7. Desarrollo de la aplicación y pruebas

7.1. Introducción

En el capítulo anterior se han descrito las pruebas diseñadas y como hay que llevarlas a cabo. Como hemos visto las pruebas consisten en la ejecución de una serie de MIDlets observando los distintos resultados. Se ha ido comentando qué es lo que hacen estas aplicaciones en cada caso particular, pero en este capítulo vamos a describirlas a fondo.

Una vez descrita la aplicación, se ejecutará y será probada. Esto se realizará sobre teléfonos de la Serie 60 de Nokia, que aunque no sean los dispositivos para los que los MIDlets están diseñados, deberán comportarse de forma bastante similar. Además en este capítulo se listará el material utilizado tanto para el desarrollo de la aplicación como para la ejecución de la misma. Pero antes de esto se proporciona una breve descripción de la Serie 40, enmarcándola dentro de las diferentes plataformas que Nokia ofrece.

7.1.1. Acerca de la Serie 40

La Serie 40 de Nokia es un conjunto de tecnologías diseñadas para permitir la creación de dispositivos con la capacidad de ejecutar aplicaciones Java MIDP y ofrecer opciones ricas en medios y contenidos. Para clarificar esto un poco más es recomendable situar la Serie 40 dentro de las distintas plataformas de desarrollo que Nokia ofrece, viendo una perspectiva de todas éstas. En la figura 7.1 se puede ver una comparativa de las distintas plataformas.

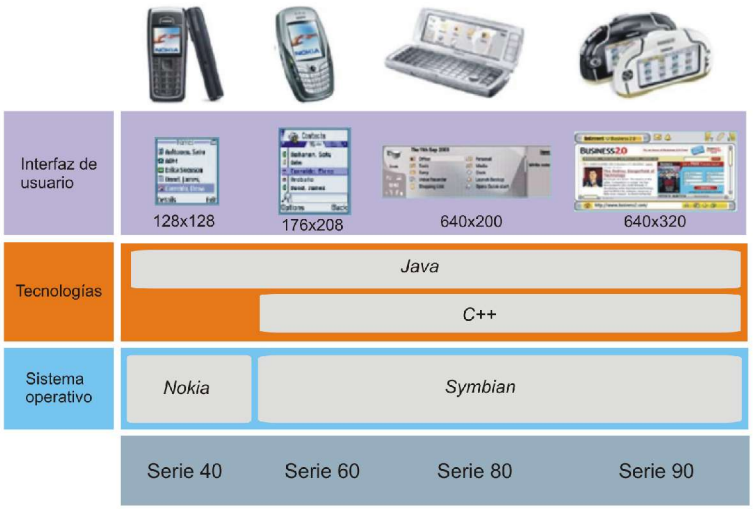


Figura 7.1 Plataformas de desarrollo Nokia

- **Serie 40:** Plataforma destinada a dispositivos móviles con tecnología Java, con sistema operativo Nokia.

- **Serie 60:** Destinada a dispositivos *smartphones* con sistema operativo Symbian, con capacidad de ejecutar aplicaciones Java.
- **Serie 80:** Para dispositivos pensados principalmente para un uso empresarial, cuyo sistema operativo es Symbian.
- **Serie 90:** Usada en dispositivos multimedia basados también en Symbian.

Como vemos la mayoría de las plataformas están implemetadas usando como sistema operativo el estándar industrial Symbian, excepto la Serie 40 que usa un sistema operativo propio de Nokia. Las plataformas con S.O. Symbian permiten el desarrollo de aplicaciones en C++, mientras que todas las plataformas (a partir de su versión 2.0) ofrecen la posibilidad de desarrollo de aplicaciones Java basadas en MIDP 2.0.

7.2.Equipo necesario

En esta sección se exponen el equipo utilizado, tanto *hardware* como *software*, para el desarrollo de la aplicación y para su prueba.

