrayIndexOutOfBoundsException si la longitud del arreglo rgb es menor que ancho x alto.

ImageItem

Esta clase está derivada de Item, representa un componente que puede contener una imagen. Cada instancia contiene una referencia a un objeto Image mutable o inmutable. Si la imagen es mutable, su contenido se copia a una imagen inmutable al crear el objeto ImageItem, es decir se toma un "screenshot" o instantánea de la imagen.

Un objeto de este tipo contiene un texto alternativo que aparece en lugar de la imagen si esta es demasiado grande como para ser mostrada. Este texto alternativo puede ser null.

Un objeto de este tipo puede especificar una apariencia. Los campos que describen la apariencia de una instancia ImageItem son los mismos descritos para StringItem.

Para construir un objeto ImageItem se dispone de los siguientes métodos:

public ImageItem(String etiqueta, Image img, int posicionamiento, String txtAlternativo)

Este método crea un objeto ImageItem con la etiqueta especificada, utilizando la imagen img, aplicando el posicionamiento especificado. Si la imagen no puede ser mostrada, se utiliza el texto alternativo especificado. Se provoca un IllegalArgumentException si el valor del posicionamiento es inválido.

public ImageItem(String etiqueta, Image img, int posicion, String txtAlternativo, int apariencia)

Este método es similar al anterior, sin embargo contiene un parámetro adicional que permite especificar la apariencia. Se provoca un IllegalArgumentException si posicionamiento o apariencia son valores inválidos.

ImageItem posee otros métodos además de los heredados de Item:

public void setAltText(String str)

public String getAltText()

public void setImage()

public Image getImage()

Los métodos setAltText() y getAltText() permiten establecer y recuperar el texto alternativo asociado a un instancia ImageItem. Los métodos setImage() y getImage() permiten establecer y obtener una referencia al objeto Image asociado a la instancia.

ChoiceGroup

Una clase que proporciona la misma funcionalidad que la clase List. Sin embargo se diferencia de List en que:

- List se deriva de Screen lo que significa que una instancia de este tipo ocupa todo la pantalla, ChoiceGroup deriva de Item , por lo tanto un objeto de esta clase ocupa espacio en un objeto Form.

- Una instancia List no puede ser de tipo Choice.POPUP, una instancia ChoiceGroup puede tener este tipo pero no Choice.IMPLICIT.
- Para una instancia ChoiceGroup la selección de un elemento puede ser monitoreado por un ItemStateListener, una instancia List no se puede monitorear este tipo de cambios.

Para construir un objeto de este tipo se dispone de los siguientes métodos:

public ChoiceGroup(String etiqueta, int tipo)

public ChoiceGroup(String etiqueta, int tipo, String[] elementos, Image[] iconos)

Estos constructores trabajan de la misma forma que los constructores de List, sin embargo para un objeto List el parámetro etiqueta es el titulo de la pantalla, para ChoiceGroup es sólo un rotulo que identifica el item.

El resto de métodos de esta clase son los mismos que los de la clase List.

Gauge

Esta clase representa a los componentes medidores, como barras de progreso. Un objeto de este tipo esta asociado a un valor entero comprendido entre 0 y un valor máximo inclusive. Un medidor puede ser interactivo o no interactivo. En modo interactivo el usuario puede modificar el valor del medidor. En modo no interactivo el usuario no puede modificar el valor, este modo se utiliza típicamente para mostrar el progreso de alguna operación larga.

Un medidor puede tener un rango definido o no definido. Un rango indefinido implica que el medidor puede estar en cuatro estados:

- Gauge.CONTINUOS_IDLE y Gauge.INCREMENTAL_IDLE: en este el indicador muestra que no hay ningún trabajo en progreso.
- Gauge.CONTINUOS_RUNNING: no se reporta ningún progreso al usuario y no se conoce un fin de la operación.
- Gauge.INCREMENTAL_UPDATING: en este estado el progreso puede ser indicado al usuario aun sin saber el punto final de la operación.

Para construir un objeto de este tipo se dispone del siguiente método:

public Gauge(String etiqueta, boolean interactiva, int valorMaximo, int valorInicial)

Este método crea una instancia Gauge con la etiqueta, en el modo, con los valores máximo e inicial especificados. En modo no interactivo el valor máximo debe ser mayor a cero o igual al valor especial Gauge.INDEFINITE, de lo contrario se provoca IllegalArgumentException.

Si el valor máximo especificado es mayor que cero, el valor inicial debe estar entre 0 y este valor inclusive. Si el valor inicial es menor que cero, este se establece a cero; si el valor inicial es mayor que el máximo este se establece al máximo.

Si el parámetro interactiva es false y el valor máximo es Gauge.INDEFINITE, el valor inicial debe ser Gauge.CONTINUOS_IDLE, Gauge.INCREMENTAL_IDLE, Gauge.CONTINUOS_RUNNING o Gauge.INCREMENTAL_UPDATING, de lo contrario se provoca un IllegalArgumentException.

Para saber si una instancia Gauge es interactivo se dispone del método isInteractive():

```
public boolean isInteractive()
```

Este método retorna true si el usuario puede cambiar el valor del objeto.

Para establecer el valor actual de una instancia Gauge se dispone del método setValue():

```
public void setValue(int valor)
```

Para Gauges interactivos y no interactivos con rango definido: Si valor es menor que cero el valor actual se establece a cero, si es mayor que el valor máximo se establece a este valor.

Para Gauges no interactivos con rango indefinido: valor debe ser uno de los siguientes: Gauge.CONTINUOS_IDLE, Gauge.CONTINUOS_RUNNING, Gauge.INCREMENTAL_IDLE o Gauge.INCREMENTAL_UPDATING, de lo contrario se provoca un IllegalArgumentException.

Para obtener el valor actual de una instancia Gauge, se dispone del método getValue():

```
public int getValue()
```

Si el medidor es no interactivo con rango indefinido, el valor de retorno será uno de los siguientes: Gauge.CONTINUOS_IDLE, Gauge.CONTINUOS_RUNNING, Gauge.INCREMENTAL_IDLE o Gauge.INCREMENTAL_UPDATING.

Para establecer el valor máximo se dispone del método setMaxValue():

```
public void setMaxValue(int valorMaximo)
```

Para medidores interactivos el nuevo valor debe ser mayor que cero. Si el medidor tiene un rango definido y el valor actual es mayor que el nuevo valor máximo, se establece al nuevo valor máximo. Este método provoca un IllegalArgumentException si valorMaximo es inválido.

Para obtener el valor máximo de una instancia Gauge, se dispone del método getMaxValue():

```
public int getMaxValue()
```

Este método retorna un valor entero mayor que cero, o el valor especial Gauge.INDEFINITE.

Otras clases relacionadas con componentes

MIPD2.0 proporciona dos nuevas clases para la construcción de interfaces de usuario: Spacer y CustomItem.

La clase Spacer representa un item no interactivo sin interfaz, cuyo fin es proporcionar espacios para la colocación flexible de items en una misma fila o para provocar espacios mayores entre filas.

La clase CustomItem es una clase que proporciona acceso de bajo nivel a la API de alto nivel. Permite construir componentes propios que pueden ser puestos dentro de un objeto Form. El programador es responsable de manejar toda la interacción con el usuario, y de escribir todo el código asociado a la representación visual del componente.

Para obtener información sobre como se utilizan estas clases, consúltese la documentación oficial.

1.4.4.3 Almacenamiento persistente

En este aparatado se describe el mecanismo utilizado por MIDP para permitir a las aplicaciones almacenar datos de forma persistente en la memoria o disco duro del dispositivo.

El Record Management System

La mayoría de las aplicaciones generalmente necesitan guardar información que persista entre sesiones, en este sentido MIDP proporciona un sistema de almacenamiento denominado RMS. El record management system (sistema de manejo de registros) es el mecanismo proporcionado por MIDP para que las aplicaciones almacenen y recuperen datos persistentes de una forma consistente e independiente del dispositivo. De esta forma se asegura la portabilidad entre dispositivos que implementen diferentes formas de almacenamiento ya que el modo de manipulación de esos mecanismos (en este caso la API) no cambia. El eje central de RMS es la clase RecordStore, que está implementada en el paquete javax.microedition.rms.

Almacenes

Un almacén es una colección de registros almacenados de alguna forma en el dispositivo. Cada almacén es identificado por un nombre (case-sensitive) de entre 1 y 32 caracteres unicode, este es el nombre utilizado para acceder a nivel de aplicaciones. El identificador a nivel de dispositivo es una combinación del nombre del almacén más el identificador del midlet suite (propiedades midlet-vendor más midlet-name del Java application descriptor).

Por defecto un almacén es privado y solamente accesible sin restricciones por midlets que se encuentren en la misma suite del creador del almacén. Sin embargo en la versión MIDP2.0 es posible crear almacenes que pueden ser compartidos por todos los midlets del aparato. Estos almacenes se crean en ubicaciones dependientes del dispositivo, que no se exponen a ninguna aplicación (nativa o no).

Los almacenes compartidos pueden ser manipulados por aplicaciones MIDP externas, sin embargo sólo pueden ser borrados y su modo de acceso cambiado por un midlet que pertenezca al suite creador.

Las operaciones en un almacén son atómicas, sincronizadas y serializadas de forma que no hay corrupción en los datos al realizar múltiples accesos. Sin embargo si se utilizan hilos es responsabilidad de la aplicación coordinar los accesos para evitar resultados inesperados.

Para crear o abrir un almacén, una aplicación utiliza uno de los siguientes métodos estáticos de la clase RecordStore:

public static RecordStore openRecordStore(String nombre,boolean crear)

public static RecordStore openRecordStore(String nombre,boolean crear,int modoAcceso,
boolean modificable)

Ambos métodos localizan un almacén de datos, lo abren, y retornan un objeto RecordStore. Si el almacén referenciado por nombre no existe y crear es true, se crea un nuevo almacén. Si el almacén no existe y crear es false, se lanza una excepción RecordStoreNotFoundException para ambos métodos.

En el primer método el almacén es privado y modificable. En el segundo, el parámetro modificable establecido a false indica que el almacén es de sólo lectura, el parámetro modoAcceso indica el permiso de acceso para el almacén, el cual es uno de los siguientes valores:

- RecordStore.AUTHMODE_PRIVATE: Sólo accesible por midlets de este suite.
- RecordStore.AUTHMODE_ANY: Accesible por midlets de todo el dispositivo.

El siguiente código abriría un almacén perteneciente a la suite llamado “Agenda” en modo sólo lectura:

```
try
{
        RecordStore rs;
        rs=RecordStore.openRecordStore(
                                        “Agenda”,
                                        false,
                                        RecordStore.AUTHMODE_ANY,
                                        false);
}
catch(Exception e){ …}
```

Se observa que el modo de acceso utilizado es RecordStore.AUTHMODE_ANY que daría acceso al almacén a todos los midlets en el dispositivo, esto no es así ya que si el almacén ya existe este parámetro es ignorado y si no existe se provocaría una excepción. Para cambiar el permiso de un almacén, un midlet dentro del suite debe llamar al método no estático setMode(int authmode, boolean modificable).

Para cerrar un almacén previamente abierto, se utiliza el método no estático closeRecordStore(). Este método debe ser llamado tantas veces haya sido llamado el método openRecordStore() para realmente cerrar el almacén. Si el almacén no ha sido abierto el método lanza un RecordStoreNotOpenException.

Para borrar un almacén se utiliza el método estático deleteRecordStore():

public static void deleteRecordStore(String nombre)

Si el almacén no ha sido cerrado el método lanza una RecordStoreException. Si el nombre no hace referencia a un almacén existente se lanza un RecordStoreNotFoundException.

El siguiente fragmento borraría el almacén llamado “Agenda”:

```
try{
        ...
        rs.closeRecordStore();
        RecordStore.deleteRecordStore(“Agenda”);
        ...
} catch(Exception e){ …}
```

Registros

Un registro es un arreglo de bytes asociado a un recordId, que no es más que un número entero positivo único utilizado para su identificación. El recordId no es parte del registro y es asignado cuando el registro es creado. Los identificadores obedecen las siguientes reglas:

- El primer recordId es 1.
- El identificador asignado a un nuevo registro es uno más que el recordId del registro creado antes que él.

Por ejemplo: si se añaden 5 registros los recorIds serán: 1, 2, 3, 4, 5. Si luego borramos el registro con recordId igual 5 y añadimos un nuevo registro; los recordIds serán: 1, 2, 3, 4, 6, es decir los recordIds no se reutilizan.

Para añadir un nuevo registro a un almacén se utiliza el método no estático addRecord(), el cual retorna el identificador del nuevo elemento.

public void addRecord(byte[] datos, int desde,int cantidad)

El registro se crea con los bytes de datos[desde] hasta datos[desde+cantidad-1]. Aunque es posible llamar directamente a este método con un array de bytes, la mayor parte del tiempo se utilizan objetos para almacenar los datos, si se desea almacenar los datos de un objeto (no el objeto en sí) la forma más sencilla es utilizar un flujo de salida de bytes. Una combinación de la clase DataOutputStream y ByteArrayOutputStream proporciona este flujo.

En el fragmento siguiente se supone que tenemos un objeto RecordStore válido llamado re y queremos escribir una cadena de caracteres.

```
try
{       ByteArrayOutputStream bos=new ByteArrayOutputStream();
        DataOutputStream dos=new DataOutputStream(bos);
        dos.writeUTF(“¿Hola cómo estás?”);
        dos.close();
        byte[] bytes=bos.toByteArray();
        int id=rs.addRecord(bytes,0,bytes.length);…
}
catch(Exception e){… }
```

Para leer los datos de un registro previamente escritos, utilizamos el método no estático getRecord() de la clase RecordStore:
public byte[] getRecord(int recordId).

Este método lanza un InvalidRecordIDException si el recordId no existe.

Para leer un registro de un almacén la forma más sencilla es utilizar un flujo de entrada de bytes, las clases ByteArrayInputStream y DataInputStream del paquete Java.io pueden ser utilizadas con este propósito. Para leer la cadena previamente escrita (supondremos que la variable rId contiene el valor de un identificador válido) utilizaríamos el siguiente fragmento de código:

```
try
{
        ...
        ByteArrayInputStream bas=new ByteArrayInputStream(
        rs.getRecord(rId));
        DataInputStream dis=new DataInputStream(bas);
        String cadena=dis.readUTF();
        ...
}
catch(InvalidRecordIDException irie){ …. }
```

Para actualizar el contenido de un registro existente se utiliza el método no estático setRecord() de la clase RecordStore.

public void setRecord(int recordId, byte[] datos, int desde, int cantidad)

Ejemplo:

```
try
{
        ...
        ByteArrayOutputStream bas=new ByteArrayOutputStream();
        DataInputStream dis=new DataInputStream(bas);
        dis.writeUTF("¿Hola cómo estás?");
        dis.close();
        byte[] bytes=bas.toByteArray();
        rs.setRecord(rId,bytes,0,bytes.length);
        ...
}
catch(Exception e) { … }
```

No hay necesidad de que los registros tengan el mismo tamaño a la hora de una modificación.

Para borrar un registro utilizamos el método no estático deleteRecord() de la clase RecordStore.

public void deleteRecord(int recordId)

Monitoreo de cambios en un almacén

Los cambios en el contenido de un almacén se reportan por medio de la interfaz RecordListener. Para registrar una clase como escucha de cambios en el almacén hay que implementar está interfaz y utilizar el método addRecordListener() de la clase RecordStore. Esta interfaz contiene tres métodos:

```
public void recordAdded(RecordStore rs,int recordId)
public void recordChanged(RecordStore rs,int recordId)
public void recordDeleted(RecordStore rs,int recordId)
```

Estos métodos corresponden a las acciones añadir, modificar y borrar un registro respectivamente.

