

## **CAPÍTULO 3. DESARROLLO DE LA APLICACIÓN BLUETOOTH PARA UN CELULAR UTILIZANDO JAVA ME.**

El Lenguaje Java es una Plataforma demasiado extensa, por esta razón se debe escoger el entorno adecuado según la aplicación que se quiera conseguir. Este proyecto se define dentro del entorno Java ME, para dispositivos pequeños.

En este capítulo se desarrollará el código en Lenguaje Java, es decir, se programará el MIDlet que permita usar la tecnología Bluetooth con que cuentan los dispositivos móviles, de una manera amigable y fácil de manipular por parte del usuario.

### **3.1 HERRAMIENTAS DE DESARROLLO**

Este proyecto fue creado con la ayuda de NetBeans IDE 5.0 + NetBeans Mobility 5.0 para Windows. También fue probado con la ayuda de Sun Java Wireless Toolkit 2.5.2 for CLDC y simulado también con la ayuda de CLDC KToolBar. A continuación se revisarán algunas características de estas herramientas.

#### **3.1.1 NETBEANS IDE 5.0**

NetBeans es un proyecto de código abierto de gran éxito con una gran base de usuarios, una comunidad en constante crecimiento, y con cerca de 100 socios en todo el mundo. Sun Microsystems fundó el proyecto de código abierto NetBeans en junio 2000 y continúa siendo el patrocinador principal de los proyectos.

NetBeans es una plataforma para el desarrollo de aplicaciones Java usando un Entorno de Desarrollo Integrado (IDE), una herramienta para programadores

usada para editar, compilar, depurar y ejecutar programas. NetBeans IDE soporta el desarrollo de todos los tipos de aplicación Java (J2SE, Web, EJB y aplicaciones móviles). La versión mas reciente es netBeans 6.1 lanzada al mercado el Abril 28 de 2008.

En la Figura 3.1 se observa la presentación de NetBeans 5.0, programa con el cual que se realizó este proyecto.

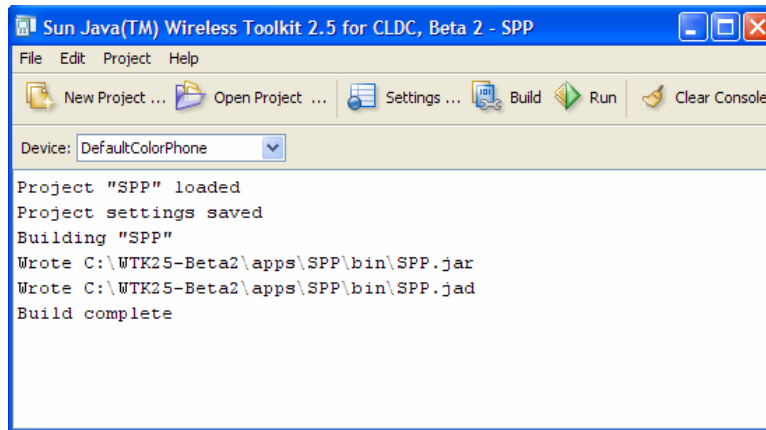


**Figura 3.1** netBeans IDE 5.0

NetBeans Mobility 5.0 es un módulo usado por la plataforma NetBeans para desarrollar aplicaciones móviles, incluye las librerías necesarias y la opción de edición del código fuente de manera interactiva para implementar aplicaciones para dispositivos móviles.

### **3.1.2 SUN JAVA WIRELESS TOOLKIT 2.5 FOR CLDC**

Es un conjunto de herramientas para el desarrollo de aplicaciones inalámbricas que se basan en la plataforma J2ME, *Connected Limited Device Configuration* (CLDC) y *Mobile Information Device Profile* (MIDP), diseñadas para funcionar en teléfonos celulares y otros pequeños dispositivos móviles. Sun Java Wireless Toolkit 2.5 incluye entornos de emulación, características de optimización y rendimiento, documentación y ejemplos muy útiles para lograr eficientes y exitosas aplicaciones. Es similar a J2ME Wireless Toolkit 2.2 y tienen incluidos varios modelos de teléfonos para la simulación.



**Figura 3.2** Pantalla principal de CLDC KTollbar

### 3.2 PROGRAMACIÓN EN JAVA ME

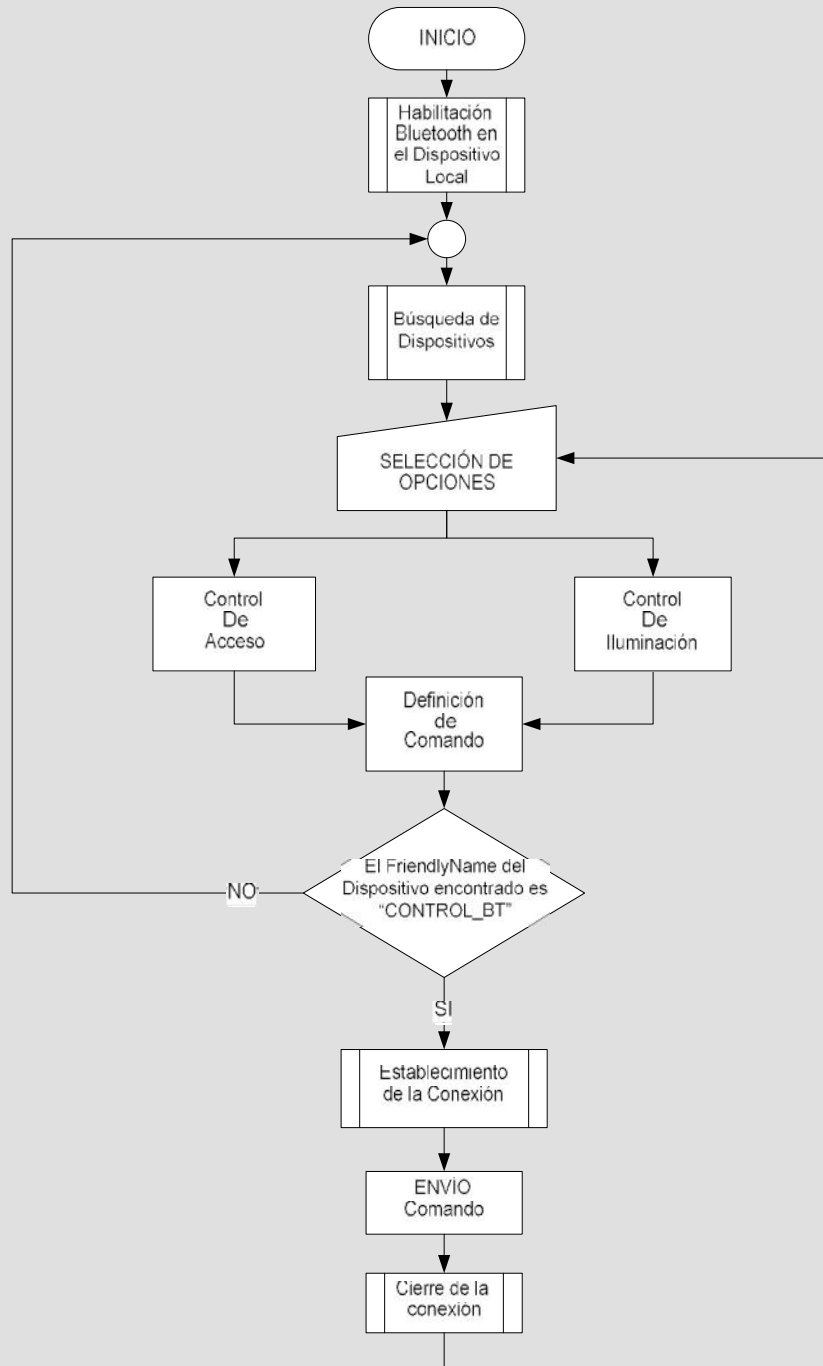
Un MIDlet tiene que ejecutarse en un entorno muy concreto (un dispositivo con soporte Java ME). Un MIDlet tiene que heredar de la clase MIDlet e implementar una serie de métodos de dicha clase. La clase de la que ha de heredar cualquier MIDlet es `javax.microedition.midlet.MIDlet`.\*