7.2.1. Hardware

- Ordenador personal tipo PC para el desarrollo de la aplicación y para instalar los MIDlets en los dispositivos.
- Prototipo Nokia 7200, que será el objeto final de los tests. Aunque no se hayan llegado a ejecutar los tests en este dispositivo, por no estar la implementación todavía a finalizada, se ha utilizado este teléfono para ir depurando los MIDlets en la medida de lo posible, exceptuando lo que a *PushRegistry* se refiere.
- Prototipo Nokia 6021, perteneciente a la Serie 40 al igual que el anterior, que se ha utilizado para el mismo fin.
- Prototipo Nokia 6801, perteneciente a la serie 60, donde ya está implementado el soporte de *PushRegistry*, que se ha utilizado para probar los MIDlets desarrollados.
- Cable DKU-2 que permite la instalación de los MIDlets en los teléfonos vía USB.
- Adaptador Bluetooth sobre USB para realizar esta instalación vía Bluetooth.
- Adaptadores internos de Nokia para instalar el *software* nativo en los dispositivos.

7.2.2. Software

- *Sun One Studio 4 update 1, Mobile Edition* como entorno de desarrollo de Java (J2ME).
- *Java 2 SDK, Standard Edition, v1.4.2* y *J2SE Runtime Environment (JRE) v5.0* necesarios para la instalación y ejecución del entorno de desarrollo *Sun One Studio*.
- *Nokia Developer´s Suite v2.2 for J2ME* que es una herramienta creada por Nokia que permite la creación de clases de aplicación y paquetes (JAR y JAD), firmar las aplicaciones e instalarlas más cómodamente en el dispositivo. Además incluye emuladores de distintos dispositivos Nokia.
- *Nokia PC suite v6.41.6* aplicación gratuita de Nokia que sirve para poder acceder al teléfono desde un PC para transferir archivos o instalar aplicaciones tipo MIDlets.
- *Phoenix* que es una herramienta interna de Nokia que se utiliza, entre otras cosas, para instalar el *software* nativo en los diferentes dispositivos.
- *Rational Rose Enterprise Edition 2003* para generar los diagramas UML.

7.3. Descripción general de la aplicación

Para conseguir una mayor sencillez a la hora de ejecutar los tests, se ha tratado de agrupar todos los tests en un solo MIDlet que cubrirá casi todos los escenarios. De este modo se ha realizado el MIDlet `PushRegistryTest`, que podrá actuar tanto de cliente como de servidor, ya que en definitiva lo que se ha de testear es un proceso de comunicaciones. Además como se vio en el capítulo anterior, se han desarrollado tres MIDlets más, pero son notablemente más sencillos. Comencemos describiendo `PushRegistryTest`.

La figura 7.2 muestra un diagrama de clases muy general del MIDlet. Como vemos éste se compone de cuatro clases principales. A continuación se describirá cada una de ellas.

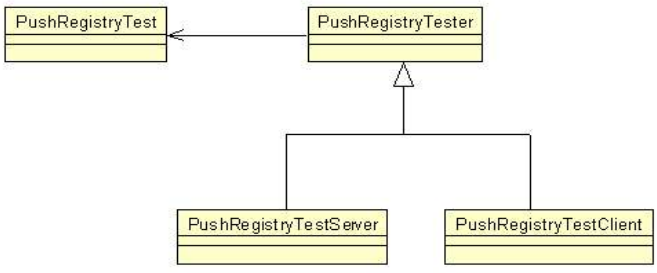


Figura 7.2 Diagrama de clases del MIDlet PushRegistryTest

7.3.1. Clase PushRegistryTest

Como vimos lo primero que hace falta a la hora de realizar un MIDlet es una clase que herede la clase `MIDlet`. Nada más iniciar el MIDlet mostrará una lista de selección, que le permitirá al usuario elegir si el dispositivo va a ser el cliente o el servidor.