Para desactivar la escucha de eventos de un almacén se utiliza el método removeRecordListener() de la clase RecordStore. Todos los escuchadores de eventos son quitados al llamar a closeRecordStore() tantas veces como el método openRecordStore() haya sido llamado.

Los otros métodos de la clase RecordStore son:

```
public int getNumRecords()
public int getRecordSize(int recordId)
public int getNextRecordId()
```

El método getNumRecords() devuelve el número de registros existentes en el almacén. El método getRecordSize() devuelve la cantidad de bytes que ocupa el registro con el identificador dado. El método getNextRecordId() devuelve el identificador que será utilizado al agregar un nuevo registro, es decir el número que devolverá el método addRecord() en su siguiente invocación.

Enumeraciones de registros

Los métodos que la clase RecordStore proporciona son útiles para la manipulación del almacén, sin embargo algunos muy importantes (getRecord, setRecord y deleteRecord) exigen un conocimiento previo de los identificadores por parte de la aplicación.

Para obtener estos identificadores la clase RecordStore proporciona un método no estático llamado enumerateRecords():

```
public RecordEnumeration enumerateRecords(RecordFilter rf, RecordComparator rc,
                                          boolean sincronizar)
```

El argumento rf es un objeto que implementa la interfaz RecordFilter y permite establecer un filtro para los registros que se incluirán dentro del objeto RecordEnumeration retornado. El parámetro rc es un objeto que implementa la interfaz RecordComparator que permite establecer el orden de los registros. El parámetro sincronizar establecido a true permite mantener actualizado el contenido del objeto RecordEnumeration con los datos en “disco”, es decir cada cambio que se haga al almacén (agregar, eliminar o modificar registros) reconstruye la enumeración.

La siguiente línea construye una representación “estática” de un almacén previamente abierto y referenciado por el objeto rs.

RecordEnumeration re=rs.enumerateRecords(null,null, false);

El numero de entradas en la enumeración puede ser obtenido llamando al método numRecords(). En este ejemplo, en el que el filtro es null el número de registros en la enumeración es igual al número de registros del almacén. El que el comparador sea también null indica que el orden de los registros en la enumeración no está definido.

La interfaz RecordEnumeration contiene métodos que pueden ser utilizados para iterar sobre los identificadores de registros que contiene. A diferencia de una enumeración Java convencional, con esta enumeración es posible desplazarse tanto hacia atrás como hacia delante. Los métodos hasNextElement() y hasPreviousElement() permiten verificar si los limites de la enumeración han sido alcanzados. Los métodos nextRecordId() y previousRecordId() son usados para obtener los elementos de la enumeración (no los datos de los registros).

En el siguiente fragmento iteramos avanzando sobre una enumeración llamada enum:

```
…
while(enum.hasNextElement())
{
        int id=enum.nextRecordId();
        //realizar algo con el id obtenido
}
…
```

Cuando una enumeración es creada, su posición lógica está antes del primer elemento; cada vez que el método nextRecordId() es llamado esa posición se incrementa y se retorna el identificador de la nueva posición.

Sobrepasar los limites de una enumeración, provoca el lanzamiento de un InvalidRecordIDException.

Para obtener los datos de un registro se utilizan lo métodos nextRecord() o previousRecord().

public byte[] nextRecord()
public byte[] previousRecord().

Estos métodos son convenientes sólo para acceder a los datos del registro, pero no para modificarlos.

RecordEnumeration tiene un método llamado reset() que permite regresar la enumeración al estado que tenía justo después de ser creada.

Cuando al llamar a enumerateRecords() de la clase RecordStore se especifica el parámetro sincronizar a false, el objeto RecordEnumeration es “estático” es decir que cambios hechos al almacén no serán reflejados en la enumeración:

- Registros añadidos posteriormente, no aparecen en la enumeración.
- Se lanza una InvalidRecordIDException, si se accede a los datos de un registro que ya ha sido borrado.

Para mantener actualizada la enumeración con los cambios en el almacén, se debe poner el parámetro sincronizar a true al llamar a enumerateRecords(). Sin embargo el

hacer esto conlleva una degradación en el rendimiento, ya que cualquier cambio efectuado en el almacén provocará una reconstrucción de esta.

Los otros métodos de la interfaz RecordEnumeration son:

```
public boolean isKeepUpdated()
public void keepUpdated(boolean sincronizar)
public void rebuild()
```

El método isKeepUpdated() retorna true si la enumeración esta sincronizada. El método keepUpdated() se utiliza para indicar si la enumeración está o no sincronizada con el almacén. El método rebuild() reconstruye la enumeración con el estado del almacén.

Para finalizar, cuando ya no se necesita una enumeración se debe llamar al método destroy() que liberará todos los recursos asociados al objeto.

Para más detalles sobre la clase RecordStore, el resto de interfaces y excepciones de esta API, consúltese la documentación oficial.

1.4.4.4 Soporte para comunicaciones

Los dispositivos para los que J2ME está diseñado son, por naturaleza, útiles debido a las capacidades que tienen de comunicarse con aparatos externos. Los teléfonos móviles, por supuesto, no tienen otro propósito que existir en una red, mientras que los PDA serían menos útiles si no pudieran ser conectados a una computadora externa para descargar o guardar datos. Estas capacidades de red, al ser tan importantes, deben pagar un precio en términos de recursos necesarios para el software que implementa los diferentes protocolos de red usados actualmente. Dadas las capacidades de procesamiento relativamente pequeñas de los aparatos móviles existentes, es necesario cumplir ciertos compromisos de diseño para proporcionar soporte de red para el tipo de hardware en el que los perfiles diseñados para CLDC se ejecutan.

En esta parte, se abordará parte del API que CLDC/MIDP proporciona para agregar capacidades de comunicación a las aplicaciones.

1.4.4.4.1 Arquitectura de red para dispositivos móviles

J2SE proporciona una infraestructura de red a bajo nivel, la cual es implementada en el paquete java.net. En J2ME las capacidades de red básicas son definidas en las configuraciones, en este caso CLDC, estas capacidades son extendidas en los perfiles que dependen de estas. En vez de utilizar funcionalidades proporcionadas por java.net; CLDC define una nueva capa o framework en la que está basada toda la conectividad proporcionada por sus perfiles dependientes. Este diseño se debe a lo siguiente:

- Requerimientos de memoria**:** el paquete java.net al momento de escribir estás líneas y utilizando la versión 1.4 de J2SE contiene 27 clases, 6 interfaces y 11 excepciones, además de las clases que se heredan de este paquete. Los requerimientos de este conjunto de clases son demasiado grandes para dispositivos con los recursos para los que CLDC está diseñado.
- Consistencia: el paquete de J2SE soporta acceso de bajo nivel a través de sockets y el uso de protocolos como http, esto hace que la forma de operación de ambos modelos sea diferente. Por ejemplo para realizar una conexión a

nivel de sockets se requiere un objeto InetAddress y un número de puerto para construir un objeto Socket. Si se desea trabajar con http primero se necesita un objeto URL para obtener otro objeto URLConnection. Si se desea trabajar con otro tipo de conexión se deben utilizar otros paquetes y clases. Debido a la gran cantidad de dispositivos y mecanismos de comunicación en los que CLDC puede funcionar, es necesario una API más uniforme.

- Flexibilidad en la implementación**:** en el paquete java.net, la mayor parte de la API gira en torno a las clases que son directamente accedidas por el programador y que permiten trabajar de diferentes maneras, por ejemplo a nivel de sockets o con protocolos específicos como http. En el contexto de J2ME, por el contrario, la forma en que un dispositivo se comunica con el exterior puede ser específica de este: una agenda electrónica puede tener conexión directa al Internet, mientras que un teléfono celular utiliza otro mecanismo para proporcionar esta conectividad. Para lograr una implementación sencilla y uniforme, los diseñadores de CLDC decidieron plasmar la arquitectura de red casi sólo con interfaces, de esta forma la aplicación no esta ligada a una clase en particular. Esta API es conocida como Generic Connection Framework (GCF).

El GCF está implementado en el paquete javax.microedition.io.

Las clases Connection y Connector

La interfaz básica del CFG es Connection, la cual representa una conexión de cualquier tipo. A este nivel, las únicas operaciones que se pueden efectuar sobre una conexión es cerrarla, esta interfaz está declarada así:

```
public interface Connection
{
        public void close() throws IOException;
}
```

Connection no posee un método open() para abrir una conexión. Todas las conexiones son obtenidas de la una de las dos clases del paquete: Connector, la cual está declarada así:

```
public class Connector extends Object{...}
```

Esta clase posee tres métodos estáticos open(), utilizados para abrir una conexión:

```
public static Connection open(String nombre)

public static Connection open(String nombre, int modo)

public static Connection open(String nombre, int modo, boolean timeouts)
```

El parámetro nombre especifica el tipo de conexión requerido, el formato general es:

```
esquema: dirección parámetros
```

Donde esquema indica el nombre del protocolo, por ejemplo http.; dirección es algún tipo de dirección de red y parámetros esta formado por una serie de pares de valores de la forma:";x=y".

Por ejemplo: http://ebay.com/

Los únicos protocolos soportados oficialmente por todas las implementaciones en la versión MIDP2.0 son http y https. En nuestro caso abordaremos la API relacionada con http.

Los parámetros adicionales del método open() especifican el propósito de la conexión en sí. El parámetro modo puede tomar uno de los siguientes valores: Connector.READ, Connector.WRITE o Connector.READ_WRITE, el cual es el parámetro por defecto. Si se utiliza un modo que no es soportado por un dispositivo o protocolo, se provoca un IllegalArgumentException. El parámetro timeouts opcional, indica que la aplicación puede hacer uso de tiempos máximos en operaciones de lectura y escritura, si son soportados por la implementación. Si la implementación soporta timeouts y este parámetro es establecido a true, un InterruptedIOException es provocado al vencer el tiempo en una operación de lectura o escritura. Esto es utilizado típicamente para evitar que una operación de lectura sea bloqueada indefinidamente. Sin embargo el tiempo que una operación puede estar bloqueada es dependiente de la implementación y no puede ser establecido por la aplicación.

Todas las versiones de este método provocan las siguientes excepciones:

- IllegalArgumentException si cualquiera de los parámetros es inválido.
- ConnectionNotFoundException si la dirección del recurso no puede ser encontrada o si el protocolo no está implementado.
- IOException si ocurre algún error de lectura o escritura.
- SecurityException si el acceso al protocolo no está permitido.

Tipos de conexión

Las interfaces InputConnection y OutputConnection están derivadas directamente de Connection. Estas interfaces proporcionan la capacidad de obtener flujos de entrada y salida respectivamente, asociados a una conexión.

Los métodos proporcionados por la interfaz InputConnection son:

```
public InputStream openInputStream() throws IOException

public DataInputStream openDataInputStream() throws IOException
```

OutputStream proporciona los métodos correspondientes para escribir en una conexión:

```
public OutputStream openOutputStream() throws IOException

public DataOutputStream openDataOutputStream() throws IOException
```

InputStream y OutputStream proporcionan acceso directo a nivel de bytes a un flujo de datos de entrada y salida respectivamente. DataInputStream y DataOutputStream permiten trabajar con tipos java primitivos como int, char, String, etc.

El hecho de que existan dos interfaces que definen métodos para leer y escribir desde y hacia una conexión se debe a que es posible que existan dispositivos o protocolos unidireccionales (al menos en lo que a transferencia de datos se refiere) y por lo tanto se retorne una subinterfaz de Connection que sólo permita leer o escribir. En donde

exista transferencia de datos bidireccional, la implementación puede retornar un objeto que implemente la interfaz StreamConnection, que combina los métodos de las interfaces InputConnection y OutputConnection en una sola:

public interface StreamConnection extends InputConnection,OutputConnection;

Un objeto StreamConnection ofrece la funcionalidad de enviar una secuencia de bytes desde un emisor hasta un receptor, pero deja la interpretación del contenido del flujo de bytes a las partes involucradas.

Si se desea una estructura más ordenada de los datos que son intercambiados, el protocolo puede utilizar la interfaz ContentConnection, que añade los siguientes métodos a StreamConnection.

public long getLength();

public String getType();

public String getEncoding();

Estos métodos retornan la longitud del contenido que se proporciona, el tipo de este contenido y la codificación utilizada, respectivamente.

Para leer datos de una conexión se utilizan las clases InputStream y DataInputStream.

Veamos un ejemplo del uso de InputStream:

```
...
//conn es una conexión válida de tipo StreamConnection
InputStream is=conn.openInputStream();
int len;
len=conn.getLength();
if(len>0)
{
        int bytes_leidos=0,actual=0;
byte[] buffer=new byte[len];
        try{
           while(bytes_leidos!=len && actual!=-1){
        actual=is.read(buffer,bytes_leidos,len-bytes_leidos);
                bytes_leidos+=actual; }
        }catch(Exception e){ … }
}
...
```

Veamos el uso de DataInputStream:

```
...
//conn es una conexión válida de tipo StreamConnection
DataInputStream dis=conn.openDataInputStream();
int len;
//leemos los 256 primeros bytes
len=conn.getLength();
```

```
if(len>0)
{
        try
{
                byte[] buffer=new byte[len];
                dis.readFully()
        //procesar los datos
        }catch(Exception e){}
}
...
```

Como puede observarse el uso de DataInputStream es mucho más sencillo que el de InputStream. De hecho DataInputStream (que es derivada de InputStream) proporciona métodos para leer del flujo de bytes tipos primitivos de datos Java.

Conexiones http

El único protocolo que todo dispositivo compatible con MIDP debe implementar es http, el cual hace uso de sockets para enviar mensajes entre un cliente (usualmente un explorador Web) y un servidor que a menudo – pero no siempre – retorna una página HTML.

La interfaz HttpConnection define los métodos y constantes necesarias para utilizar una conexión http.

Para crear una conexión http se invoca a Connector.open(), en este caso el parámetro de este método debe ser una cadena de caracteres con el siguiente formato:

http://host:puerto/ruta?parametros

Donde:

- http: es el nombre del protocolo. La especificación MIDP requiere soporte de http1.1, en MIDP2.0 se añade el soporte obligatorio de http seguro (HTTPS).
- host:puerto es el nombre y número de puerto del servidor. Si no se especifica un puerto, se utiliza el 80.
- ruta: una ruta relativa al directorio raíz del servidor Web que identifica el recurso que deseamos.
- parámetros: utilizados para obtener contenido dinámico del servidor basado en el valor de los parámetros.

Una conexión http existe en tres estados diferentes:

- Preparada, en los que los parámetros pueden ser establecidos.
- Conectada, en los que los parámetros han sido enviados y se espera una respuesta.
- Cerrada, en el que la conexión ha sido finalizada.

Los siguientes métodos de la interfaces pueden ser invocados sólo al estar en el estado preparado:

public void setRequestMethod(String metodo)

Este método establece la forma de consulta, sólo HttpConnection.GET, HttpConnection.HEAD y HttpConnection.POST son soportados. HttpConnection.GET es el método por defecto.

public void setRequestProperty(String propiedad, String valor)

Este método establece el valor de una propiedad o cabecera de petición. Esta propiedad puede ser recuperado con el método getRequestProperty().