Un MIDlet puede estar en tres estados diferentes: en ejecución, en pausa o finalizado. Dependiendo del estado en el que esté, la máquina virtual llamará al método heredado correspondiente, es decir, `startApp()` cuando entre en ejecución, `pauseApp()` cuando el MIDlet entre en pausa y `destroyApp()` a la finalización del MIDlet.

Las clases de `javax.microedition.lcdui.*` dan soporte para la interfaz de usuario. Permiten controlar la pantalla del dispositivo y también la entrada/salida desde el teclado.

Dentro de la aplicación, la interfaz de usuario interactuará con la interfaz de comunicación, la cual establecerá la conexión Bluetooth, para alcanzar el objetivo de control de este proyecto.

En la Figura 3.3 se puede observar el Diagrama de Flujo definido para esta aplicación.



**Figura 3.3** Diagrama de Flujo de la Aplicación Bluetooth

### 3.2.1 PROGRAMACIÓN DE LA INTERFAZ DE USUARIO

La aplicación Java ME estará sustentada principalmente en dos APIs, por un lado CLDC que hereda algunas de las clases de J2SE, y MIDP que añade nuevas clases que permitirán crear interfaces de usuario.

Las clases más importantes de J2SE que ofrece CLDC son las siguientes:

- a. java.lang
- b. java.util
- c. java.io

Además MIDP añade los siguientes paquetes:

- a. javax.microedition.midlet
- b. javax.microedition.lcdui
- c. javax.microedition.io
- d. javax.microedition.rms

El paquete javax.microedition.midlet, es el más importante de todos. Sólo contiene a la clase MIDlet, que ofrece un marco de ejecución para aplicaciones sobre dispositivos móviles. El paquete javax.microedition.lcdui ofrece una serie de clases e interfaces de utilidad para crear interfaces de usuario.

#### 3.2.1.1 Elementos de la Interfaz de Usuario

**Command** es un elemento que permite interaccionar con el usuario y le permite introducir comandos. Están disponibles los siguientes tipos de comandos:

COMANDO	DESCRIPCIÓN
OK	Confirma una selección
CANCEL	Cancela la acción actual
BACK	Traslada al usuario a la pantalla anterior
STOP	Detiene una operación
HELP	Muestra una ayuda
SCREEN	Tipo genérico referente a la pantalla actual
ITEM	Tipo genérico referente a un elemento de la pantalla actual

**Tabla 3.1** Tipos de comandos<sup>31</sup>

---

<sup>31</sup> Tomado de la documentación del JSR-82

A veces, y dependiendo del modelo y marca del dispositivo, sólo se pueden mostrar un número limitado de comandos en la pantalla. Al resto se accederá mediante un menú. En la Figura 3.4 se puede observar una parte del código de este programa. El método `get_itemCommand_ON()` define un comando con sus respectivas características. En la función `get_form_cIluminacion()` se define un formulario y se le añade el comando creado, conjuntamente con otros comandos.

```
public Command get_itemCommand_ON() {
    if (itemCommand_ON == null) {
        itemCommand_ON = new Command("ON", "On", Command.ITEM, 1);
    }
    return itemCommand_ON;
}

public Form get_form_cIluminacion() {
    if (form_cIluminacion == null) {
        form_cIluminacion = new Form("Control de
Iluminacion", new Item[] {get_imageItem1()});
form_cIluminacion.addCommand(get_backCommand_ACCaOPC());
form_cIluminacion.addCommand(get_itemCommand_ON());
form_cIluminacion.addCommand(get_itemCommand_OFF());
form_cIluminacion.addCommand(get_screenCommand_INFO());
form_cIluminacion.setCommandListener(this);
    }
    return form_cIluminacion;
}
```

**Figura 3.4** Métodos `get_itemCommand_ON()` y `get_form_cIluminacion()`

Dentro de la aplicación, el resultado del código anterior se puede ver en la Figura 3.5.



**Figura 3.5** Comandos dentro del formulario de Control de Iluminación

### 3.2.1.1.1 *La clase Screen*

Hereda directamente de Displayable y permite crear las interfaces gráficas de alto nivel. Un objeto que herede de la clase Screen será capaz de ser mostrado en la pantalla. Se pueden encontrar cuatro clases que heredan de Screen y que sirven de base para crear las interfaces de usuario, son: Alert, Form, List y TextBox.

Se puede imaginarlo como una serie de fichas de las cuales sólo se puede mostrar una cada vez. Para cambiar de una pantalla a otra se debe usar el método setCurrent de la clase Display.

Cada uno de las clases anteriores dispone de los métodos (realmente lo heredan de Screen):

- a. String getTitle () - Devuelve el título de la pantalla
- b. void setTitle (String s) - Establece el título de la pantalla
- c. Ticker getTicker () - Devuelve el ticker de la pantalla
- d. void setTicker(Ticker ticker) - Establece el ticker de la pantalla

Ticker es una línea de texto que aparece en la parte superior de la pantalla con un scroll lateral automático.

**La clase Alert**, permite mostrar una pantalla de texto durante un tiempo o hasta que se produzca un comando de tipo OK. Se utiliza para mostrar errores u otro tipo de mensajes al usuario. El tipo de alerta puede ser uno de los siguientes: ALARM, CONFIRMATION, ERROR, INFO, WARNING.