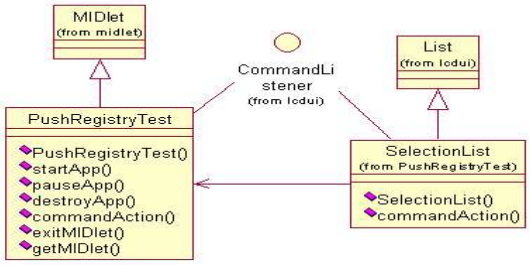


Figura 7.3 Clase PushRegistryTest

La lista de selección que se muestra será una instancia de `SelectionList` que hereda la clase `List`. `SelectionList`, que a su vez estará definida dentro de la clase `PushRegistryTest`. Además implementará la interfaz `CommandListener` para responder a los eventos generados por el usuario. Lo que se hará es iniciar en un nuevo hilo un nuevo objeto cliente o servidor según lo que se haya seleccionado. Así se podrá crear un objeto de la clase `PushRegistryTestServer` que implementa al servidor o uno de la clase `PushRegistryTestClient` que implementará al cliente.

En el siguiente listado se ve el código más importante correspondiente a la clase `PushRegistryTest`.

```
public class PushRegistryTest
    extends MIDlet
    implements CommandListener
{
...
    public void startApp()
    {
        SelectionList select = new SelectionList( this );
        display.setCurrent( select);
    }
    ...
    public class SelectionList
    extends List
    implements CommandListener
    {
        ...
        public void commandAction( Command c, Displayable d )
        {
            if( c.equals( List.SELECT_COMMAND ) ||
                ( c.getCommandType() == Command.OK ) )
            {
                int i = getSelectedIndex();
                String s = getString(i);

                if( s.equals( "Server" ) )
                {
                    Thread serverT =
                        new Thread( new PushRegistryTestServer( prObject ) );
```

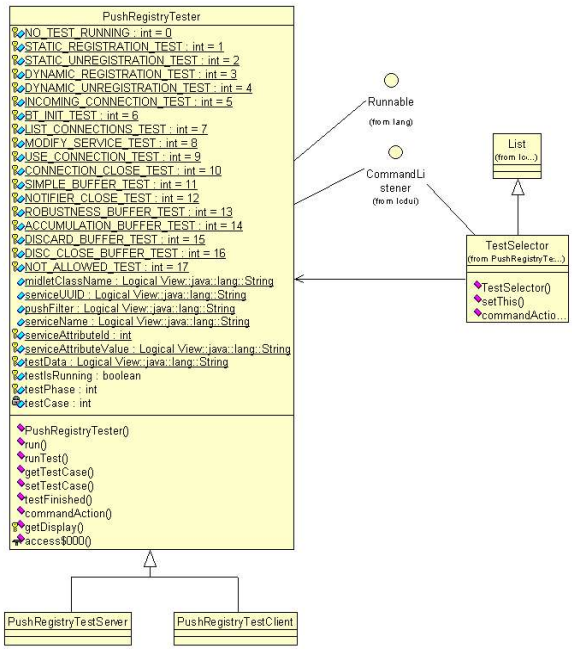
```
                    serverT.start();
                }

                if( s.equals( "Client" ) ) {
                    Thread clientT =
                        new Thread( new PushRegistryTestClient( prObject ) );

                    clientT.start();
                }
            ...
            }
        }
    }
}
```

7.3.2. Clase PushRegistryTester

El cliente y el servidor no son iguales y, por lo tanto, sus clases tampoco lo serán. Pero ambas tendrán muchas cosas en común. Por ello se ha creado la clase `PushRegistryTester`, que será heredada tanto por la clase del cliente como por la del servidor. Para empezar las dos serán hilos por lo que implementarán la interfaz `Runnable`, además de la interfaz `CommandListener` para interactuar con el usuario.



Además de esto se definirán una serie de variables y métodos a usar por ambas subclases. La mayoría son variables estáticas o constantes, como por ejemplo los identificadores de los diferentes tests, aunque también se definen algunas variables. Además de estas variables que se pueden ver en el diagrama de la figura se definen los siguientes métodos:

- `runTest()`: Con este método se iniciará un test. Como parámetro recibirá el identificador del test que se quiera llevar a cabo.