La transición del estado preparado al conectado es causada por cualquier método que requiera que datos sean enviados al o recibidos desde el servidor. Esos métodos son los siguientes:

public InputStream openInputStream();
public DataInputStream openDataInputStream();
public int getLength();
public String getType();
public String getEncoding();

Todos estos métodos son heredados de la interfaz ContentConnection y sus superinterfaces.

public String getHeaderField(String campo);

Este método permite extraer el valor del campo especificado de la cabecera de una respuesta http. El método retorna el valor como un objeto String o null si el campo no existe.

public int getResponseCode();

Este método retorna el código de respuesta del servidor. Por ejemplo de una respuesta como:

HTTP/1.0 200 OK

Este método extraería el valor entero 200.

public String getResponseMessage();

Este método extraería el mensaje de respuesta del servidor. Partiendo del ejemplo anterior, retornaría “OK”.

public int getHeaderFieldInt(String nombre, int def);

public long getHeaderFieldDate(String nombre, long def);

Estos métodos son utilizados con campos de la cabecera que contienen valores numéricos, por ejemplo “date”. Son una versión más conveniente de getHeaderField(), si el campo no se encuentra en la cabecera se retorna el valor especificado en el parámetro def.

public long getExpiration();

public long getDate();
public long getLastModified();

Estos métodos retornan el valor de los campos “expires”, “date” y “last-modified” respectivamente, de la cabecera de una respuesta http o cero si no se conoce el valor. El valor retornado es el número de milisegundos transcurridos desde Enero 1,1970 GMT. Se provoca un IOException si se da un problema con la conexión.

public String getHeaderFieldKey(int indice);

Este método retorna el nombre de una cabecera identificada por indice o null si indice está fuera de rango. Se provoca un IOException si existe un error en la conexión.

Los siguientes métodos pueden ser invocados cuando una conexión se encuentra en el estado preparado o conectado:

public void close()

Este método es heredado de la interfaz Connection y permite cerrar la conexión.

public String getRequestMethod()

Este método devuelve la forma utilizada para efectuar la petición. El valor retornado es uno de HttpConnection.HEAD, HttpConnection.POST o HttpConnection.GET.

public String getRequestProperty(String nombre)

Este método devuelve el valor de una propiedad especificada por el parámetro nombre o null si está propiedad no existe.

public String getURL()

Este método devuelve la parte que hace referencia al URL en la petición. Por ejemplo en http://www.ebay.com, devolvería http://www.ebay.com

public String getProtocol()

Devuelve un objeto String que contiene el nombre del protocolo utilizado. Por ejemplo: “http” o “https”.

public String getHost()

Devuelve el nombre del host. La cadena puede ser la representación de un nombre o una dirección Ipv4. Por ejemplo en http://www.sun.com devolvería “www.sun.com”.

public String getFile()

Devuelve la parte relacionada al archivo del URL utilizada por la conexión. Por ejemplo en http://www.forum-nokia.com/documents/nds.pdf retornaría “/documents/nds.pdf”.

public String getRef()

Retorna la parte ref del URL utilizado por la conexión. Por ejemplo en http://www.nokia.com/phones.html#nokia6120 devolvería “nokia6120”.

public int getPort()

Retorna la parte puerto del URL utilizado por esta conexión. Por ejemplo en http://www.gmail.com retornaría 80, que es el puerto por defecto.

public String getQuery()

Retorna la parte query del URL utilizado por esta conexión. Por ejemplo en http://www.google.com/search?topic=dario retornaría “topic=dario”.

Flujos simplificados

La clase Connector define los siguientes métodos que permiten obtener flujos de entrada y salida directamente de un URL especificado:

public static InputStream openInputStream(String url)

public static DataInputStream openDataInputStream(String url)

public static OutputStream openOutputStream(String url)

public static DataOutputStream openDataOutputStream(String url)

Estos métodos sólo permiten el acceso a los flujos de datos, no a las conexiones. Esto reduce la complejidad al trabajar con cualquier conexión, sin embargo también impide la manipulación directa de esta.

Segunda Parte:

Servicios basados en localización.

2.1 Introducción

En años recientes la industria de teléfonos móviles ha visto un incremento exponencial en el número de usuarios de aparatos de este tipo, esto se debe principalmente al mejoramiento de las prestaciones en los equipos, los cuales se han convertido prácticamente en computadoras de bolsillo.

Dada la fuerte competencia entre compañías proveedoras de telefonía celular, estas buscan quedarse con la mayor cantidad de clientes. Muchas buscan atraerlos con mejores ofertas y teléfonos más atractivos, sin embargo, esta claro que al consumidor no le sirve de mucho un aparato con increíbles funcionalidades si no puede maximizar su uso; es por esto que los proveedores están apuntando a ofrecer servicios avanzados. Entre estos servicios están los basados en localización.

El origen de los servicios de localización se remonta a los años 70, cuando el departamento de defensa de Estados Unidos creó el GPS (Global Positioning System - Sistema de Posicionamiento Global) y la extinta Unión Soviética el GLONASS (Global Orbiting Navigation Satellite System- Sistema de Satélites de Navegación de Orbita Global), ambos con propósitos militares. Estos sistemas han sido liberados en su uso por sus respectivos gobiernos (el GPS en la década de los 80 y recientemente el GLONASS). Esto ha permitido a las industrias crear nuevos productos o servicios y mejorar los existentes.

Es hasta ahora que los servicios basados en localización empiezan a jugar un papel protagónico en las funcionalidades ofrecidas a los usuarios de telefonía celular por parte de los proveedores, ya que su acceso empieza a generalizarse, principalmente impulsado por nuevas tecnologías de posicionamiento. Debido a esta importancia adquirida, las principales plataformas de desarrollo para móviles (J2ME, BREW y Windows Mobile) y PDA los han integrado y ofrecen la posibilidad de desarrollar este tipo de aplicaciones. En este apartado abordaremos como es la integración de SBL con J2ME, pero antes abordaremos algunos conceptos relacionados a los mismos.

¿Qué son en sí los servicios basados en localización?

Servicios basados en localización (LBS) pueden ser definidos como aquellas aplicaciones que integran información sobre la posición geográfica de un dispositivo, para proporcionar un servicio o experiencia mejorada al usuario.

Los servicios de este tipo son muy adecuados en redes de telefonía móvil, dada la naturaleza de los equipos.

2.2 Clasificación

Estos servicios se clasifican en orientados al usuario y orientados al dispositivo:

- Servicios orientados al usuario son todas aquellas aplicaciones donde el servicio está basado en el usuario. Es decir, que la aplicación se enfoca en localizar a una persona o utilizar la posición de una persona para mejorar el servicio. Usualmente la persona que es localizada puede controlar el servicio, p. Ej.: Una aplicación de búsqueda de amigos.

- Servicios orientados al dispositivo son aquellos externos al usuario. Además de una sola persona, también un objeto (p. Ej. un carro) o un grupo de personas pueden ser localizados. En aplicaciones orientadas al dispositivo la

persona u objeto que se localiza no controla el servicio, p. Ej.: Búsqueda de un automóvil para recuperación después de un robo.

Además de las dos clasificaciones anteriores se distinguen dos tipos de servicios: push y pull.

- Servicios push: estos implican que el usuario recibe información sin solicitarla de forma explicita, pero como resultado de sus preferencias. La información pudo haber sido enviada al usuario con un consentimiento previo (p. Ej. un sistema de alertas de emergencias) o sin consentimiento previo (p. Ej.: un mensaje de bienvenida al entrar a una ciudad).

- Servicios pull: estos implican que el usuario explícitamente solicita la información enviando una petición a la red.

Muchos servicios pueden combinar ambos tipos de funcionalidad.

La siguiente tabla enumera algunos ejemplos de servicios push y pull:

	Servicios PUSH	Servicios PULL
Orientado al usuario		
Comunicaciones	❑Se recibe la alerta de que una persona ha entrado a un área de cercana. ❑Se envía un mensaje a una persona de que alguien quiere localizarla.	❑Una persona solicita información sobre personas conocidas ubicadas en una determinada área.
Información	❑Se envía información sobre eventos ocurriendo cerca de una ubicación.	❑Se busca el cine más cercano, e instrucciones de cómo llegar a él.
Publicidad y Comercio móvil	❑Se envía un cupón electrónico de descuento a todas las personas cercanas a un determinado centro comercial.	❑Se buscan por ciertos eventos ocurriendo en una determinada área.
Orientados al dispositivo		
	❑Una aplicación de seguimiento de paquetes envía una alerta que indica que un paquete determinado ha sido desviado de su ruta.	❑Se solicita información sobre la ubicación de un vehículo de una flota.

2.3 El modelo de comunicación en SBL

La puesta en marcha de servicios basados en localización, puede ser descrita en tres capas:

- Capa de posicionamiento.
- Capa middleware.
- Capa de aplicación.

Figura 5: Modelo de comunicación de los servicios basados en localización.

La capa de posicionamiento es la encargada de calcular la posición del usuario o dispositivo. Esto lo realiza con la ayuda de algún equipo de determinación de posicionamiento (Positioning Determination Equipment- PDE) y datos geoespaciales almacenados en un sistema de información geográfica (Geographic Information System – GIS). El PDE calcula donde se encuentra el dispositivo con respecto a la red, mientras que el GIS traduce esta información en términos de longitud y latitud o a una forma semántica. El resultado de esta operación es enviado a una pasarela, la cual lo envía directamente a una aplicación o a una plataforma middleware.

Originalmente la capa de posicionamiento, manejaba y enviaba la información de localización directamente a la aplicación que la había solicitado para prestar el servicio. La capa de aplicación incluía todos aquellos servicios que integraban la información proporcionada por la capa de posicionamiento.

¿Dónde queda la capa middleware?

La capa middleware se utiliza para permitir una integración más fácil entre los diferentes servicios. Esto permite proporcionar los mismos servicios a más aplicaciones, y mantener una complejidad relativamente más baja que si estos estuvieran ligados a una sola aplicación o servicio.

La siguiente figura aclara lo expuesto:

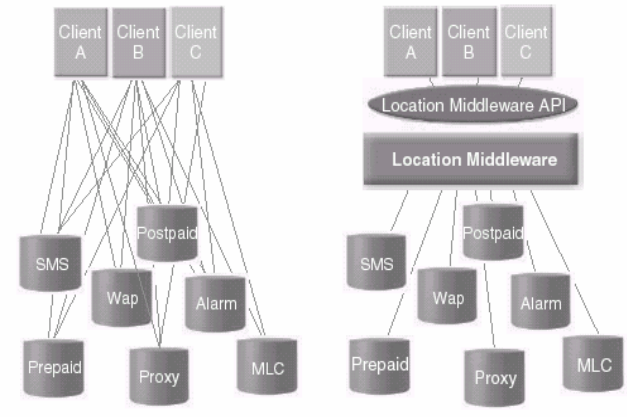


Figura 6: Comparativa entre los servicios que no utilizan un middleware y los que sí.

Además, por medio de una capa middleware el operador puede permitir a sus usuarios manejar preferencias sobre la interacción que pueden o no tener las aplicaciones con sus aparatos.

A continuación examinaremos con más detalle cada una de las capas antes mencionadas.

2.3.1 Capa de posicionamiento

La capa de posicionamiento se encarga de proporcionar los datos de localización. Estos datos poseen diversas características: sistemas de coordenadas utilizado (latitud, longitud, altitud), alcance (limitado a un área determinada, por Ej. un edificio), cobertura (parte del área de alcance en la que se dispone de la información, p. Ej. un cuarto dentro del edificio), precisión, datos adicionales (orientación, velocidad), información semántica (una dirección textual en vez de una coordenada); la disponibilidad de estos datos depende de la técnica utilizada para determinar la localización.

Técnicas básicas de posicionamiento

Los sistemas utilizados para obtener la localización de un usuario pueden ser divididos en dos categorías: localización (tracking) y posicionamiento (positioning).

Hablamos de localización (tracking) cuando es un censor en la red el que determina la ubicación del aparato, en este caso este se encarga de enviar la información al aparato.

Hablamos de posicionamiento (positioning) cuando es el aparato el que determina su propia posición. Por ejemplo el aparato efectúa el cálculo partiendo de señales (de radio, infrarrojo, etc.) constantes recibidas por un conjunto de transmisores; lo que significa que esta información está siempre disponible a todos los dispositivos.

Muchos sistemas que utilizan técnicas “tracking” son referidos como basados en redes, aquellos que utilizan técnicas “positioning” son referidos como basados en el dispositivo. Aquellos que utilizan ambos se conocen como híbridos.

Muchos sistemas que utilizan localización y posicionamiento en redes móviles se fundamentan en las siguientes técnicas básicas, las que a menudo son combinadas:

- Celda de origen (Cell Of Origin- COO): esta técnica utiliza el área de cobertura de una señal de una estación base en la red, esta área se conoce como celda y es utilizada para identificar la posición del aparato. La precisión depende del tamaño de la celda y puede llegar a tener errores de hasta 150 metros. Está es la técnica más barata y sencilla de implementar.
- Hora de llegada (Time Of Arrival - TOA), Diferencia en la hora de llegada (Time Difference of Arrival: estas técnicas están basadas en la velocidad a la que viajan las señales transmitidas. Para calcular la posición de un aparato se mide el tiempo que se tarda en recibir una respuesta de varias estaciones base a partir de una señal enviada. Si se mide el tiempo de diferencia entre dos señales se utiliza el término Time Difference of Arrival[1] (TDOA).
- Angulo de llegada (Angle Of Arrival- AOA): en esta técnica se utilizan un conjunto de antenas direccionales para encontrar la dirección desde la que una determinada señal llega.

[1] En redes GSM a menudo se utiliza el término E-TDOA (Enhaced TDOA).

- Medición de la intensidad de la señal: esta técnica mide la intensidad de las señales electromagnéticas recibidas, la cual disminuye al aumentar la distancia. Sin embargo en áreas urbanas este método no es preciso debido a la existencia de obstáculos (edificios, árboles, etc.).
- Procesamiento de datos de video: este método utiliza datos visuales obtenidos por cámaras de video. Lo que se hace es procesar esos datos en búsqueda de patrones que permitan determinar la localización de un determinado usuario. Este tipo de sistemas están basados en AOA: cada píxel de la imagen representa un cierto ángulo relativo al eje visual de la cámara.

Todos los sistemas de posicionamiento que mencionaremos a continuación utilizan alguna o varias de las técnicas anteriores.

2.3.1.1 Sistemas de posicionamiento basados en satélites

GPS

Es el sistema de posicionamiento basado en satélites más conocido. Inicialmente creado en la década del 1970, fue declarado completamente funcional el 8 de diciembre de 1995.

Está dividido en tres segmentos:

- El segmento de usuario: está compuesto por los dispositivos móviles de los usuarios (receptores).
- El segmento espacial: compuesto por los satélites.
- El segmento de control: es utilizado para administrar los satélites y corregir datos internos y orbitas. Está compuesto de varias estaciones de control que constantemente supervisan las señales de cada satélite y efectúan correcciones que son enviadas a una central de control en Colorado, EE.UU.