La diferencia entre uno y otro tipo de alerta es básicamente el tipo de sonido o efecto que produce el dispositivo. En la Figura 3.6 se puede observar el código usado para crear este tipo de pantalla a través del método get\_Informacion().

```

public Alert get_Informacion() {
    if (Informacion == null) {
        Informacion = new Alert("EPN", "PROYECTO DE\n
        TITULACION\nJavier\n    Villagran",
        get_image_informacion(), AlertType.INFO);
        Informacion.setTicker(get_ticker_about());
        Informacion.setTimeout(-2);
    }
    return Informacion;
}

```

**Figura 3.6** Pantalla de Alerta

En este método, se define un Alert llamado Información, se le añade el título, un texto, una imagen y se configura el tiempo que se debe mostrar. La Figura 3.7 muestra el resultado del código anteriormente detallado.



**Figura 3.7** Alerta de Información mostrada en la aplicación

**La clase List**, permite crear listas de elementos seleccionables. Los posibles tipos de lista son: EXCLUSIVE, que permite seleccionar un solo elemento a la vez; IMPLICIT, que permite seleccionar un elemento usando un comando; MULTIPLE, que permite tener varios elementos seleccionados simultáneamente.

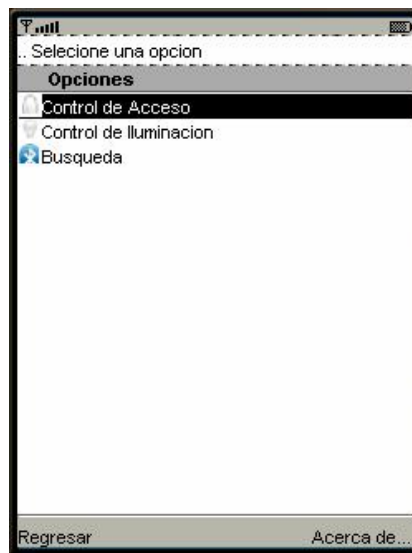


Un ejemplo del uso de esta clase se encuentra en la Figura 3.8, una parte del código fuente que incluye la clase List.

```
public List get_Opciones() {
    if (Opciones == null) {
        Opciones = new List("Opciones", Choice.IMPLICIT, new String[] {
            "Control de Acceso",
            "Control de Iluminacion",
            "Busqueda"
        }, new Image[] {
            get_image_acceso(),
            get_image_iluminacion(),
            get_image_busqueda()
        });
        Opciones.addCommand(get_backCommand_ILMaOPC());
        Opciones.addCommand(get_screenCommand_INFO());
        Opciones.setCommandListener(this);
        Opciones.setTicker(get_ticker_opciones());
        Opciones.setSelectedFlags(new boolean[] {
            true, false, false
        });
    }
    return Opciones;
}
```

**Figura 3.8** Código fuente para crear una List.

En el método get\_Opciones() se define la lista Opciones con los elementos que se pueden seleccionar, el texto que se debe mostrar, una imagen para cada opción y además se añaden comandos a la lista y se define la primera opción seleccionada por defecto. En la Figura 3.9 se puede observar el resultado.



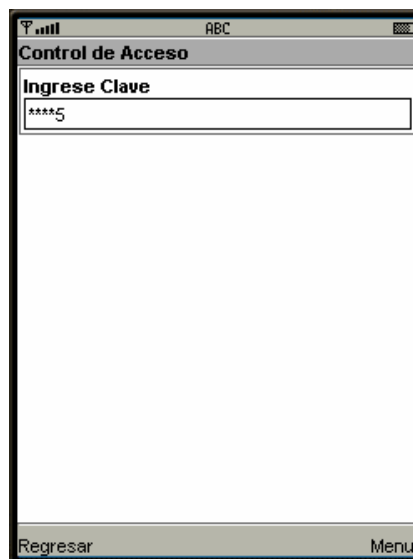
**Figura 3.9** Lista Opciones para seleccionar un formulario

**La clase TextBox** permite introducir y editar texto a pantalla completa. Es como un pequeño editor de textos. Las limitaciones o tipos de texto aceptado pueden ser alguna de los siguientes: ANY, sin limitación; EMAILADDR, dirección de email; NUMERIC, sólo se permiten números; PASSWORD, los caracteres no serán visibles; PHONENUMBER, sólo número de teléfono; URL, sólo direcciones URL.

```
public Form get_form_cAcceso() {
    if (form_cAcceso == null) {
        form_cAcceso = new Form("Control de Acceso", new
        Item[] {get_image_cAcceso()});
        form_cAcceso.addCommand(get_backCommand_ACCaOPC());
        form_cAcceso.addCommand(get_itemCommand_ABRIR());
        form_cAcceso.addCommand(get_screenCommand_INFO());
        form_cAcceso.setCommandListener(this);
        TextField_clave = new TextField("Ingrese Clave", "",
        20, TextField.PASSWORD);
        form_cAcceso.append(TextField_clave);
    }
    return form_cAcceso;
}
```

**Figura 3.10** Código para insertar un Cuadro de Texto.

El método `get_form_cAcceso()` define un formulario que incluye un cuadro de texto, `TextField_clave`, de tipo `PASSWORD`. En la Figura 3.11 se observa dicho formulario y el cuadro de texto con el texto oculto.



**Figura 3.11** Formulario que incluye un cuadro de texto

**La clase Form** es un elemento de tipo contenedor, es decir, es capaz de contener una serie de elementos visuales con los que se construyen interfaces más elaboradas. Los elementos que se podrían añadir a un formulario son:

- a. StringItem
- b. ImageItem
- c. TextField
- d. DateField
- e. ChoiceGroup
- f. Gauge

En las Figuras 3.8 y 3.10 se pudieron apreciar ejemplos de métodos que definen formularios.

La clase Form es capaz de manejar objetos derivados de la **clase Item**, que representa a un elemento visual que no ocupará toda la pantalla, sino que formará parte de la interfaz de usuario junto con otros elementos.