- `getTestCase()`: Devolverá el identificador del test que se está efectuando en el momento. Si no se está efectuando un test devuelve `NO_TEST_RUNNING`.
- `setTestCase()`: Establece el test a ejecutar. Tomará como parámetro el identificador del test a ejecutar.
- `getDisplay()`: Devuelve el objeto `Display` de la aplicación.
- `testFinished()`: Método al llamar al final de cada test que reinicializa las variables necesarias.

A continuación se expone el código que pueda resultar más importante de la clase `PushRegistryTester`.

```
public class PushRegistryTester
    implements Runnable, CommandListener
{
    /** Indicates that no test is running at the moment. The value is 0.
     */
    protected static final int NO_TEST_RUNNING = 0;

    /** ID of the Static Push Registration Test. The value is 1
     */
    protected static final int STATIC_REGISTRATION_TEST = 1;
    ...

    /** Name of the MIDlet to be registered in the Push Registry. By default is
     * PushRegistryTest.
     */
    public static String midletClassName = "PushRegistryTest";

    /** Service UUID to be used. By default is 2000000000001000800000057028C19
     */
    public static String serviceUUID = "2000000000001000800000057028C19";
    ...

    /** Creates a PushRegistryTester.
     * @param p PushRegistry object unique to this MIDlet.
     */
    public PushRegistryTester( PushRegistryTest p)
    {
        prObject = p;
        display = Display.getDisplay( prObject.getMIDlet() );
    }
    ...

    /** Get the test case running at the moment.
     * @return Test case ID.
     */
    public int getTestCase()
    {
        return testCase;
    }


    /** Set the test case to be performed.
     * @param testCaseID The test case ID.
     */
    public void setTestCase( int testCaseID )
    {
        testCase = testCaseID;
    }
```

```
    /** Indicates that the test case has finished. This method has to be called
     * at the end of every test case.
     */
    public void testFinished()
    {
        testCase = NO_TEST_RUNNING;
        testPhase = 0;
    }
    ...
}
```

Por último también se define una clase interna denominada `TestSelector` que no será más que una lista de selección. En ésta el usuario podrá seleccionar el tipo de test que quiera ejecutar. Una vez seleccionado se iniciara con el método `runTest()`.

```
    ...

    protected class TestSelector
        extends List
        implements CommandListener
    {

        /** Creates a TestSelector with the test cases that can be performed.
         */
        public TestSelector()
        {
            super( "Test Case:", List.IMPLICIT );

            append( "Static Unregistration Test", null );
            ...

        }


        /** Sets the TestSelector as the current Displayable and CommandListener.
         */
        public void setThis()
        {
            display.setCurrent(this);
            setCommandListener(this);
        }
        ...

        public void commandAction(Command command, Displayable displayable)
        {
            ...

                if( ( command.equals( List.SELECT_COMMAND ) ) ||
                    ( command.getCommandType() == Command.OK) )
                {

                    int i = getSelectedIndex();
                    String s = getString(i);

                    if( s.equals( "Static Unregistration Test" ) )
                    {
                        runTest( STATIC_UNREGISTRATION_TEST );
                    }
                    ...
```