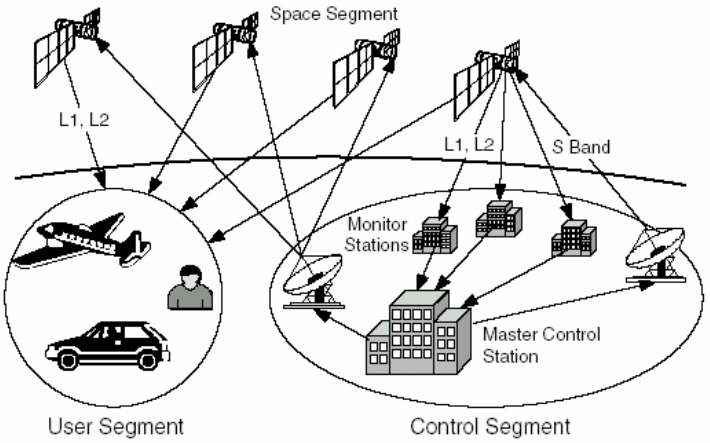


Figura 7: Sistema GPS

El sistema GPS consta de 24 satélites que se mueven en 6 orbitas diferentes. Cada satélite orbita la tierra a unos 20200 Km. de altura con respecto a la superficie terrestre y tarda 12 horas aproximadamente para efectuar una orbita completa. El movimiento es tal que desde cualquier punto de la tierra al menos 5 y no más de 11 satélites son "visibles" en un momento determinado (aunque no siempre se puedan recibir todas las señales debido a la existencia de obstáculos).

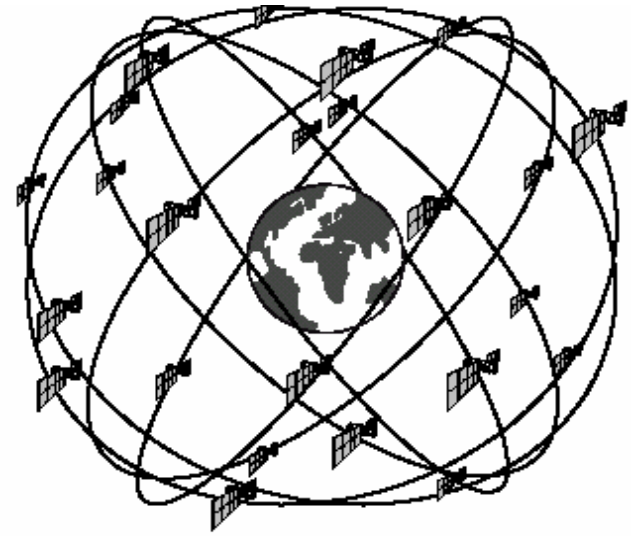


Figura 8: Constelación de satélites GPS

GPS es de uso gratuito para cualquier usuario. El mecanismo de comunicación utilizado es de una vía (del satélite al usuario). Se proporcionan dos tipos de servicios:

- Servicio de posicionamiento preciso (Precise Positioning Service –PPS): permite un posicionamiento con precisión de 22m en la horizontal y 27.7m en la vertical. Es un servicio cifrado que sólo puede ser accedido por las fuerzas armadas estadounidenses y los miembros de la OTAN.
- Servicio de posicionamiento estándar (Estándar Positioning Service – SPS): para uso civil, tiene una precisión de 100m en la horizontal y 156m en la vertical.

Los satélites envían señales constantes de aproximadamente 20W. Utilizan dos frecuencias: L1 (1575.42 MHz) para PPS y SPS, y L2(1227.6 MHz) exclusivo para PPS.

Las variantes del GPS son Differential GPS y el Wide Augmentaction System. La primera utiliza estaciones base en tierra que efectúan cálculos utilizando los satélites y efectuando correcciones utilizando sus propias posiciones (conocidas) y enviándolas al dispositivo del usuario que utiliza esas correcciones y le permite un posicionamiento más preciso. El WAS es similar al DGPS sin embargo la transmisión de las correcciones son enviadas al usuario a través de satélites geoestacionarios.

Otro sistema de este tipo es el GLONASS Ruso, el cual por problemas financieros no es global y el sistema GALILEO de la Unión Europea que empezará a funcionar en el año 2008 y ofrecerá servicios similares a los ofrecidos por GPS.

2.3.1.2 Sistemas de posicionamiento interno

El otro tipo de sistemas son los de posicionamiento interno. Estos sistemas permiten efectuar localizaciones de objetos dentro de edificios, donde los sistemas basados en satélites fallan. Existen muchas formas de efectuar el posicionamiento interno, entre las cuales están:

- Posicionamiento por transmisores infrarrojos.
- Posicionamiento por transmisores de radio.

- Posicionamiento basado en señales de ultrasonido.
- Sistemas basados en video.

2.3.1.3 Posicionamiento en redes GSM

El Global System for Mobile Comunnication (GSM) es un estándar servicios de telefonía celular usado en más de 190 países. Es posible lograr un posicionamiento simple para redes GSM, la cual conoce siempre en que celda un teléfono móvil está registrado. Cualquier teléfono que entre en un área determinada genera un registro en una base de datos denominada VLR (Visitor Locator Register). Esa información es transmitida a una base de datos central, el HLR (Home Locator Register) desde donde puede ser recuperada. Cada operador de telefonía tiene su propio HLR. Con esta base de datos, se pueden ofrecer servicios simples, que permiten localizar a los usuarios con la precisión del tamaño de una celda.

La Información basada en localización puede ser obtenida por señales de radio enviadas desde estaciones base. Las estaciones base pueden transmitir esa información a través del llamado Canal de Transmisión de Celdas (Cell Broadcast Channel- CBC). En este caso el teléfono debe leer pequeños piezas de dato que contienen información de localización sobre diferentes servicios.

La resolución de este posicionamiento es inadecuada para muchos servicios. El radio de cada celda puede variar entre menos de 1Km en el centro urbano y 35Km en la periferia. La exactitud en el posicionamiento es inversamente proporcional al tamaño de la celda utilizada.

Ericsson® desarrolló un sistema llamado Sistema de Posicionamiento Móvil (Mobile Positioning System – MPS), el cual permite un posicionamiento más exacto en celdas grandes. El MPS utiliza el sistema GSM y solo requiere de modificaciones mínimas en la infraestructura de comunicación. El MPS puede ser mejorado con la ayuda del GPS, aunque este no es un requerimiento.
Para efectuar los cálculos de posicionamiento, MPS utiliza los siguientes mecanismos:

- Cell of Global Identity (Celda de Identidad Global): este mecanismo utiliza la identificación de una celda para determinar la posición de un usuario. Este método no es muy preciso y es utilizado solamente cuando no se dispone de uno mejor.

- Segmentos de antenas: las estaciones base, dividen los 360° en segmentos (usualmente dos, tres o cuatro). De esta forma la estación base puede dividir la localización del usuario en un segmento angular de 180°, 120°, 90°.

- Avance en la medida del tiempo (Timing Advance): las estaciones base y los terminales móviles utilizan ciertos periodos de tiempo para comunicarse, el mecanismo toma en cuenta el tiempo de viaje de una señal entre la terminal y la estación base. La distancia a la estación base es medida en “pasos” de aproximadamente 555 m. Esta técnica puede ser combinada con segmentos de antena para mejorar la precisión.

- Uplink Time of Arrival (UL-TOA): esta técnica mejora el cálculo siempre y cuando el aparato móvil se encuentre en el alcance de por lo menos 4 estaciones base. Midiendo el tiempo de viaje de una señal desde el aparato a las estaciones base se puede medir su posición con una precisión de 50m a 150m.

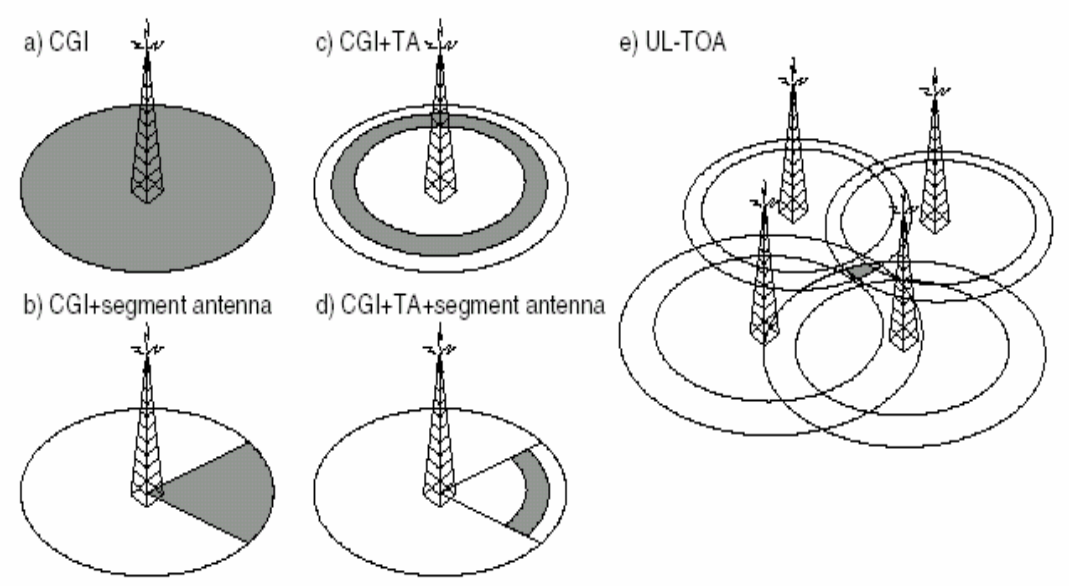


Figura 9: Mobile Positioning System

Todos los datos de localización determinados se envían a un centro de posicionamiento móvil (Mobile Positioning Center- MPC), donde son almacenados de forma temporal y pueden ser accedidos por terceras partes, p. Ej. En Internet después de un proceso de autenticación.

2.3.2 Capa middeware

Un middleware es definido como un conjunto de servicios que facilitan el desarrollo y puesta en marcha de aplicaciones distribuidas en entornos heterogéneos. Consiste de un conjunto de servicios que exponen interfaces, un modelo de programación y un modelo de interacción para el desarrollador de la aplicación. En el contexto de servicios basados en localización, esto se refiere a los servicios, abstracciones y modelos que implementan la coordinación, correlación y diseminación de la información de usuarios móviles.

Un middleware para SBL debe poseer las siguientes características:

- Soporte para servicios basados en modelos pull y push.
- Perfiles directos o indirectos: es decir relacionar la petición con información del perfil del solicitante, obtenida directamente de una suscripción, por observación de patrones o de terceras partes.
- Escenarios de interacción: soporte para diferentes formas de interacción:
  - Consumidor y proveedor estacionario.
  - Consumidor estacionario, proveedor móvil.
  - Proveedor estacionario, consumidor móvil.
  - Proveedor y consumidor móvil.
- Fuente de la información de localización: posibilidad de obtener la localización del cliente, del mismo aparato, de la red o de terceras partes.
- Exactitud de la información de localización: soporte para exactitud de localizaciones diversas.
- Interacción basada en estados: una interacción está basada en estados si el servicio mantiene un registro de múltiples peticiones del mismo cliente y no basada en estados si tal registro no existe y cada petición es procesada

independientemente de solicitudes anteriores. En aplicaciones de seguimiento puede ser muy importante mantener información de posiciones anteriores de los objetos para proporcionar un esquema de predicción de comportamiento futuro.

- Tipo de fuentes de información: los SBL están basados en la correlación efectiva de información que se origina de diferentes fuentes. Existe una distinción entre fuentes estáticas y dinámicas. Las fuentes de información estáticas se refieren a bases de datos de información geográfica (tipos diferentes de mapas, atracciones, edificios, etc.). Las fuentes dinámicas ofrecen información sobre condiciones cambiantes (tráfico, clima, etc.).

Requerimientos impuestos en la plataforma middleware

Partiendo de las características anteriores y debido al número potencialmente alto de usuarios de SBL, un middleware debe ser capaz de:

- Poder manejar cualquier tipo de información de movilidad requerida por todas las aplicaciones.
- Manejar cambios en la topología de red subyacente.
- Poder manejar millones de consumidores de información y por lo tanto millones de subscripciones.
- Manejar un número potencialmente grande de proveedores de información.
- Enviar notificaciones a miles de consumidores simultáneamente como resultado de manejar gran cantidad de contenido enviado al sistema.
- Manejar perfiles de usuario altamente volátiles (actualizaciones, inserciones, borrados)
- Procesar diferentes formatos de contenido.
- Soportar diferentes canales de notificación (SMS, WAP, protocolos de Internet, etc.).
- Proporcionar alta fidelidad aún después de fallas en alguno de sus subsistemas.
- Manejo de cuentas, soporte para cuentas por servicio brindado para cada proveedor y consumidor.
- Soporte para funciones de seguridad (autenticación del proveedor y consumidor), distribución segura de contenido.
- Soporte a políticas de privacidad, permitir que el usuario seleccione aquellas aplicaciones que pueden utilizar su información de localización.
- Soportar altas tasas de transmisión de información por cada usuario.

Estos requerimientos son muy variados, y afectan a todas las capas del middleware.

Arquitectura

Un middleware para SBL, es puesto en marcha en la red operador del servicio móvil o en por el proveedor de la aplicación. Conecta a las terminales móviles de los clientes con la Internet, proveedores de aplicaciones y otros operadores de redes para ofrecer un portal único de aplicaciones basadas en localización compuesto de muchos servicios personalizables. El middleware integra la infraestructura de red, que incluye servidores de localización, pasarelas WAP, portales de suscripción, sistemas operacionales (de cuentas, de cobro, etc.).

La siguiente figura describe en ejemplo una arquitectura completa que muestra a los usuarios, operadores de red, aplicaciones de terceros además de los sistemas mencionados.

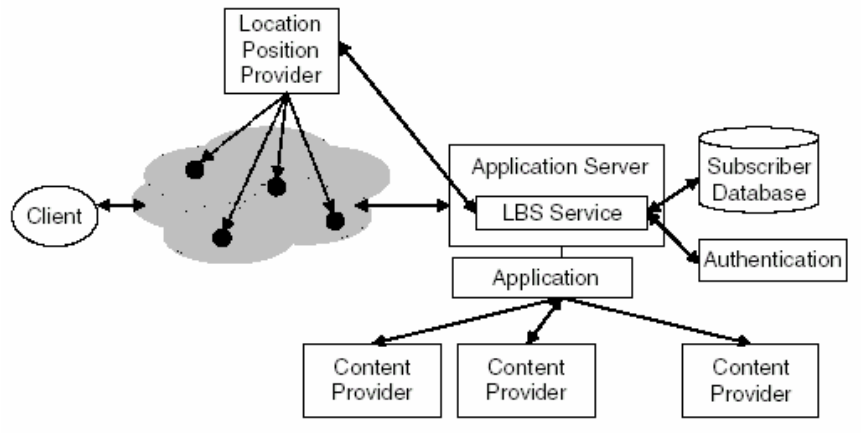


Figura 10: Integración del middleware en la infraestructura del proveedor

Un middleware difiere en el tipo de servicios ofrecidos a sus usuarios (suscriptor, operador de red y proveedor de aplicación). Para los suscriptores, un control completo de sus perfiles, información de localización, así como información contextual son funcionalidades muy importantes. Debe ser posible habilitar o inhabilitar esta información para aplicaciones específicas. Para los operadores de red y proveedores de aplicación entre los servicios ofrecidos están: esquemas de ingresos, acceso a información de suscriptores, etc.

Los servicios ofrecidos por el middleware a las aplicaciones son completamente transparentes.

Debido a que un middleware depende de los servicios ofrecidos, actualmente existen muchas arquitecturas, sin ninguna dominante por completo. De hecho no hay un modelo de referencia que indique que componentes y que interacción entre estos se debe dar para formar el sistema, pero sí hay estándares propuestos para los servicios esenciales que el middleware debe proporcionar a las aplicaciones.

Estándares para middleware

Existen estándares propuestos para la interacción entre diferentes sistemas middleware, no estándares para el diseño de arquitecturas.

En la especificación de esos estándares, dos organizaciones juegan un papel muy importante: el Open Mobile Alliance y el Open Geospatial Consortium.