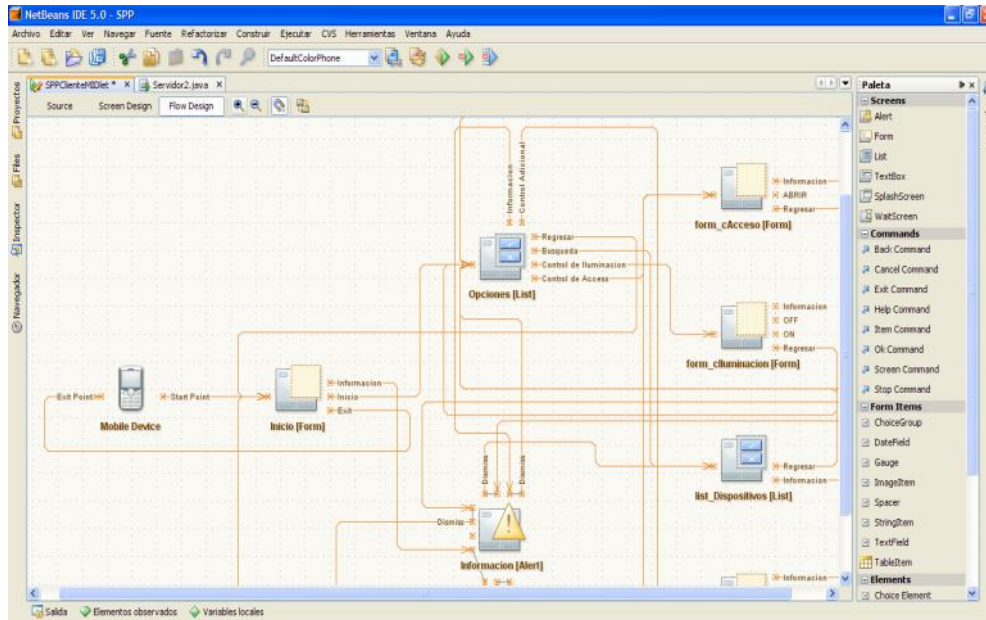
Hay métodos de la clase Form que permiten añadir, eliminar y modificar elementos del formulario, son las siguientes:

- a. append() añade al formulario un elemento.
- b. delete() elimina un elemento del formulario.
- c. insert() inserta un elemento en una posición indicada

### **3.2.1.2 Flow Design de Netbeans Mobility 5.0**

NetBeans Mobility también tiene la posibilidad de crear la interfaz de usuario a través de la opción Flow Design, en donde se agregan listas, alertas, formularios, comandos, etc. de manera interactiva mientras el código fuente se va modificando automáticamente. Por ejemplo, para incluir un formulario en la aplicación solo es necesario arrastrarlo hasta la hoja de trabajo.

En la Figura 3.12 se ve la pantalla de edición interactiva con el diseño de la Aplicación completa.



**Figura 3.12** Vista de la aplicación usando Flow Design

El código generado por Flow Design aparece resaltado en color celeste, Figura 3.13, y no se puede cambiar de forma manual, sino únicamente usando esta misma herramienta.

```

574 public ImageItem get_image_cAcceso() {
575     if (image_cAcceso == null) {
576         // Insert pre-init code here
577         image_cAcceso = new ImageItem("", get_image_c_acceso(), Item.LAYOUT_CENTER | Item.LAYOUT_VCE
578         // Insert post-init code here
579     }
580     return image_cAcceso;
581 }
582
583 /** This method returns instance for image_apagado component and should be called instead of accessi
584  * @return Instance for image_apagado component
585  */
586 public Image get_image_apagado() {
587     if (image_apagado == null) {
588         // Insert pre-init code here
589         try {
590             image_apagado = Image.createImage("/focoapagado.png");
591         } catch (java.io.IOException exception) {
592         }
593         // Insert post-init code here
594     }
595     return image_apagado;
596 }
597
598 /** This method returns instance for imageItem1 component and should be called instead of accessing

```

**Figura 3.13** Vista del código generado por Flow design

Flow Design facilita la creación de la interfaz de usuario. Es beneficioso ir supervisando visualmente cada elemento incluido en una pantalla y estar seguro de que el resultado es satisfactorio paso a paso. Esta herramienta es de gran ayuda, aunque suele crear muchas líneas de código que algunas veces no son indispensables.

La programación de la parte de comunicación no se puede realizar con esta herramienta, solamente se puede hacer manualmente como se revisará a continuación.

### **3.2.2 PROGRAMACIÓN DE LA INTERFAZ DE COMUNICACIÓN**

Un MIDlet puede establecer diversos tipos de conexiones: *Sockets*, *http*, *https*, *datagramas*, *bluetooth* y otras.

En una comunicación Bluetooth existe un dispositivo que ofrece un servicio (servidor) y otros dispositivos que acceden a él (clientes). En este caso se está programando la parte del cliente y la parte del servidor está ya implementada en el módulo Bluetooth que ofrece el servicio de puerto serial, SPP.

El módulo Bluetooth, como servidor, deberá hacer las siguientes operaciones:

- a. Crear una conexión servidora
- b. Especificar los atributos de servicio
- c. Aceptar las conexiones clientes

La aplicación Bluetooth deberá realizar las siguientes funciones:

- a. Búsqueda de dispositivos. La aplicación realizará una búsqueda de los dispositivos Bluetooth a su alcance que estén en modo conectable.
- b. Búsqueda de servicios. La aplicación realizará una búsqueda de servicios por cada dispositivo encontrado.
- c. Establecimiento de la conexión. Una vez encontrado un dispositivo que ofrece el servicio deseado se podrá realizar la conexión.

- d. Comunicación. Ya establecida la conexión es posible leer y escribir sobre ésta.

Estas funciones se revisarán a continuación analizando el código que permite llevarlas a cabo.

### **3.2.2.1 Búsqueda de Dispositivos**

#### **3.2.2.1.1 BCC (*Bluetooth Control Center*)**

<sup>32</sup>El BCC es un conjunto de capacidades que permiten al usuario o al fabricante resolver peticiones conflictivas de aplicaciones previniendo que una aplicación pueda perjudicar a otra. Gracias al BCC, los dispositivos que implementen el API de Bluetooth pueden permitir que múltiples aplicaciones se ejecuten simultáneamente.

El BCC o pila Bluetooth, puede ser una aplicación nativa, una aplicación en un API separado, o sencillamente un grupo de parámetros fijados por el proveedor que no pueden ser cambiados por el usuario.

El BCC define las características de seguridad. Se encargará de manejar el modo de seguridad que la pila debe usar y mantendrá las listas de dispositivos seguros. El API de Bluetooth permite a la aplicación especificar sus requerimientos de autenticación y encriptación.

La pila Bluetooth es la responsable de controlar el dispositivo Bluetooth, por lo que es necesario inicializarla antes de iniciar el proceso de conexión. La especificación deja la implementación del BCC a los fabricantes, y cada uno maneja la inicialización de una manera diferente.

---

<sup>32</sup> Tomado de "Java™ APIs for Bluetooth™ Wireless Technology (JSR 82)", *Specification Version 1.1*

### **3.2.2.1.2 *Habilitación dispositivo local***

Los objetos Bluetooth esenciales en una conexión son LocalDevice y RemoteDevice. La clase LocalDevice provee acceso y control del dispositivo local. Las clases DeviceClass y BluetoothStateException dan soporte a la clase LocalDevice. La clase RemoteDevice representa un dispositivo remoto y provee métodos para obtener información de dicho dispositivo remoto.