7.3.3. Clase PushRegistryTestServer

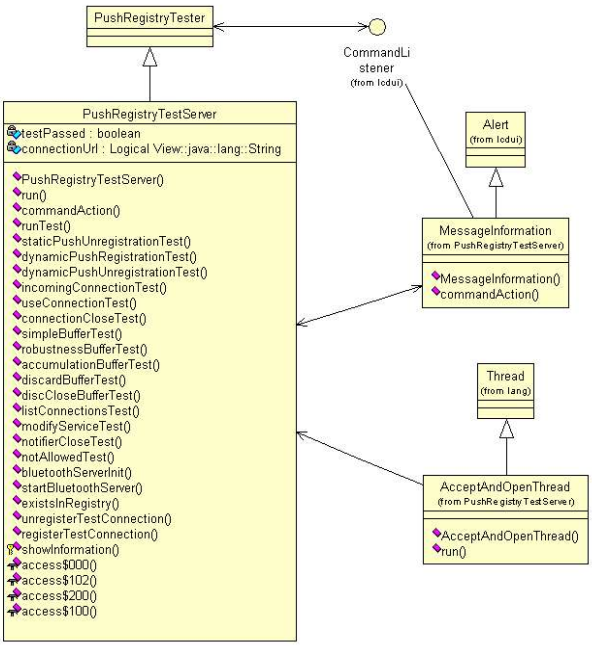


Figura 7.4 Clase PushRegistryTestServer

Esta es la clase que se utiliza para el servidor, que como hemos mencionado heredará la clase `PushRegistryTester`. Al heredarla implementa la interfaz `Runnable`, debiendo definir el método `run()` de ésta. Este método se ejecutará nada más iniciar el hilo, por lo que lo aprovecharemos entre otras cosas para iniciar distintas variables y objetos. Para comenzar se crean tres pantallas de diferentes tipos:

- Instancia de `Form` que se utilizará como pantalla para mostrar información general al usuario.
- Otra de tipo `ListSelector` definida en `PushRegistryTester` para mostrar el menú de selección de tests.
- Y una de tipo `MessageInformation` que muestra alertas al usuario para indicarle, por ejemplo, que realice una determinada acción para continuar.

Por último, también se llamará al método `bluetoothServerInit()` que inicializará el servidor Bluetooth como veremos un poco más adelante.

```
public void run()
    {
        // initialization of variables
        connectionUrl = "btspp://localhost:2000000000001000800000 6057028C19;name="
                        + serviceName;
        ...

        // creates the form of the server
        testerForm = new Form( "Test Server" );
        testerForm.addCommand( new Command( "Exit", Command.EXIT, 1 ) );
```

```
        testerForm.setCommandListener( this );
        getDisplay().setCurrent( testerForm );

        // creates an Alert to show information to the user
        info = new MessageInformation( "" );

        // initialization of Bluetooth System
        bluetoothServerInit();

        // creates a TestSelector, which is a List for selecting the test case
        testSelector = new TestSelector();

        ...
    }
```

Además de `Runnable` también implementa la interfaz `CommandListener` por lo que deberá sobreescribir el método `CommandAction()`, que además de responder a un comando de tipo `EXIT` para salir de la aplicación, responderá a los comandos con etiquetas `"Test Case"` que muestra la lista con los distintos tests para elegir, y `"Clear Log"` que borra todo lo que haya escrito en la pantalla de tipo `Form`.

```
public void commandAction( Command c, Displayable d )
    {
        if( c.getCommandType() == Command.EXIT )
        {
            prObject.exitMIDlet();
        }
        else
        {
            if( c.getLabel() == "Test Case" )
            {
                testSelector.setThis();
            }
            if( c.getLabel() == "Clear Log" )
            {
                testerForm.deleteAll();
            }
        }
    }
```

El método `runTest()` de `PushRegistryTester` será sobreescrito. En función del identificador del test que recibe el método como parámetro, se llamará al método que lleva a cabo el tests en cuestión.

```
public void runTest(int tC) {
...
    switch(tC) {
        case STATIC_UNREGISTRATION_TEST:
            staticPushUnregistrationTest();
            break;
        case DYNAMIC_REGISTRATION_TEST:
            dynamicPushRegistrationTest();
            break;
        ...
    }
}
```

Por lo que vemos, en general a cada test le corresponde un método en la clase que implementa al servidor. Cada método realizará lo que vimos en la descripción de cada test. A modo de resumen en la tabla 7.1 se especifica que es lo que realiza cada método.