El OMA es una organización que trabaja en especificaciones para toda la industria inalámbrica, proporcionando estándares para los diferentes tipos de aplicaciones y servicios que esta puede proporcionar, entre estos se encuentran SBL. En este sentido, el aporte de este grupo a la interoperabilidad de aplicaciones SBL es el Mobile Location Protocol (MLP- protocolo de localización móvil), el cual está basado en XML y define las interfaces comunes que facilitan el intercambio de información de localización entre las aplicaciones SBL y la redes representadas por servidores de localización.

Por otro lado, el área principal de interés del OGC es SBL. La mayor parte de su trabajo involucra cuestiones de interoperabilidad entre software geoespacial. Los esfuerzos de este organismo junto al de otros han dado como resultado la plataforma OpenLS (Open Location Services).

La plataforma OpenLS

Los miembros del OGC han desarrollado de forma conjunta el servidor GeoMobility, el cual esencialmente es un conjunto de especificaciones para interfaces abiertas de middleware SBL.

Este servidor es una plataforma abierta que comprende todos los servicios esenciales (core services) que un middleware SBL debe proporcionar. La meta es permitir la comunicación de información de localización, rutas, tipos de servicios, etc. en múltiples plataformas tecnológicas, dominios de aplicaciones, clases de productos, infraestructuras de red, etc.

Su primer objetivo es definir el acceso a los servicios esenciales, y tipos abstractos de datos (Abstract Data Types – ADT) que componen el servidor.

Estos servicios son:

- Determinación de rutas: determinar información de rutas entre localizaciones.
- Utilidades de localización:
  - Geocodificador: transforma direcciones textuales en coordenadas x, y.
  - Geocodificador inverso: transforma coordenadas x, y en direcciones textuales.
- Presentación: Determina la forma en que se muestra la información proporcionada por las aplicaciones (mapas, texto, audio, etc.).
- Location Gateway: Obtiene la posición del dispositivo móvil “desde la red”.
- Servicios de directorio: Proporciona búsquedas de point of interest (POI-sitios de interés).

Además de definir las interfaces para los servicios esenciales para SBL, la especificación OpenLS, define los ADTs. Un ADT es la construcción básica de información utilizada por el servidor GeoMobility y sus servicios esenciales asociados. Los ADTs son definidos como esquemas de aplicación en lenguaje XML para servicios basados en localización (XLS).

Los ADTs para OpenLS son:

- Rutas: meta datos relacionados a una ruta. Esta compuesto por dos ADT:
  - Sumario de una ruta: contiene todas las características de una ruta, como su punto de inicio, puntos o lugares intermedios, tipo(s) de transporte, distancia total, tiempo de viaje y límites geográficos.
  - Geometría de ruta: contiene una lista de puntos a lo largo de la ruta, ordenados según el plan de viaje. La geometría incluye la posición de todos los nodos a lo largo de la ruta incluyendo lugares intermedios. Estos dos ADTs son generados por el servicio de rutas y presentados de alguna forma al suscriptor por el servicio de presentación o utilizados directamente por la aplicación para guiar al usuario a su destino.
- Lista de instrucciones de ruta: proporciona una guía de navegación paso a paso para una ruta. Este ADT contiene una lista de instrucciones de viaje a lo largo de una ruta, ordenados de acuerdo a su ocurrencia. Este ADT es generado por el servicio de ruta y mostrado al suscriptor por el servicio de presentación.
- Localización: es un tipo de ADT abstracto y por lo tanto extensible para cualquier tipo de expresión de localización que puede ser utilizado en OpenLS para especificar la localización de un punto o suscriptor. Es la raíz de un árbol semántico que incluye los ADTs Punto, Posición, Dirección y POI como sus subtipos.

- Posición: contiene cualquier posición observada o calculada. Esta compuesta de una posición geográfica e información sobre la calidad (margen de error) de esta. Es el tipo de dato primario utilizado por el servicio Gateway. Este ADT puede convertirse en información semántica sobre los elementos de una localización, forma, calidad de posición definidos en MLP.
- Área de interés: este ADT contiene un área de interés representada por un circulo, cuadrado o polígono geográfico. Es utilizado como un parámetro de búsqueda y puede ser mostrado a un suscriptor.
- Punto de interés: representa la localización donde se puede encontrar un lugar, un producto o servicio. Este ADT es un lugar o entidad con una posición fija que puede ser utilizado como un punto de referencia o destino en un servicio OpenLS. POI es la respuesta primaria de un servicio de directorio. Contiene un nombre, tipo, categoría, dirección, teléfono y otra información sobre el lugar, producto o servicio.
- Dirección: este ADT contiene información de direcciones sobre un lugar geográfico. Se utilizan como medio para referenciar lugares, residencias o edificios de todos los tipos.
- Mapa: el ADT de mapas contiene información sobre un mapa que resulta de la operación de representación gráfica efectuada por el servicio de presentación. Puede ser utilizada como base para otros servicios de presentación. Está compuesta de información de contenido (formato, ancho y altura) e información de contexto (limites del mapa, punto central y escala).

La siguiente figura ilustra un modelo de middleware basado en OpenLS:

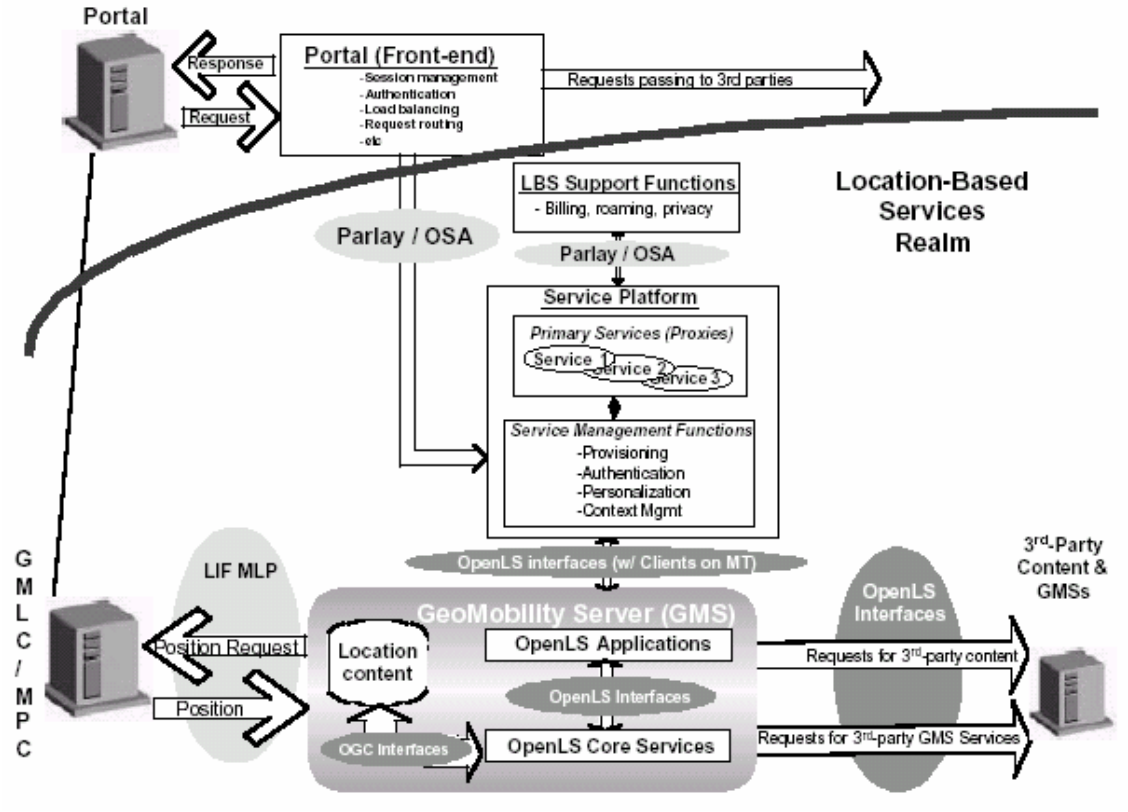


Figura 11: La plataforma OpenLS

La figura anterior muestra de forma detallada la interacción entre los subsistemas que componen un middleware basado en OpenLS.

Una ilustración de una típica interacción ante petición de servicio se proporciona en la siguiente figura:

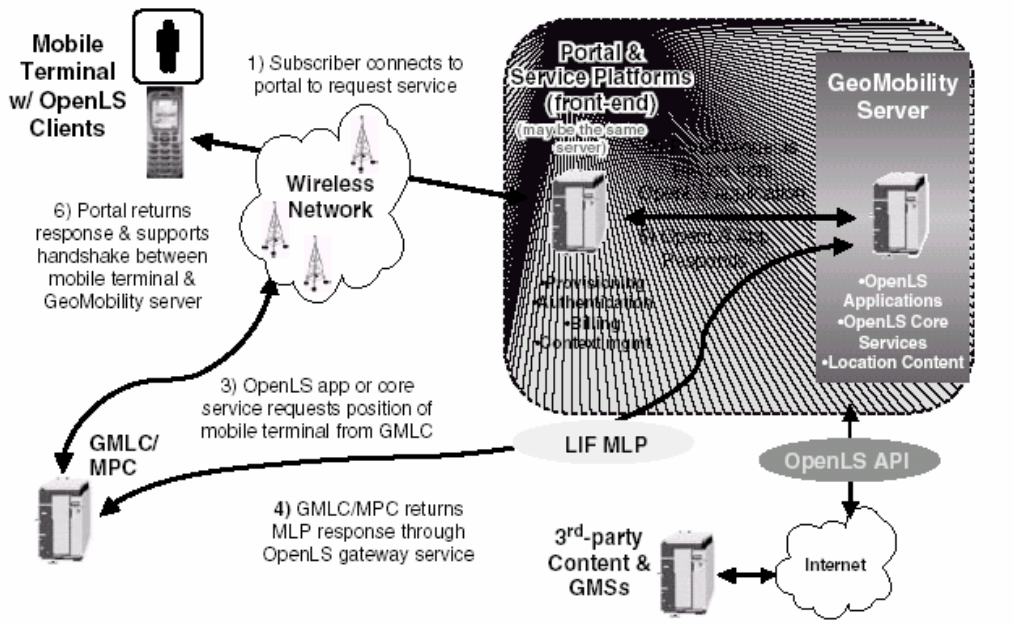


Figura 12: Interacción entre las diferentes partes de la infraestructura

1. Un usuario móvil solicita un servicio a un portal.
2. El portal pasa la petición al servidor OpenLS que contiene el servicio adecuado.
3. La aplicación solicita la posición del usuario a través del servicio Gateway a un servidor de localización.
4. El servidor de localización pide la información de localización a la red. Al obtenerla responde con una respuesta MLP al servicio Gateway y este la envía a la aplicación.
5. La aplicación utiliza la información para solicitar información a un proveedor de contenido local o de terceras partes.
6. Se envía una respuesta al cliente móvil a través del portal.

Es importante mencionar que la autenticación de un "usuario" puede efectuarse en varias fases. El usuario móvil puede autenticarse ante el portal, mientras que la plataforma puede necesitar autenticarse ante sí misma (un servicio se autentifica ante otro) o ante proveedores externos.

El portal puede mantener información de contexto para cada usuario, perfiles y otros parámetros útiles para la personalización del servicio.

2.3.3 Capa de Aplicación

La capa de aplicación está compuesta por todos aquellos servicios que basan su funcionamiento en los servicios esenciales de un middleware, sea este propietario o basado en la especificación OpenLS.

Es importante indicar que muchas veces los mismos servicios esenciales podrían formar parte de la capa de aplicación, por lo que no siempre existe una línea divisoria visible entre las aplicaciones y aquellos servicios brindados por el middleware. Un ejemplo de esto es el servicio de presentación que es cliente del resto de servicios y ocasionalmente "conecta directamente" al servidor con el aparato móvil del usuario.

2.4 Soporte J2ME a los servicios basados en localización

Java 2 Microedition proporciona soporte para el desarrollo de aplicaciones basadas en localización en aparatos móviles. Esto lo hace a través del jsr179: "Location API for J2ME v1.0". Este paquete puede ser utilizado con cualquier perfil J2ME, sin embargo el uso con CLDC1.0 es imposible ya que este no proporciona soporte para datos de coma flotante utilizados por la especificación.

El paquete javax.microedition.location contiene clases básicas que permiten pedir y obtener información de localización. Está diseñado para ser un API compacta y genérica que produce información sobre la posición geográfica actual de la terminal a las aplicaciones Java independientemente del método de posicionamiento utilizado y del tipo de red en que la aplicación se ejecute.

Este paquete también incluye una base de datos de Landmarks. Un Landmark es una ubicación especifica que esta asociada con un nombre representativo de cara al usuario, por ejemplo la casa del usuario, su oficina, etc. El usuario puede almacenar ubicaciones usadas comúnmente en esta base de datos. Está base de datos es compartida con todas las aplicaciones Java en el dispositivo y puede que la implementación permita compartirla con aplicaciones nativas del aparato.

En esta API al menos se garantiza que (independientemente del método de localización utilizado) se:

- Proporcionará coordenadas de latitud, longitud y su margen de error.
- Proporcionará información del tiempo utilizado para efectuar las mediciones.
- Se proporcionará al menos una base de datos para almacenar Landmarks.

Aquella funcionalidad que depende del método utilizado es:

- Proporcionar información de altitud y su margen de error.
- Proporcionar información sobre dirección y velocidad.
- Proporcionar información textual sobre dirección relacionada a la posición.
- Proporcionar notificación de eventos de notificación sobre proximidad a Landmaks.

Seguridad

Algunos de los métodos provocan un SecurityException si no se poseen los permisos necesarios para efectuar las operaciones. Esto debe ser reforzado por la capa de seguridad de cada implementación. Para obtener información adicional sobre los métodos que están protegidos, consúltese la documentación oficial.

API de localización

El punto de inicio en el uso de esta API es la clase javax.microedition.location.LocationProvider. Esta clase abstracta representa modulo

que proporciona información de localización, que genera objetos Location. Su declaración es la siguiente.

```
public abstract class LocationProvider {…}
```

Todas aquellas aplicaciones que quieran utilizar información de localización necesitan obtener instancias de una clase derivada de esta (desconocida) a través del método getInstance(), declarado así:

```
public static LocationProvider getInstance(Criteria c) throws LocationException
```

Este método es utilizado para obtener una implementación real de la clase, la cual está basada en el criterio establecido como parámetro. La implementación escoge un proveedor que se adecue mejor a lo establecido. Si no se puede crear un LocationProvider que satisfaga lo establecido, pero existen otros proveedores que no cumplen con el criterio se retorna null. Se provoca un LocationException si todos los proveedores de localización están fuera de servicio.

La instancia retornada (si es que alguna) está en estado disponible o fuera de servicio temporalmente. Si el parámetro proporcionado es null la implementación retorna el LocationProvider menos restrictivo y que no está fuera de servicio temporalmente.

Una vez obtenida la instancia LocationProvider, la aplicación puede obtener la posición del dispositivo llamando al método getLocation():

```
public abstract Location getLocation( int tiempoEspera) throws LocationException,
                                                              java.lang.InterruptedException
```

Este método obtiene la localización de acuerdo a los criterios establecidos en getInstance(). Si no se puede obtener un resultado o este no se puede obtener en el tiempoEspera (en segundos) especificado se provoca un LocationException. Si el proveedor está temporalmente fuera de servicio, se espera la cantidad de tiempo indicada en el parámetro. Si el parámetro es –1 se indica utilizar el tiempo por defecto establecido por la implementación.