#### **Clase `javax.bluetooth.LocalDevice`**

Esta clase provee acceso y control sobre el dispositivo local Bluetooth. Está diseñada para cumplir con los requerimientos del Generic Access Profile, GAP, definidos para Bluetooth.

Un objeto LocalDevice representa al dispositivo local. Este objeto será el punto de partida de cualquier operación que se pueda llevar a cabo con este API.

Alguna información de interés que se obtiene de este objeto a través de sus métodos es, por ejemplo, la dirección Bluetooth de nuestro dispositivo, el apodo o "friendlyName". A través de este objeto también se puede obtener y establecer el modo de conectividad: la forma en que nuestro dispositivo está o no visible para otros dispositivos usando el método `setDiscoverable()`.

<sup>33</sup>Los posibles valores que admite el método `setDiscoverable()` están definidos en la clase `DiscoveryAgent` como campos estáticos. El valor que se usa para esta clase es GIAC, General/Unlimited Inquiry Access Code.

En la Figura 3.14, se observa el método `comenzar()` y el uso de algunos métodos de la clase `LocalDevice` para habilitar el Bluetooth en el dispositivo local.

---

<sup>33</sup> Tomado del `jsr82_1.1_javadocs` incluido en el JSR82 de [www.sun.com](http://www.sun.com)

```

private void comenzar() {
    LocalDevice localDevice = null;
    try {
        localDevice = LocalDevice.getLocalDevice();
        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
        discoveryAgent = localDevice.getDiscoveryAgent();
        String dir_local = localDevice.getBluetoothAddress();
        Inicio.append("dispositivo local:"+dir_local);
    } catch(Exception e) {
        e.printStackTrace();
        Alert alert = new Alert("Error", "No se puede hacer uso de
Bluetooth", null, AlertType.ERROR);
        Display.getDisplay(this).setCurrent(alert);
    }
}

```

**Figura 3.14** Función comenzar()

Dado que los dispositivos inalámbricos son móviles, necesitan un mecanismo que permita encontrar, conectar, y obtener información sobre las características de dichos dispositivos, todas estas tareas son realizadas por el API de Bluetooth. La búsqueda de dispositivos y servicios son tareas que solamente realizarán los dispositivos clientes, en este caso, el teléfono celular.

### **3.2.2.1.3 Descubrimiento de Dispositivos**

A través de cualquier aplicación se puede encontrar dispositivos y crear una lista de todos ellos, para esto se usa startInquiry(). Este método requiere que la aplicación tenga especificado un “listener”, una interfaz que es notificada cuando un nuevo dispositivo es encontrado después de haber lanzado un proceso de búsqueda. Esta interfaz es **javax.bluetooth.DiscoveryListener**.

```

private void buscar() {
    dispositivosEncontrados= new Vector();
    etiquetasDospositivos= new Vector();
    try {
        discoveryAgent.startInquiry(discoveryAgent.GIAC, this);
    } catch(BluetoothStateException e) {
        e.printStackTrace();
        Alert alert = new Alert("AlertError","No se pudo comenzar la
búsqueda",null, AlertType.ERROR);
        Display.getDisplay(this).setCurrent(alert);
    }
}

```

**Figura 3.15** Función Buscar()



La interfaz `javax.bluetooth.DiscoveryAgent` provee métodos para descubrir dispositivos y servicios. En este caso, para comenzar una nueva búsqueda de dispositivos es llamado el método `startInquiry()`, como se puede apreciar en la Figura 3.16, este método requiere dos argumentos. El primer argumento es un entero que especifica el modo de conectividad definido para la búsqueda, este valor deberá ser `DiscoveryAgent.GIAC`, para tener un acceso ilimitado. El segundo argumento es un objeto que implemente `DiscoveryListener`. A través de este último objeto serán notificados los dispositivos que se vayan descubriendo. Para cancelar la búsqueda se debe usar el método `cancellInquiry()` de la misma interfaz `javax.bluetooth.DiscoveryAgent`.

La **interfaz `DiscoveryListener`** tiene los siguientes métodos usados para el descubrimiento de dispositivos:

- a. `public void deviceDiscovered(RemoteDevice rd, DeviceClass c)`
- b. `public void inquiryCompleted(int c)`

Cada vez que se descubre un dispositivo se llama al método **`deviceDiscovered(RemoteDevice rd, DeviceClass c)`**. Se pasan dos argumentos. El primero es un objeto de la clase `RemoteDevice` que representa el dispositivo encontrado. El segundo argumento permitirá determinar el tipo de dispositivo encontrado.

```
public void deviceDiscovered(RemoteDevice remoteDevice, DeviceClass
deviceClass) {
    String address = remoteDevice.getBluetoothAddress();
    String friendlyName = null;
    try {
        friendlyName = remoteDevice.getFriendlyName(true);
    } catch(IOException e) { }
    String device = null;
    if(friendlyName == null) {    device = address;
    } else {    device = "("+friendlyName+" : "+address+"";
    }
    etiquetasDospositivos.addElement(device);
    dispositivosEncontrados.addElement(remoteDevice);
    Display.getDisplay(this).setCurrent(list_dispositivos);
}
```

**Figura 3.16** Método `deviceDiscovered()`

En la Figura 3.17 se observa el código que permite obtener el friendlyName y la dirección de los dispositivos hallados en una búsqueda para presentarlos como elementos de una Lista llamada list\_dispositivos.

El método **inquiryCompleted(int c)** es llamado cuando la búsqueda de dispositivos ha finalizado pasa un argumento entero indicando el motivo de la finalización. Este argumento podrá tomar los valores:

DiscoveryListener.INQUIRY\_COMPLETED si la búsqueda ha concluido con normalidad, DiscoveryListener.INQUIRY\_ERROR si se ha producido un error en el proceso de búsqueda, o DiscoveryListener.INQUIRY\_TERMINATED si la búsqueda fue cancelada. Para cada caso se ha definido un acción, se puede revisar en la Figura 3.18.

```
public void inquiryCompleted(int i) {  
    switch(i) {  
        case DiscoveryListener.INQUIRY_COMPLETED:  
            System.out.println("Busqueda de dispositivos concluida con  
normalidad");  
            break;  
        case DiscoveryListener.INQUIRY_TERMINATED:  
            System.out.println("Busqueda de dispositivos cancelada");  
            break;  
        case DiscoveryListener.INQUIRY_ERROR:  
            System.out.println("Busqueda de dispositivos finalizada debido a un  
error");  
            break;  
    }  
    Display.getDisplay(this).setCurrent(list_dispositivos);  
}
```

**Figura 3.17** Función inquiryCompleted(int i)

### **Clase javax.bluetooth.RemoteDevice**

Esta clase representa al dispositivo Bluetooth remoto, el dispositivo con el que se podría realizar la conexión Bluetooth. De ella se obtiene la información básica acerca de un dispositivo remoto, su dirección Bluetooth y su friendlyName), entre otros, usando los métodos que se muestran en la Tabla 3.2, a continuación.