Los otros métodos de esta clase son:

```
public abstract int getState()
```

Este método obtiene el estado del proveedor, los posibles valores de retorno son:

- LocationProvider.AVAILABLE: el proveedor esta disponible.
- LocationProvider.OUT_OF_SERVICE: el proveedor esta fuera de servicio.
- LocationProvider.TEMPORARILY_UNAVAILABLE: el proveedor no está disponible temporalmente.

```
public static Location getLastKnowLocation()
```

Retorna el objeto Location que representa la última ubicación conocida. Más adelante abordaremos esta clase.

Criterios de selección de proveedores de localización

Los criterios de selección de proveedores de localización en el dispositivo son seleccionados por la implementación en base a los valores de un objeto de esta clase. La implementación considera los diferentes campos que contiene este objeto para elegir el proveedor más adecuado, sin embargo no existe una prioridad específica de estos.

Para construir un objeto de este tipo con valores por defecto se dispone del siguiente método:

public Criteria()

Para establecer los valores para objetos Criteria se dispone de los siguientes métodos (los métodos is* y get* permiten recuperar los valores efectuados en métodos set*):

public void setAddressInfoRequired(boolean valor)
public boolean isAddressInfoRequired()

Establece que el proveedor debe ser capaz de determinar información textual.

public void setAltitudeRequired(boolean valor)
public boolean isAltitudeRequired()

Establece si el proveedor debe ser capaz de proporcionar información sobre altura. El parámetro por defecto es false.

public boolean setCostAllowed(boolean valor)
public boolean isAllowedToCost()

Establece si el proveedor seleccionado es gratuito o no. Si la implementación no puede determinar si el proveedor es gratuito, debe asumir que no lo es. Si el parámetro es establecido a false, debe retornar un proveedor por el que este seguro, el usuario no va a pagar. El parámetro por defecto es true.

public void setHorizontalAccuracy(int exactitud)
public int getHorizontalAccuracy()
public void setVerticalAccuracy(int exactitud)
public int getVerticalAccuracy()

Los métodos set*Accuracy anteriores establecen la exactitud horizontal y vertical respectivamente. Este valor está medido en metros. Los valores por defecto para ambos campos es Criteria.NO_REQUIREMENT que indica que el campo correspondiente no es un factor determinante.

public void setPreferredPowerConsumption(int criterio)
public int getPreferredPowerConsumption()

Este método indica la cantidad de energía que debería ser consumida a la hora de calcular la posición, o la cantidad de procesamiento que debe dedicarse al cálculo. El valor por defecto es Criteria.NO_REQUIREMENT; los otros son:

- Criteria.POWER_USAGE_LOW
- Criteria.POWER_USAGE_MEDIUM
- Criteria.POWER_USAGE_HIGH

Estos campos representan un consumo bajo, medio y alto respectivamente.

public void setPreferredResponseTime(int tiempo)
public int getPreferredResponseTime()
Este método permite indicar el tiempo de respuesta tolerable para la selección de un proveedor, es decir que la implementación debe seleccionar un proveedor que pueda producir la información en el periodo establecido. El valor por defecto es Criteria.NO_REQUIREMENT.

Location

Esta clase representa un conjunto básico de información de localización. Incluye coordenadas de tiempo, exactitud, velocidad, curso e información sobre el método de posicionamiento utilizado, además de una dirección textual (opcional).

Un objeto de este tipo puede ser "válido" o "no válido". Un objeto no válido no posee coordenadas geográficas válidas.
Los métodos de esta clase son los siguientes:

public float getCourse()

Este método retorna la dirección que tiene el terminal relativo al Norte o Float.NaN si no se conoce.

public int getLocationMethod()

Retorna información sobre el método de posicionamiento utilizado. El valor retornado es una combinación de bits del método utilizado, el tipo e información de asistencia. Si la información es desconocida todos los bits valen cero. Los campos que definen esta información son:

- Location.MTA_ASSISTED: el método es asistido por otras partes (Asistido por la terminal para la red, asistido por la red para el terminal).
- Location.MTA_UNASSISTED: el método no es asistido.
- Location.MTE_ANGLEOFARRIVAL: el método está basado en ángulos de llegada.
- Location.MTE_CELLID: el método es Cell-ID o en GSM CGI (Cell of Global Identity).
- Location.MTE_SATELLITE: se utilizan satélites, por ejemplo GPS.
- Location.MTE_SHORTRANGE: método de localización de área corta, por ejemplo basados en Bluetooth.
- Location.MTE_TIMEDIFFERENCE: método basado en la técnica Time of Difference.
- Location.MTE_TIMEOFARRIVAL: método basado en la técnica Time of Arrival.
- Location.MTY_NETWORKBASED: método basado en la red.
- Location.MTY_TERMINALBASED: método basado en el terminal.

Información sobre direcciones:

La clase Location proporciona información textual sobre la posición que representa, esto lo hace a través del método getAddressInfo()

public AddressInfo getAddressInfo()
Este método devuelve un objeto AddressInfo asociado al objeto Location. Si no está disponible retorna null.

AddressInfo mantiene información textual sobre una posición. Está información incluye cosas como calles, código postal, ciudad, etc.) . Se definen campos que pueden ser utilizados para obtenerla:

public String getField(int campo)

Se retorna el valor del campo especificado como parámetro o null si la información no está disponible. Entre los campos disponibles están:

- AddressInfo.CITY: utilizado para pueblos o ciudades.
- AddressInfo.COUNTRY: utilizado para nombres completos de países.
- AddressInfo.STREET: utilizado para denotar nombres de calles.
- AddressInfo.PHONENUMBER: utilizado para denotar números de teléfono en este lugar.
- Etc., etc...

Coordenadas de una localización

Las coordenadas de una posición se pueden obtener llamando al método getQualifiedCoordinates() de la clase Location:

public QualifiedCoordinates getQualifiedCoordinates()

Este método devuelve las coordenadas representadas por el objeto Location asociado en un objeto QualifiedCoordinates.

La clase QualifiedCoordinates está declarada así:

public class QualifiedCoordinates extends Coordinates{…}

A como indica la declaración, esta clase está derivada de la clase Coordinates, la cual representa una coordenada de tipo latitud- longitud- altitud. QualifiedCoordinates adicionalmente proporciona información sobre el grado de exactitud de la información proporcionada.

public QualifiedCoordinates(double latitud, double longitud, float altitud, float exactitudHorz, float exactitudVert)

Este método permite construir un objeto QualifiedCoordinates con la latitud, longitud, altitud especificada e información sobre la exactitud en la horizontal y vertical.

Para recuperar esta información se dispone de los siguientes métodos:

public double getLongitude()
public double getLatitude()
public float getAltitude()

Estos métodos devuelven la longitud, latitud en grados. El método getAltitude() devuelve la longitud en metros con respecto a WGS84 o Float.NaN.

float getHorizontalAccuracy()
float getVerticalAccuracy()

Estos métodos devuelven la exactitud horizontal y vertical respectivamente asociada a las coordenadas del objeto QualifiedCoordinates que los invoca.

Almacén de Landmarks

Como ya se ha mencionado, todo dispositivo que implemente el jsr179, debe proporcionar una base de datos de landmarks (ubicaciones que tienen asociado nombres representativos para el usuario), la cual es compartida por todas las aplicaciones del aparato.

Un almacén de este tipo es representado por una instancia de la clase LandmarkStore. Esta clase proporciona los métodos necesarios para almacenar, borrar y obtener objetos Landmarks de una forma consistente. Todo almacén posee un nombre que lo caracteriza. Para poder leer y escribir landmarks de un objeto de este tipo la aplicación debe poseer para cada operación los permisos necesarios de acuerdo a lo establecido por la especificación MIDP2.0.

Para obtener una instancia de un objeto LandmarkStore debe recurrirse al siguiente método:

public static getInstance(String nombreAlmacen)

Este método devuelve una instancia LandmarkStore cuyo nombre corresponde al parámetro especificado o null si no existe. Si se especifica null como parámetro el LandmarkStore por defecto es devuelto. Se provoca un SecurityException si la aplicación no tiene derecho a leer objetos LandmarkStore.

Si se desean crear LandmarkStore se debe utilizar el siguiente método:

public static void createLandmarkStore(String nombre) throws IOException

Este método crea un almacén con el nombre especificado, se provoca NullPointerException si el parámetro es null, IllegalArgumentException si es muy largo o ya existe un almacén con ese nombre, IOException si ocurre un error en la escritura, SecurityException si no se poseen los permisos adecuados y LandmarkException si la implementación no soporta la creación de nuevos LandmarkStore.

Para abrir otros almacenes es necesario conocer su nombre, para estos las aplicaciones pueden utilizar el siguiente método:

public static String[] listLandmarkStores() throws java.io.IOException

Este método lista en un array de objetos String con todos los nombres de almacenes que están disponibles, el nombre del almacén por defecto no es incluido. Si no existen

otros almacenes además del implícito, se retorna null. Se provoca un IOException si hay algún problema de lectura y un SecurityException si no se poseen los permisos de adecuados.

Para borrar un almacén se dispone del método:

public static void deleteLandMarkStore(String nombre) throws IOException

Se borra el almacén especificado en nombre. No se puede borrar el almacén por defecto de forma que este método lanza un NullPointerException si el parámetro es null. También se lanzan IOException si ocurre algún error en la escritura, SecurityException si no se poseen permisos adecuados y LandmarkException si la implementación no soporta el borrado de almacenes.

Contenido de un LandmarkStore

Todo LandmarkStore contiene objetos Landmark. Un objeto Landmark es el contenedor de información de una ubicación asociada con un nombre. Posee también una descripción, coordenadas y opcionalmente información sobre ubicación encapsulada en un objeto AddressInfo.

Para construir un objeto de este tipo se dispone de:

public Landmark(String nombre,String descripcion, QualifiedCoordinates coordenadas,
AddressInfo info)

Este método permite construir un objeto con el nombre, descripción, coordenadas y opcionalmente la información de dirección especificada. Se provoca un NullPointerException si nombre es null. El resto de parámetros pueden ser null.

Los métodos get* y set* se utilizan para establecer y obtener respectivamente, los campos especificados en el constructor.

Operaciones en un almacén de landmarks

Un almacén maneja todos sus objetos landmarks relacionados en categorías (p. Ej.: restaurantes, hoteles, etc.) . Es posible que un mismo landmark esté almacenado en muchas categorías o no pertenezca a ninguna en específico.

Para operar con las categorías de un objeto LandmarkStore se dispone de los siguientes métodos:

java.util.Enumeration getCategories()

Este método devuelve un objeto Enumeration que contiene los nombres de todas las categorías disponibles.

public void addCategory(String nombre) throws LandmarkException,IOException

Este método añade una nueva categoría cuyo nombre se especifica en el parámetro. El parámetro es una cadena de caracteres de cualquier longitud, sin embargo sólo se garantiza que se conservarán los primeros 32 caracteres. Se provoca un IllegalArgumentException si la categoría ya existe, NullPointerException si el parámetro es null; LandmarkException si el almacén no soporta añadir nuevos

recursos, IOException si ocurre algún error de escritura y SecurityException si no se poseen los permisos adecuados para escribir en el almacén.

Para borrar una categoría se utiliza el método:

public void deleteCategory(String nombre)

Este método borra la categoría especificada, se provoca NullPointerException si el parámetro es null, LandmarkException si el almacén no soporta borrados de categorías, IOException si ocurre un error de escritura, SecurityException si no se poseen los permisos adecuados.

Para añadir objetos Landmark se utiliza:

public void addLandmark(Landmark lm, String categoria)

El objeto lm pertenecerá a la categoría especificada o a ninguna si el parámetro categoría es null.

El objeto Landmark se añade a la categoría especificada. Se provoca SecurityException si la aplicación no está autorizada para añadir objetos Landmark, IllegalArgumentException si el objeto referenciado por lm tiene un campo nombre demasiado largo para la implementación o si la categoría es no null y no existe dentro del almacén; IOException si ocurre un error de escritura y NullPointerException si el parámetro lm es null.

Para borrar un objeto Landmark de un almacén se dispone de los siguientes métodos:

public void deleteLandmark(Landmark lm)
public void removeLandmarkFromCategory(Landmark lm, String categoria)

El primer método borra la instancia referenciada por lm de todas las categorías y luego del almacén. El segundo método borra el objeto lm de la categoría especificada pero no del almacén, si el objeto no pertenece a la categoría especificada la llamada es ignorada. Un landmark que solo pertenece la categoría especificada cambia su categoría a null cuando este método retorna. Ambos métodos provocan SecurityException, LandmarkException, IOException y NullPointerException si no se poseen los permisos adecuados, la instancia Landmark no pertenece al almacén especificado, ocurre un error de lectura/ escritura o cualquiera de los parámetros es null respectivamente.

Para obtener todos los objetos de un almacén se dispone de:

java.util.Enumeration getLandmarks()
java.util.Enumeration getLandmarks(String categoria, double latMinima, double latMaxima,
double longMinima, double longMaxima)
java.util.Enumeration getLandmarks(String categoria, String name)

La primera sobrecarga de este método devuelve un objeto Enumeration que contiene todos los objetos Landmark o null si no existe ninguno. La segunda versión devuelve una enumeración basado en la latitud y longitud mínima y máxima especificadas. La tercera sobrecarga devuelve una enumeración con los objetos Landmark que pertenezcan a la categoría y nombre especificados o null si ninguno, si ambos parámetros son null este método se comporta como el primero. Para todos los

métodos se provoca IOException si ocurre un error de lectura y SecurityException si la aplicación no tiene los permisos para leer del almacén.

Para actualizar un la información de un objeto Landmark:

public void updateLandmark(Landmark lm) throws IOException, LandmarkException

Este método actualiza la información de una instancia que pertenece al almacén. Se provoca un SecurityException si la aplicación no tiene los permisos para actualizar landmarks. LandmarkException si el objeto referenciado por lm no pertenece a este LandmarkStore, IOException si ocurre algún error de lectura/ escritura y NullPointerException si lm referencia null.

Para más detalles sobre este paquete, puede consultarse la documentación oficial.

Tercera parte:

Desarrollo de una aplicación J2ME.

El ejemplo a desarrollar es una simulación de un servicio de consulta de POI comerciales.

POI son las siglas de Point Of Interest (Lugar de interes). Un POI es cualquier punto geográfico asociado a un producto o servicio (p. Ej.: escuelas, aeropuertos, parques, etc.). Un servicio de este tipo usualmente permite realizar los siguientes tipos de búsquedas:

- **Búsqueda por nombre**: filtra aquellos puntos por nombre, el filtro puede incluir coincidencia exacta, un patrón, o una búsqueda *soundex*[2]. Puede también utilizar nombres primarios y nombres alternativos. Son muy adecuadas cuando el usuario conoce bien lo que está buscando.
- **Alrededor de un punto**: una búsqueda por nombre que filtra todos aquellos puntos ubicados dentro del radio de un punto. Ese punto puede ser la ubicación del usuario o cualquier otra ubicación especificada por este. El radio puede ser especificado por el usuario o manejado por la misma aplicación.
- **En una ruta:** más compleja que las anteriores, permite filtrar puntos ubicados a lo largo de una determinada ruta especificada por el usuario.
- **Dentro de una ciudad o región:** similar a las búsquedas alrededor de un punto, excepto de que la extensión no está determinada por una distancia sino por los límites del sitio donde se busca.
- **Búsqueda por categorías:** búsquedas en las que se filtra por la categoría de los puntos (restaurantes, hospitales, etc.). Las mismas variaciones a las búsquedas por nombre son aplicables a este tipo de búsquedas.

Muchas aplicaciones pueden combinar ambos tipos de búsqueda (nombre y categoría) así como sus variaciones o añadir otras, como por ejemplo buscar todos aquellos lugares que no estén en una determinada área (quizás el usuario ha tenido malas experiencias utilizando sus servicios).

La siguiente figura ilustra el funcionamiento del servicio a desarrollar:

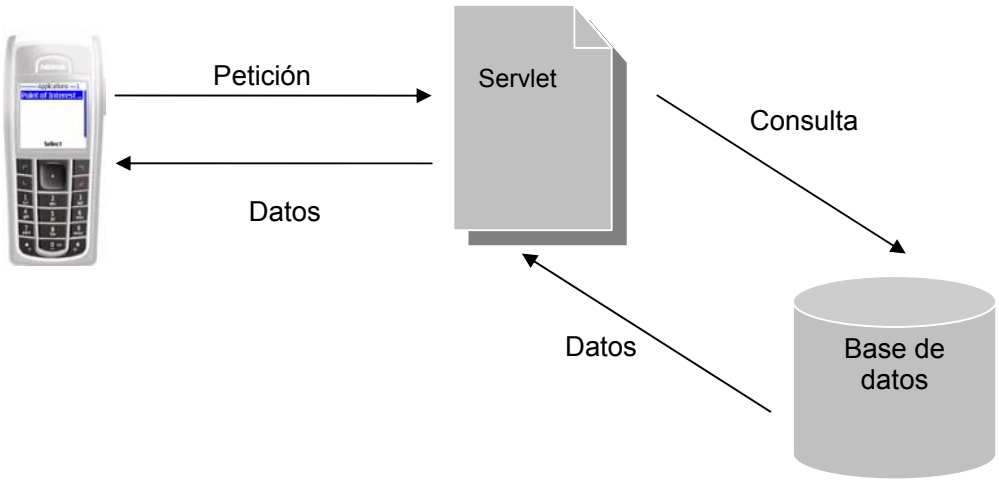


Figura 13: Esquema de funcionamiento de la aplicación de ejemplo.

[2] Puede encontrar más sobre soundex en:
http://www.nara.gov/genealogy/soundex/soundex.html

El cliente (nuestra aplicación) envía al servicio de POI (representado por un Servlet Java) una petición conteniendo el criterio y todos los parámetros relacionados con una consulta. El servlet procesa la petición y extrae los datos del proveedor de contenido, representado por una base de datos; al obtenerlos, los retorna al servlet el cual los prepara y envía a nuestro cliente.

El cliente es una aplicación J2ME llamada **POIExplorer** que utiliza la configuración CLDC1.1 y el perfil MIDP2.0, además de la API de localización (JSR179). Para nuestro ejemplo, el usuario podrá efectuar los siguientes tipos de búsquedas:

- Por nombre utilizando su ubicación actual.
- Por nombre en la ciudad en que se encuentra ubicado, obtenida a partir de su ubicación actual.
- Por nombre utilizando el nombre de la ciudad que el usuario proporcione.
- Por categoría utilizando los mismos modificadores que las búsquedas por nombre.

En el caso de las búsquedas por nombre se le permitirá seleccionar si estas son exactas o no. Para las búsquedas por categorías estas no son exactas independientemente de lo que el usuario haya seleccionado.

También se le permitirá almacenar los resultados de una búsqueda para su posterior visualización. No se le permitirá editar ninguno de los registros. Además se le permitirá almacenar ciertos parámetros por defecto, relacionados con sus búsquedas. Estos parámetros podrán ser modificados en cada búsqueda si el usuario lo desea.

El diagrama de clases se muestra en la siguiente figura:

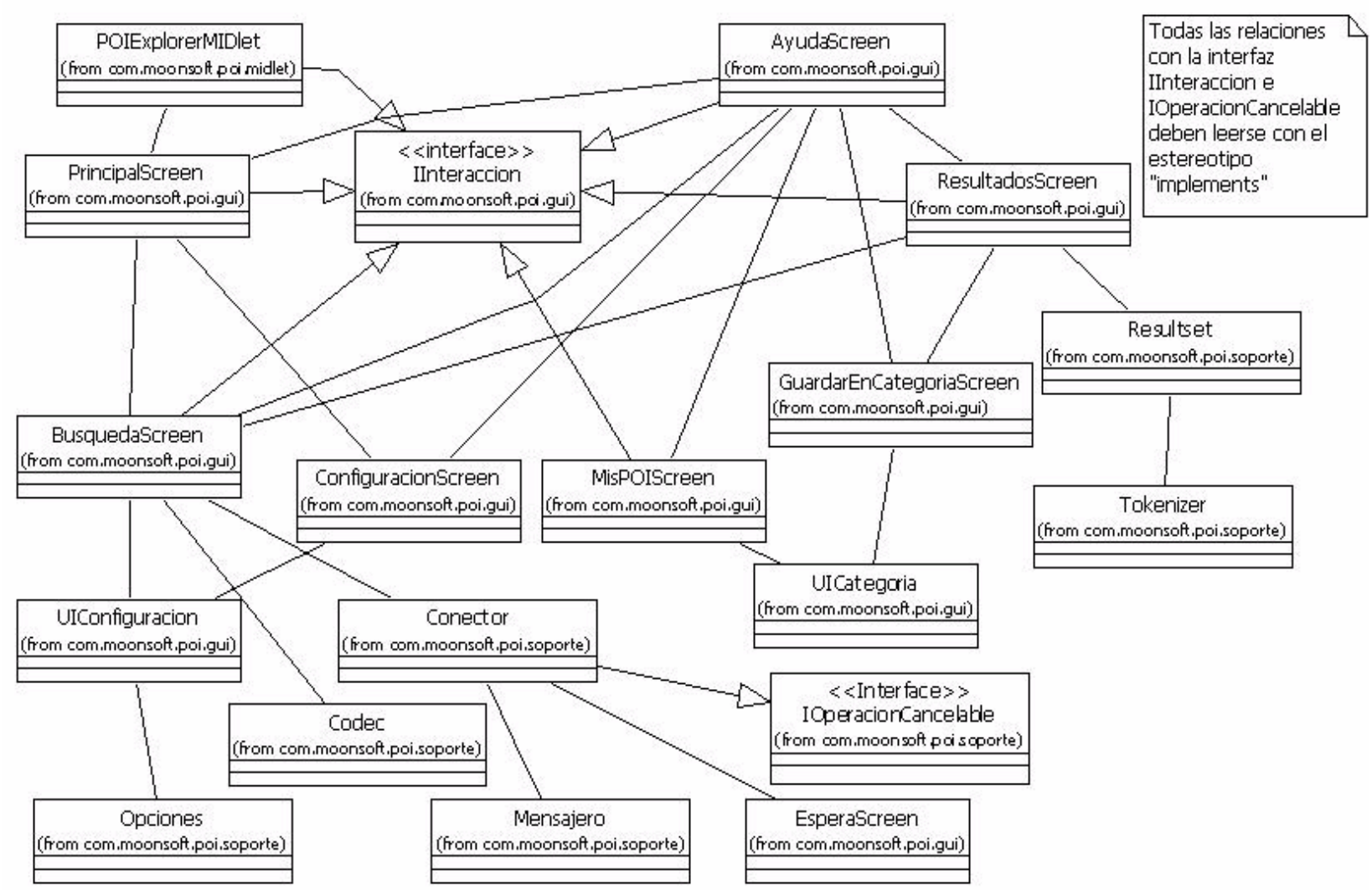


Figura 14: Diagrama de clases del cliente.

El código completo de estas clases se puede encontrar en el CD que acompaña a este documento. A continuación se describirá brevemente la funcionalidad de las clases más importantes:

Paquete com.moonsoft.poi.midlet

Este paquete contiene la única clase que representa el objeto aplicación. **POIExplorerMIDlet** maneja la interacción entre el entorno J2ME y nuestra aplicación, aunque no está relacionada con la interfaz gráfica implementa la interfaz IInteraccion y su método mostrar, la razón de implementar esta interfaz en esta clase es que se realiza una verificación sobre las capacidades de localización del dispositivo al iniciar el programa. Si este no contiene alguna implementación del jsr179 se provoca una excepción que es atrapada por el midlet el cual muestra el error al usuario, y posteriormente se llama al método mostrar de nuestro MIDlet para provocar su salida.

Paquete com.moonsoft.poi.gui

Este paquete contiene todas aquellas clases que están relacionadas a la interfaz de usuario.

IInteraccion

Esta interfaz es utilizada principalmente por las clases relacionadas con la interfaz de usuario. Los campos utilizados por todas las clases son los comandos “Atrás” y “Ayuda”, ya que la mayoría de las pantallas poseen al menos uno de ellos. Otros como los campos OPERACIÓN_* están relacionadas con las pantallas que permiten manipular registros. Esta interfaz declara también métodos que deben proporcionar todas las clases que la implementen:

- mostrarAyuda invocado para proporcionar ayuda al usuario para una determinada pantalla.
- mostrarMensaje invocado para proporcionar información al usuario sobre algún suceso.
- mostrar invocado para forzar a hacer visible una determinada pantalla.

Todas las clases *Screen de este paquete y la clase POIExplorerMIDlet implementan esta interfaz.

PrincipalScreen

Esta clase derivada de la clase javax.microedition.lcdui.Screen, proporciona un menú con las principales opciones de la aplicación. En el constructor de esta clase se verifica si el aparato contiene una implementación del jsr179.

BusquedaScreen

Esta clase descendiente de javax.microedition.lcdui.Form representa la pantalla donde el usuario puede efectuar las búsquedas de POI. Utiliza además la interfaz proporcionada por la clase UICategoria (descrita posteriormente), esto para permitir al usuario cambiar ciertos parámetros para una determinada búsqueda sin necesidad de modificar sus preferencias globales. Esta clase también crea un objeto Conector que envía una consulta al servidor. Para estandarizar el envió de la información esta clase invoca también a los métodos de la clase Codec.

MisPOIScreen

Esta clase representa la pantalla que permite al usuario visualizar registros de POI almacenados en su aparato. Utiliza la interfaz de usuario proporcionada por la clase UICategoria para obtener la información sobre las categorías de los POI almacenados. También se vale de los servicios de la clase ResultadosScreen para mostrar los POI relacionados con una determinada categoría.

ConfiguracionScreen

Esta clase representa la pantalla que permite al usuario modificar sus opciones de búsqueda. Esta clase se vale totalmente de la interfaz proporcionada por la clase UIConfiguracion.

ResultadosScreen

Esta clase representa la pantalla que permite al usuario visualizar la información y efectuar ciertas operaciones sobre ellos.

EsperaScreen

Esta clase representa una pantalla que visualiza un medidor o Gauge indicando el progreso de una operación que puede tardar cierto tiempo en completarse.

UIConfiguracion

Esta clase proporciona interfaz de usuario de la que se sirven las clases ConfiguracionScreen y BusquedaScreen. Se sirve de la funcionalidad proporcionada por la clase Opciones para brindar las operaciones de carga y salvaguarda de las preferencias del usuario.

UICategoria

Esta clase proporciona interfaz de usuario útil a las clases MisPOIScreen y GuardarEnCategoriaScreen. Interactúa en parte con el sistema de LandmarkStores del dispositivo.

Paquete com.moonsoft.poi.soporte

Este paquete contiene todas las clases que proporcionan la funcionalidad necesaria para que la interfaz de usuario pueda mostrar la información.

Opciones

Esta clase maneja todo lo relacionado con el almacenamiento y recuperación de las preferencias de búsqueda del usuario. Se encarga de la interacción con el RMS del dispositivo.

ResultSet

Esta clase maneja el almacenamiento en memoria de los registros obtenidos del aparato o de resultados de una búsqueda. Los servicios que proporciona son utilizados por la clase **ResultadosScreen.**

Mensajero

Esta clase proporciona los servicios de conexión externo de la aplicación, es la encargada de enviar petición y recibir respuestas.

El servicio

El servicio en el ejemplo está compuesto por un Servlet y un archivo de base de datos en Microsoft Access. La base de datos simula una fuente de POI de algún proveedor y el Servlet simula la aplicación que extrae la información de la fuente de POI y se la entrega al usuario.

Los datos devueltos representan sólo una fracción de los datos que podría tener un POI. En este caso solo se utilizan:

- El nombre del POI.
- La dirección del POI.
- El número de teléfono.

El servlet envía una respuesta al cliente utilizando el siguiente formato:

Para algún error en la consulta o cualquier condición que no devuelva registros:

ERROR\n<Mensaje indicando la causa del error>

Para una consulta con éxito:

OK\n<Nombre POI>: <Dirección POI>:<Teléfono POI>\n<Nombre POI>: <Dirección POI>:<Teléfono POI>\n….

La base de datos es una simulación de una fuente de datos de POI. En este caso la fuente podría estar en diferentes formatos: Un mapa digital, una base de datos geoespacial, un mapa digital con referencias a una base de datos relacional; o una base de datos orientada a objetos.

Por simplicidad se ha escogido utilizar una base de datos relacional, aunque este no es el formato más adecuado para representar datos que están relacionados geoespacialmente.

Para efectuar el diseño de la base de datos se ha tomado como ejemplo la forma en que en nuestro país se utiliza para referenciar determinados lugares. Por ejemplo si queremos dar una dirección utilizamos un punto de referencia y a partir de eso nos movemos cierta distancia en dirección a algún punto cardinal, p. Ej.: si queremos referirnos al edificio central de la UNAN León, podríamos decir: “del parque central 1 ½ c. al norte” o “costado norte de la iglesia la merced” o “iglesia la recolección 1 c. al sur”, etc.

En nuestra base de datos utilizamos la misma lógica, si queremos referirnos a un POI hablaremos en términos de su relación a un landmark. Un landmark en este caso es un lugar conocido por todo el mundo (casi siempre). Dado que un POI puede estar relacionado con diferentes landmarks hay varias posibles direcciones para ese POI. Cuando al consultar la base de datos se toma en cuenta la localización del usuario y existen varias posibles direcciones, se utiliza la dirección que refleje la distancia más cercana al usuario efectuando un simple calculo con la formula de la distancia. Si no hay varias posibilidades, se toma la única que exista siempre y cuando llene los

requerimientos de la consulta. En este caso existe una tabla de landmarks que contiene aquellos puntos de la ciudad (de León) más conocidos; estos principalmente son iglesias pero es posible que aparezcan parques o monumentos.

El problema con este tipo de soluciones es: ¿qué pasaría si los mismos landmarks fueran sujetos de consulta?, una solución sería duplicar la información de la tabla de landmarks en la tabla de POI (aunque quizás no la más optima pero la más barata desde mi punto de vista); aunque esto no aportaría mucho a la calidad de las consultas. La otra sería crearse un mapa digital que contará con referencias a la base de datos relacional, esta sería una de las soluciones más adecuadas, pero requeriría una inversión en la creación de los datos y en otra en la obtención de un motor de análisis geoespacial para efectuar las consultas. Una última opción sería crear los datos e integrarlos a un motor de análisis geoespacial en Internet como Google Maps o Google Earth.

Cada POI de la base de datos tiene asociado una o más palabras claves, estas palabras indican el servicio o producto que se encuentra u ofrece en un determinado POI. Dado que un determinado servicio o producto podría ser referido de diferentes formas por diferentes usuarios es posible utilizar varias palabras para referirse un mismo servicio.