MÉTODO	RESUMEN
<u>authenticate()</u>	Intenta autenticar el RemoteDevice.
<u>authorize()</u>	Determina si el RemoteDevice esta habilitado para acceder al servicio local provisto por la conexión.
<u>encrypt()</u>	Intenta activar o desactivar la encriptación para una conexión existente.
<u>equals()</u>	Determina si dos RemoteDevices son iguales.
<u>getBluetoothAddress()</u>	Devuelve la dirección Bluetooth del dispositivo.
<u>getFriendlyName()</u>	Retorna el nombre del dispositivo.
<u>getRemoteDevice()</u>	Retorna el dispositivo Bluetooth remoto de una conexión SPP, L2CAP u OBEX.
<u>hashCode()</u>	Calcula el código Hash para este objeto
<u>isAuthenticated()</u>	Determina si el RemoteDevice ha sido autenticado.
<u>isAuthorized()</u>	Determina si el RemoteDevice ha sido autorizado previamente por el BCC del dispositivo local para intercambiar datos.
<u>isEncrypted()</u>	Determina si el intercambio de datos con el RemoteDevice está bien encriptada.
<u>isTrustedDevice()</u>	Determina si el dispositivo es de confianza según el BCC.

**Tabla 3.2** Métodos admitidos por la clase RemoteDevice<sup>34</sup>

Al llamar al **método getLocalDevice()** se puede producir una excepción del tipo BluetoothStateException, cuando un dispositivo no puede atender una petición que normalmente atendería, probablemente debido a algún problema propio del equipo. Esto significa que no se pudo inicializar el sistema Bluetooth. La excepción BluetoothStateException extiende de java.io.IOException y no añade ningún método adicional.

El método **getDeviceClass()** devuelve un objeto de tipo DeviceClass. Este tipo de objeto describe el tipo de dispositivo, se puede saber, por ejemplo, si se trata de un teléfono o de un ordenador.

### 3.2.2.2 Búsqueda de Servicios

En un área de cobertura se pueden encontrar otros dispositivos Bluetooth a más del módulo usado en este proyecto. Para identificar un dispositivo

<sup>34</sup> Tomado de la documentación del JSR-82

Bluetooth se puede aprovechar alguna característica que lo hace diferente de los demás, por ejemplo, su friendlyName o su Bluetooth Address.

#### **3.2.2.2.1 Selección del Dispositivo de Control**

En el programa se lleva a cabo una comparación entre el Friendlyname del dispositivo encontrado y el que se espera: "CONTROL\_BT", si concuerda, se aceptará la comunicación con este dispositivo, se realizará la búsqueda de servicios y posteriormente se enviará la información de control. En la Figura 3.18 se observa la parte del código del programa donde se realiza la comparación del friendlyName de cada dispositivo encontrado con el esperado.

```
private RemoteDevice seleccionarDispositivo(){
    try{
        for(int i=0;i<dispositivosEncontrados.size();i++){
            RemoteDevice remoteDeviceTmp =
                (RemoteDevice) dispositivosEncontrados.elementAt(i);
            if("CONTROL_BT".equals(remoteDeviceTmp.getFriendlyName(true))){
                return remoteDeviceTmp; }
        }
    } catch(IOException io){ io.printStackTrace(); }
    return null;
}
```

**Figura 3.18** Función seleccionarDispositivo()

#### **3.2.2.2.2 Descubrimiento de Servicios**

Para realizar una búsqueda de servicios también se usa la clase DiscoveryAgent y se implementa la interfaz DiscoveryListener.

La clase DiscoveryAgent provee de métodos para buscar servicios en un dispositivo servidor Bluetooth e iniciar transacciones entre el dispositivo y el servicio.

Para comenzar la búsqueda se usa `searchServices()` de la clase `DiscoveryAgent`, que se verá con más detalle:

### **Método `searchServices(int[], UUID[], RemoteDevice, DiscoveryListener)`**

El primer argumento es un arreglo de enteros con el que se especifica los atributos de servicio que interesan. El segundo argumento es un arreglo de identificadores de servicio. Permite especificar los servicios en los que el cliente está interesado. La clase `UUID` (*universally unique identifier*) representa identificadores únicos universales. Más adelante, en la Figura 3.20 se observará que el valor de `UUID` usado en el presente proyecto es "0x1101" que es el correspondiente para el Perfil de Puerto Serial, SPP. El tercer argumento es el dispositivo remoto sobre el que se va a realizar la búsqueda.

Por último argumento se pasará un objeto que implemente `DiscoveryListener` que será usado para notificar los eventos de búsqueda de servicios.

En la Figura 3.19, se puede ver estas definiciones dentro del código del programa, en donde se hace la búsqueda de servicios en el dispositivo remoto seleccionado.

```
private void escogerOpcionEncendido(){
    RemoteDevice remoteDevice2 = seleccionarDispositivo();
    try {
        System.out.println("remote device:"
                           +remoteDevice2.getFriendlyName(true));
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    String addr = remoteDevice2.getBluetoothAddress();
    try {
        int transId = discoveryAgent.searchServices(ATRIBUTOS, SERVICIOS,
remoteDevice2, this);
        get_form_cIluminacion().append("Encontrado Dispositivo de
Control" + remoteDevice2.getFriendlyName(true) + ";" + transId);
    } catch (Exception e) {
        e.printStackTrace();
        System.err.println("No se pudo comenzar la busqueda");
    }
}
```

**Figura 3.19** Función `escogerOpcionEncendido()`

Los campos del método `discoveryAgent.searchServices`, en el cuadro anterior, están declarados de la siguiente manera:

```
public static final UUID SERVICIO_CONTROL = new UUID(0x1101);
public static final UUID[] SERVICIOS = new UUID[] {SERVICIO_CONTROL };
public static final int[] ATRIBUTOS = null;
private RemoteDevice remoteDevice2 =null;
```

**Figura 3.20** Valores del método `searchService`

Es posible que se desee hacer diversas búsquedas de servicios al mismo tiempo. El método `searchServices()` devolverá un entero que identificará la búsqueda. Este valor entero servirá para saber a qué búsqueda pertenecen los eventos `servicesDiscovered()` y `serviceSearchCompleted()`.

### **Método `serviceSearchCompleted(int transID, int respCode)`**

Este método es llamado cuando se finaliza un proceso de búsqueda. El primer argumento identifica el proceso de búsqueda (el valor devuelto al invocar el método `searchServices()` de la clase `DiscoveryAgent`). El segundo argumento indica el motivo de finalización de la búsqueda.

En la Figura 3.21 se observa esta función dentro del código general de la aplicación.

```
public void serviceSearchCompleted(int i, int i) {
    System.out.println("Terminada la busqueda de servicios");
}
```

**Figura 3.21** Función `serviceSearchCompleted`

Se puede cancelar un proceso de búsqueda de servicios llamando al método **`cancelServiceSearch()`** pasándole como argumento el identificador de proceso de búsqueda, que es el número entero devuelto cuando se comenzó la búsqueda con `searchServices()`.

### **Método servicesDiscovered(int transID, ServiceRecord[] sr)**

Este método notifica que se han encontrado servicios. El primer argumento es un entero que es el que identifica el proceso de búsqueda. Este entero es el mismo que devolvió searchDevices() cuando se comenzó la búsqueda.

El segundo argumento es un arreglo de objetos ServiceRecord. Un objeto ServiceRecord describe las características de un servicio Bluetooth mediante atributos, los cuales se identifican numéricamente, es decir, un servicio Bluetooth tiene una lista de pares identificador-valor que lo describen. El objeto ServiceRecord se encarga de almacenar estos pares.

Los identificadores son números enteros y los valores son objetos de tipo DataElement. Los objetos DataElement encapsulan los posibles tipos de datos mediante los cuales se pueden describir los servicios Bluetooth. Estos tipos de datos son: valor nulo, enteros de diferente longitud, arreglos de bytes, URLs, UUIDs, booleanos, strings de los tipos anteriores.

```
public void servicesDiscovered(int transID,ServiceRecord[]  
                                servRecord) {  
    ServiceRecord service = null;  
    service = servRecord[seleccion];  
    url =  
service.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT,  
                                false);  
    System.out.println("url:"+url);  
    enviar();  
}
```

**Figura 3.22** Función servicesDiscovered()

La clase **javax.bluetooth.RemoteDevice** contiene los métodos que pueden ser usados en cualquier momento para hacer una petición de cambio en las configuraciones de seguridad, o de averiguar la configuración actual de seguridad en la conexión.

### 3.2.2.3 Establecimiento de la Conexión

Una vez que la aplicación Bluetooth ya ha realizado la búsqueda de los dispositivos Bluetooth a su alcance, seleccionará al módulo Bluetooth y buscará información de sus servicios. Luego se establecerá la conexión y posteriormente se transmitirán los datos.

#### 3.2.2.3.1 *Comunicación Cliente Servidor*

El paquete `javax.bluetooth` permite usar dos mecanismos de conexión: SPP y L2CAP. En este caso se usará el perfil de puerto serial. Mediante SPP se obtiene un `DataInputStream` y un `DataOutputStream`. Para abrir una conexión se hará uso de la clase `javax.microedition.io.Connector`. En concreto se usará el método estático `open()`, su versión más sencilla requiere un parámetro que es un `String` que contendrá la URL con los datos necesarios para realizar la conexión.

La URL, necesaria para realizar la conexión, se obtiene a través del método `getConnectionURL()` de un objeto `ServiceRecord`. Un objeto `ServiceRecord` representa un servicio, es decir, una vez encontrado el servicio deseado (un objeto `ServiceRecord`), él mismo proveerá la URL necesaria para conectarse a él.

Este método requiere dos argumentos, el primero de los cuales indica si se debe autenticar y/o cifrar la conexión. Los posibles valores de este primer argumento son:

- a. `ServiceRecord.NOAUTHENTICATE_NOENCRYPT`: No se requiere ni autenticación ni cifrado.
- b. `ServiceRecord.AUTHENTICATE_NOENCRYPT`: Se requiere autenticación, pero no cifrado.
- c. `ServiceRecord.AUTHENTICATE_ENCRYPT`: Se requiere tanto autenticación como cifrado.



El segundo argumento del método `getConnectionURL()` es un booleano que especifica si nuestro dispositivo debe hacer de maestro (`true`) o bien no importa si es maestro o esclavo (`false`).

```
url = service.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
false);
```

Este código puede ser revisado en su contexto dentro de la función `servicesDiscovered()`, mostrada en la Figura 3.22.

Una vez que se tiene la URL, se utiliza el método `Connector.open()` para realizar la conexión. Este método devuelve un objeto distinto según el tipo de protocolo usado. En el caso de un cliente SPP devolverá un `StreamConnection`. A partir del `StreamConnection` se obtienen los flujos de entrada y de salida:

```
StreamConnection con = (StreamConnection) Connector.open(url);
DataOutputStream out = con.openDataOutputStream();
DataInputStream in = con.openDataInputStream();
```

Estas definiciones están incluidas como parte del método `enviar()`, que se revisará mas adelante en la Figura 3.23.

Una vez establecida la conexión ya es posible leer y escribir en ella concretando la comunicación.

#### **3.2.2.4 Transmisión de Datos**

Una vez establecida la conexión Bluetooth, se enviará un comando en forma de una cadena de texto, que será recibida e interpretada por el módulo Bluetooth. Para enviar una cadena de texto sobre una conexión SPP, se procede de la siguiente manera:

```

private void enviar(){
    StreamConnection connection = null;
    DataInputStream in = null;
    DataOutputStream out = null;
    try {
        connection = (StreamConnection)
Connector.open(url);
        in = connection.openDataInputStream();
        out = connection.openDataOutputStream();
        out.writeUTF(comando);
        out.flush();
        out.close();
        in.close();
        connection.close();
    } catch(IOException e) {
        e.printStackTrace();
    } finally{
        try {
            if(out != null)
                connection.close();
        } catch(IOException e) {}
    }
}

```

**Figura 3.23** Método enviar()

Dependiendo de la opción que se escoja en el menú, la variable “comando” tomará un valor diferente, en la Tabla 3.24, se pueden ver los valores correspondientes para cada opción del menú.