La siguiente figura muestra la estructura del archivo de base de datos utilizado:

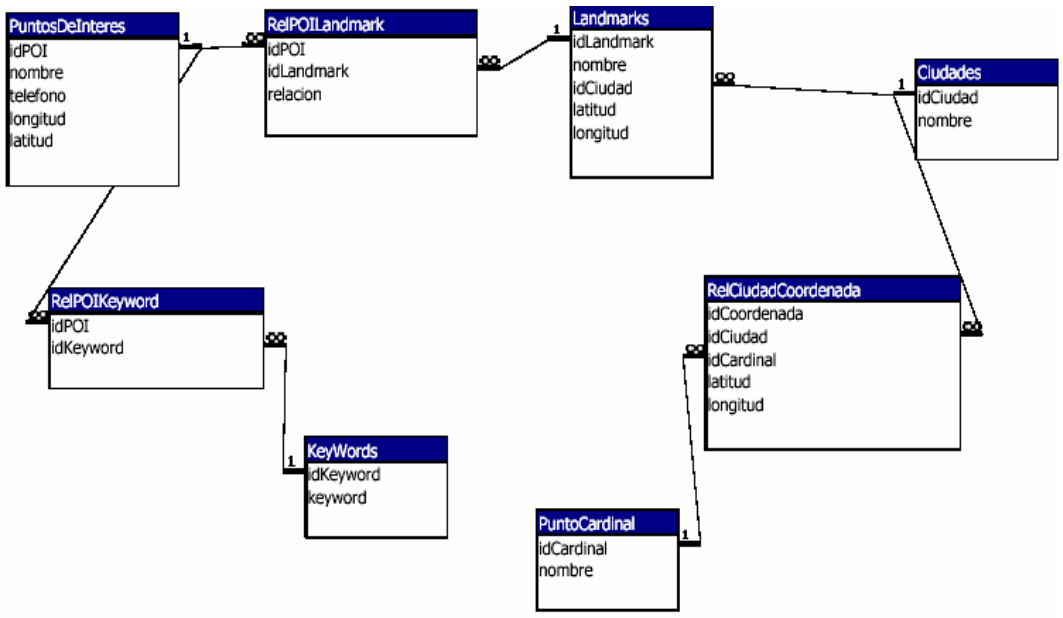
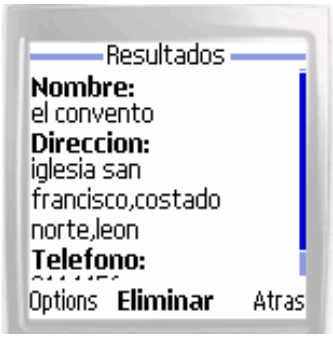


Figura 15: Diagrama de la base de datos utilizada.

Las siguientes imágenes son capturas de pantalla que muestran parte de la funcionalidad de la aplicación:

Pantalla principal: PrincipalScreen

Pantalla búsqueda: BusquedaScreen



Pantalla de resultados: Resultados Screen

Pantalla Categorías: GuardarEnCategoriaScreen

Pantalla Mis Sitios: MisPOIScreen

Pantalla Configuración: ConfiguracionScreen

Figura 16: Capturas de pantalla de la aplicación en funcionamiento.

Comentarios sobre los emuladores del Nokia® Prototype SDK 2.0:

Al ejecutar el programa, efectuar búsquedas y guardar resultados; terminar una sesión y luego reabrir el emulador los datos que estaban almacenados no aparecen; es decir que la aplicación POIExplorer muestra un mensaje que dice 'No hay datos' al seleccionar la opción 'Mis sitios', esto no es un error en el código de la aplicación, sino que se debe a que el emulador recrea todos sus archivos de almacenamiento cada vez que se ejecuta; probablemente para evitar que alguien corrompa los datos utilizados por él mismo y cause su mal funcionamiento.

Conclusión

J2ME es una plataforma de desarrollo software para móviles flexible. Proporciona las API necesarias para desarrollar aplicaciones de una forma rápida y uniforme. Entre sus ventajas está el proporcionar un modelo de seguridad al entorno que permite garantizar que las aplicaciones no corrompen el sistema ni hacen uso indebido de los datos del dispositivo. Además proporciona la portabilidad necesaria que permite a las aplicaciones ejecutarse de igual forma en diferentes aparatos, abstrayendo los detalles de implementación subyacentes (interfaces gráficas, sistemas de archivos, protocolos de red, etc.) lo que permite al programador enfocarse más en la funcionalidad de la aplicación.

Entre sus desventajas más notables está el rendimiento, que es una fracción del rendimiento de las aplicaciones nativas de los dispositivos, aunque esto ha sido disminuido con técnicas como compilación adaptiva y mejoras en la maquina virtual. La misma independencia del dispositivo ha hecho que invocación de métodos nativos del aparato sea una barrera para cierto tipo de aplicaciones, aunque en la mayoría de los casos esto significaría más una ventaja que una limitante.

Los servicios basados en localización son una de las aplicaciones más recientes de la industria de equipos móviles, ya que permiten crear aplicaciones que entregan información sumamente personalizada. Su infraestructura está soportada por un modelo en capas: posicionamiento, middleware y aplicaciones.

La capa de posicionamiento permite obtener la posición de los dispositivos en términos de coordenadas absolutas, relativas o de forma semántica.

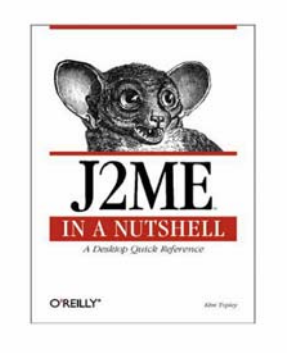
El middleware permite integrar la capa de posicionamiento con una serie de servicios base que son utilizados por las aplicaciones, entre estos servicios se encuentran: geocodificadores, geocodificadores inversos, determinación de rutas, de presentación, etc.

J2ME soporte para el desarrollo de aplicaciones basadas en localización a través del Java Specification Request 179 (JSR179), el cual proporciona una API de localización que abstrae los detalles de implementación de la tecnología de posicionamiento utilizada en una determinada red, facilitando el desarrollo e integración de los servicios en el aparato.

Bibliografía

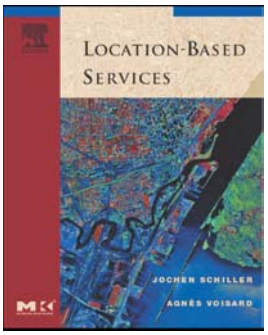
Kim Topley. ***J2ME in a nutshell***. O' Reilly. Marzo 2002.

Libro muy completo sobre J2ME. Aborda entre otras cosas la especificación CLDC1.0/MIDP1.0.



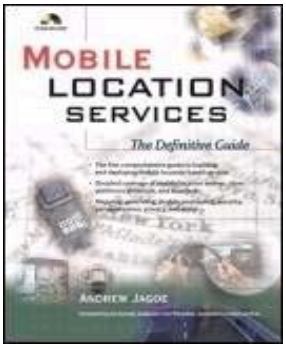
Jochen Schiller & Agnès Voisard. ***Location Based Services***. Morgan Kaufmann. Abril, 2004.

Libro muy explicativo sobre todos los aspectos relacionados al funcionamiento del lado del servidor de los servicios basados en localización.



Andrew Jagoe. ***Mobile Location Services: The Definitive Guide***. Prentice Hall. Diciembre, 2002.

Otro excelente libro sobre servicios basados en localización. Proporciona una perspectiva completa tanto del lado del servidor como del lado de las aplicaciones en los dispositivos móviles.



Otros recursos

Estos son sitios que han sido utilizados como referencia para complementar la información de este documento y para elaborar la aplicación de ejemplo, sin embargo no me responsabilizo por cualquier cambio en las direcciones proporcionadas, ya que tales cambios están fuera de mi alcance.

- http://www.java.sun.com

  El sitio Web de Sun Microsystems dedicado a Java.

- http://www.jcp.org

  Sitio del Java Comunnity Process.

- http://www.forum.nokia.com

  El sitio de Nokia® dedicado a J2ME y otras tecnologías de desarrollo.

- http://ericsson.com/developers/

  El sitio de Sony Ericcson® para desarrolladores, ahí se puede encontrar referencias y documentación sobre MPS.

- http://www.jlocationservices.com

  Sitio de Sun® para los desarrolladores de servicios basados en localización con Java.

- http://opengeospatial.org
- http://opengis.org

  Sitio oficial de OGC.

- http://www.openmobilealliance.org

  El sitio oficial del OMA.

- http://maps.google.com

  Sitio de Google® que brinda servicios de basados en localización en el Web.

Apéndice

Configuración de las herramientas software

Para poder probar la aplicación de ejemplo, se necesitará lo siguiente:

- Una maquina con al menos 128 MB de RAM ejecutando a 480 MHZ. Sistema operativo Windows 2000 SP3 o posterior.
- Java 2 Standard Development Kit v.>=1.4, el cual puede obtener desde http://java.sun.com./j2se
- Sun J2ME Wireless Toolkit v.>=2.2 si es impaciente, si no lo es puede obviar este paquete. Lo puede descargar desde http://java.sun.com/j2me
- Nokia Prototype SDK v.>=2.0. No utilice el Nokia Developer Suite a menos que este seguro que proporciona soporte para el JSR179. El prototype SDK 2.0 puede obtenerse desde la siguiente dirección: http://ncsp.forum.nokia.com:80/download/?asset_id=12384
- Apache Tomcat, que puede obtener desde http://jakarta.apache.org/tomcat
- Microsoft Access

Proceda como sigue:

Descargue e instale el J2SDK. Configure las siguientes variables de entorno para asegurarse que las referencias a librerías y clases funcionen correctamente:

- CLASSPATH=%JAVA_HOME%\jre\rt.jar
- PATH=%PATH%;%JAVA_HOME%\bin\;.

En este caso %JAVA_HOME% se refiere al directorio de instalación del J2SDK.

Para hacer esto en Windows 2000/XP ir a 'Propiedades de Mi PC'→'Avanzado'→'Variables de entorno'.

Descargue e instale el J2MEWTK. No se debe configurar ninguna variable en este caso, únicamente debe asegurarse instalarlo en un directorio que cuelgue directamente de una unidad, como C:\wtk22.

Descargue e instale el Nokia Prototype SDK, como directorio de instalación escoja %WTK_HOME%\wtklib\devices, para integrarlo con el J2MEWTK. En este caso %WTK_HOME% es la carpeta de instalación del wireless toolkit.

Con esto se han completado los pasos para hacer funcionar al cliente.

Para poner en marcha el servicio, descargue e instale Apache Tomcat. Debe configurar las siguientes variables de entorno:

- CATALINA_HOME=%DIR_INST_TOMCAT%\Tomcat 5.0, donde %DIR_INST_TOMCAT% es el directorio de instalación de Tomcat.
- Agregue a la variable CLASSPATH la siguiente cadena: ;%CATALINA_HOME%\common\lib*.jar. Este seguro que está cadena aparezca antes de la cadena que hace referencia al JDK.

Ahora ya se tiene configurado el servidor: haga una prueba en su explorador Web con la dirección http://127.0.0.1:%PUERTO%, donde %PUERTO% hace referencia al puerto elegido al instalar Tomcat, si no está seguro que puerto eligió pruebe el 8080.

Debe ver algo como lo siguiente:

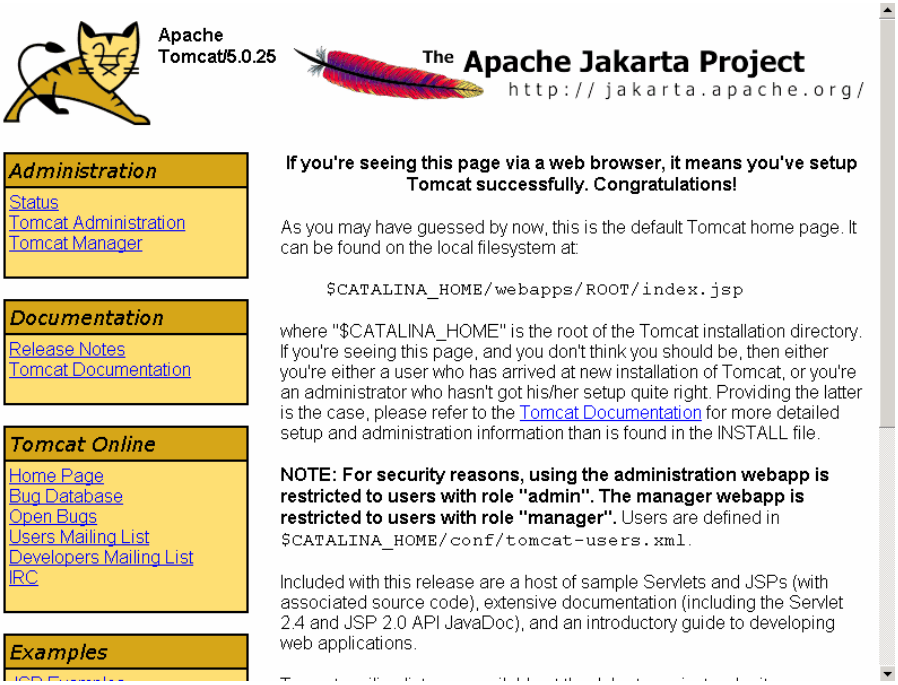


Figura 17: La pagina principal del servidor Apache Tomcat.

Este seguro de tener instalado Microsoft Access correctamente. Para configurar la conexión a la base de datos utilizada por el Servlet, vaya al 'Panel de control de Windows'→'Herramientas administrativas'→'Fuentes de datos (ODBC)', seleccione la pestaña 'DNS de sistema', clic en 'Agregar', seleccione 'Microsoft Access Driver (*.mdb)', clic en 'Finalizar'. En el campo 'Nombre de la fuente de datos' escriba 'POI_Source', en campo 'Base de datos' clic en 'Añadir', ubique el archivo llamado 'POIDatabase.mdb' en la carpeta donde haya copiado los archivos.

Para activar el servicio, copie la carpeta 'POI_Servlet' a la carpeta %CATALINA_HOME%\webapps y reinicie el servidor si es necesario.

Para ejecutar el ejemplo:

Copie la carpeta de 'Point of Interest Sample' a la carpeta %WTK_HOME%\apps. Ejecute el Wireless Toolkit, clic en 'File'→'Open project', seleccione 'Point of Interest Sample'. En el campo 'Device' seleccione 'Nokia prototype SDK_2_0', clic en el menú 'Project'→'Package'→'Create package'. Al terminar de compilar el proyecto, clic en el menú 'Project'→'Run'; aparecerá una ventana que le pide escoger el dispositivo, en este caso puede seleccionar 'Prototype_2_0_S40_MIDP_Emulator' o 'Prototype_2_0_S60_MIDP_Emulator', también puede seleccionar otro emulador sin embargo verá un mensaje de error indicando problemas de compatibilidad. En dependencia del hardware de su computadora el ejemplo debe ejecutarse más o menos rápido. Verá ciertos mensajes aparecer por la consola del WTK indicando la operación que se está efectuando.

Si al compilar el proyecto aparecen errores con el siguiente mensaje:
cannot resolve symbol
symbol : class <javax.microedition.Location*>

Copie el archivo:

%WTK_HOME%\wtklib\devices\Nokia_Prototype_SDK_2_0\lib\ext\location.zip

a

%WTK_HOME%\apps\lib.

Una vez copiado, con un programa manejador de archivos zip borre dentro del archivo, la carpeta llamada *com* y deje únicamente las carpetas *META-INF y javax*. Luego intente recompilar el archivo, los errores deberán haber desaparecido.