Formulario	Opción Seleccionada	Comando
Control de Acceso	Abrir	AT+ZV GPIOWrite 5 1 AT+ZV GPIOWrite 5 0
Control de Iluminación	ON	AT+ZV GPIOWrite 7 1 AT+ZV GPIOWrite 1 0
	OFF	AT+ZV GPIOWrite 7 0 AT+ZV GPIOWrite 1 1

**Figura 3.24** Posibles valores de la variable comando

#### 3.2.2.4.1 Control de Acceso

Para esta opción, la variable comando está definida de la siguiente manera:

```

comando = new String("/nAT+ZV GPIOConfig 1 o\n
AT+ZV GPIOConfig 5 o\n
AT+ZV GPIOConfig 7 o\n
AT+ZV GPIOwrite 5 1\n
AT+ZV GPIOwrite 5 0\n");

```

Además, para el control de acceso se implementa un nivel de seguridad básico por medio de una contraseña, que consiste en un cuadro de texto en el que el usuario ingresará una cadena en texto plano para que sea comparada con la que se tiene guardada: "1234", en la Figura 5.25 se puede observar el código necesario para realizar la comparación y habilitar el envío del comando para abrir la cerradura eléctrica. Además se añade una pantalla Alert para el caso de que la contraseña no se haya ingresado correctamente.

```
private void escogerOpcionAbrir(){
RemoteDevice remoteDevice2 = seleccionarDispositivo();
    try {
        System.out.println("remote
device:"+remoteDevice2.getFriendlyName(true));
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    String addr = remoteDevice2.getBluetoothAddress();
    if("1234".equals(TextField_clave.getString())){
        try { int transId =
            discoveryAgent.searchServices(ATRIBUTOS,
                SERVICIOS, remoteDevice2, this);
            form_cAcceso.append("Encontrado Dispositivo de Control " +
                remoteDevice2.getFriendlyName(true) + "; " + transId);

            form_cAcceso.append("la puerta debe estar abierta");
        } catch (Exception e) {
            Alert ClaveIncorecta = new Alert("Clave Incorrecta","", null,
                AlertType.ERROR);
        }
    }
}
```

**Figura 3.25** Función escogerOpcionAbrir()

#### 3.2.2.4.2 Control de Iluminación

Dentro del formulario Control de Iluminación, tenemos dos opciones, para encender ON y para apagar OFF. Al seleccionar la primera opción, la variable comando se llena con la siguiente cadena:

```
comando = new String("/nAT+ZV GPIOConfig 1 o\n
AT+ZV GPIOConfig 5 o\n
AT+ZV GPIOConfig 7 o\n
AT+ZV GPIOwrite 7 1\n
AT+ZV GPIOwrite 1 0\n");
```

Para la opción OFF, la variable comando tomará el siguiente valor:

```
comando = new String("/nAT+ZV GPIOConfig 1 o\n
AT+ZV GPIOConfig 5 o\n
AT+ZV GPIOConfig 7 o\n
```

```
AT+ZV GPIOwrite 7 0\n
AT+ZV GPIOwrite 1 1\n");
```

La función enviar() se ejecuta después de haber seleccionado una opción, cuando la variable “comando” se haya llenado según el caso. En esta función también se definen elementos de datos de entrada y datos de salida, DataInputStream y DataOutputStream, porque la comunicación Bluetooth se realiza en dos vías. Se transmite el texto ingresado en la variable “comando” sobre la conexión establecida y luego se cierra la conexión para que el medio quede liberado y se pueda repetir el proceso.

En el Anexo 12, se muestra el código fuente del programa completo con todos los métodos revisados en este capítulo.

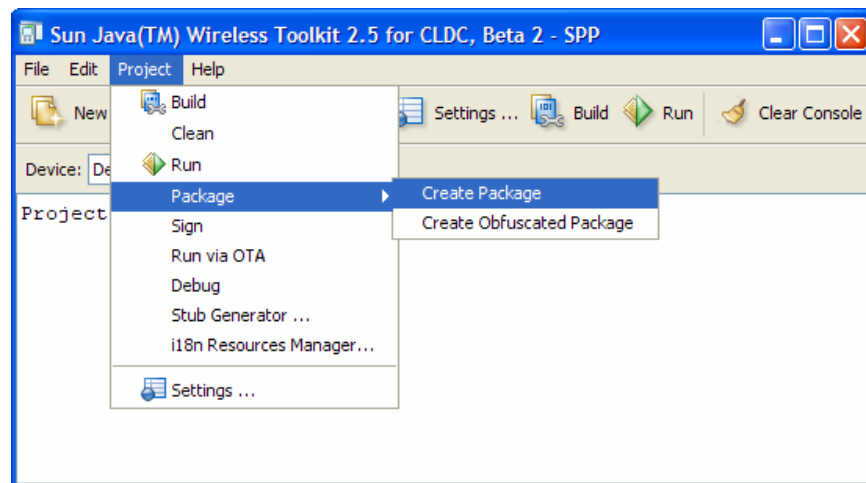
### 3.2.3 PREPARACIÓN DE LA APLICACIÓN

En la fase de **Compilación** se generó el archivo .class asociado al archivo .java y se creó el directorio del proyecto que contiene toda la información que será usada para crear un MIDlet, incluyendo el archivo de código fuente, recursos, imágenes, y un descriptor del mismo. Después de que se hayan completado los procesos de compilación y preverificación, el MIDlet pasa a la fase de Depuración y Ejecución que se debe realizar en un simulador.



**Figura 3.26** La aplicación ejecutándose en el simulador  
Cuando ya se han realizado todas las pruebas y simulaciones en el computador, la aplicación debe ser descargada al dispositivo móvil, para hacerlo es necesario que pase por el proceso de **empaquetamiento**, aquí se prepara el MIDlet para que pueda ser descargado sobre el dispositivo móvil.

Generalmente el empaquetamiento involucra a varios *MIDlets* que conformarían lo que se denomina una *suite* de *MIDlets*.



**Figura 3.27** Empaquetamiento usando Sun Java Wireless Toolkit

### 3.2.3.1 Instalación de la Aplicación en el Celular

El proceso de empaquetamiento genera los archivos JAD, JAR y manifiesto en la carpeta *bin* del proyecto cuando se usa Wireless ToolKit o en la carpeta *dist* cuando se ha trabajado con NetBeans.

El MIDlet implementado es una aplicación como muchas que se pueden instalar normalmente en los teléfonos celulares, similares a los juegos Java que están bastante difundidos actualmente.

La **instalación** consiste en copiar los archivos JAR al teléfono celular para que puedan ser ejecutados. Hay varias formas de hacerlo dependiendo de la marca y modelo del dispositivo. Se puede transferir como cualquier otra aplicación o archivo, usando infrarrojos, Bluetooth o un cable de datos desde un

computador.

Existen muchas aplicaciones útiles para administración de archivos e instalación de aplicaciones en dispositivos móviles según la marca y modelo del teléfono móvil. Por ejemplo, para teléfonos Nokia se puede usar Nokia PC Suite, para Sony Ericsson también hay una versión de PC Suite, para Motorola se puede usar P2ktools, entre otros.