Método	Descripción
`staticPushUnregistrationTest()`	- Comprueba que la conexión no exista en el *Push Registry.*
`dynamicPushRegistrationTest()`	- Registra la conexión con un nombre falso para el MIDlet, para comprobar que se lanza ClassNotFoundException. - Realiza un registro válido de conexión. - Comprueba que la conexión exista en el *Push Registry.* - Vuelve a registrar la conexión para comprobar que se lanza IOException.
`dynamicPushUnregistrationTest()`	- Elimina el registro de la conexión de forma válida. - Comprueba que la conexión no exista en el *Push Registry.* - Vuelve a eliminar el registro para comprobar que el método devuelve "false".
`incomingConnectionTest()`	- Este método no hace nada pues en este test no es necesaria ninguna acción por parte del MIDlet del servidor.
`useConnectionTest()`	- Abre la conexión llamando a Connector.open(). - Pregunta al usuario si llamar a notifier.acceptAndOpen() y actúa en consecuencia.
`connectionCloseTest()`	- Espera la conexión por parte de clientes llamando a acceptAndOpen(). - Cuando el cliente se conecte se cerrará la conexión y este método volverá a llamar a acceptAndOpen() para esperar una nueva conexión.
`simpleBufferTest()`	- Espera la conexión por parte de clientes llamando a acceptAndOpen().
`robustnessBufferTest()`	- Espera la conexión por parte de clientes llamando a acceptAndOpen().
`accumulationBufferTest()`	- Espera la conexión por parte de dos clientes llamando dos veces a acceptAndOpen().
`discardBufferTest()`	- Espera la conexión por parte de clientes llamando a acceptAndOpen().
`discCloseBufferTest()`	- Espera la conexión por parte de clientes llamando a acceptAndOpen().
`listConnectionsTest()`	- Comprueba que el método PushRegistry.listConnections(true) devuelva una lista vacía pues no hay ninguna conexión activa. - Pide al usuario que ejecute el test del cliente. - Vuelve a hacer lo mismo que en el primer paso, pero ahora se debe devolver una conexión.
`modifyServiceTest()`	- Se recupera el *Service Record* del servicio. - Se le añade un atributo. - Se actualiza el *Service Record* de la SDDB.
`notifierCloseTest()`	- Espera la conexión por parte de clientes llamando a acceptAndOpen().
`notAllowedTest()`	- Se registra la conexión con un filtro que impide la conexión por parte del cliente.

Tabla 7.1 Métodos asignados a cada test en el servidor

En el listado siguiente se ve cómo es el código de algunos de estos métodos. No se incluyen todos por el parecido que hay en ocasiones entre ellos. Destacar además que la conexión del cliente se acepta desde otro hilo. Esto implica que una vez establecida, el desarrollo del test se continuará en este nuevo hilo. Esto se verá con más detalle un poco más adelante.

```
    public void staticPushUnregistrationTest()
    {
        ...

        // check whether the connection was deleted from Push Registry
        if(existsInRegistry()) {
            testPassed = false;
            testerForm.append("\nEntry exists already in Push Registry");
        } else {
            testerForm.append("\nEntry was deleted from Push Registry");
        }

        if(testPassed) {
            testerForm.append("\nRun Client Test");
        } else {
            testerForm.append("\nTest FAILED");
        }

        // test finished
        testFinished();
    }

    public void dynamicPushRegistrationTest()
    {
        ...

        // check whether ClassNotFoundException is thrown
        testerForm.append("\nChecking ClassNotFoundException...");
        try {
            PushRegistry.registerConnection(connectionUrl, "falseClassName", "*");
            testPassed = false;
        } catch(ClassNotFoundException e) {
            testerForm.append("\nOK");
        } catch(IOException e) {
            testerForm.append("\nERROR: IOException");
            testPassed = false;
        }

        // register connection
        if(registerTestConnection(false) == false) {
            testPassed = false;
        }

        // check whether an entry in the Push Registry was created
        if(existsInRegistry()) {
            testerForm.append("\nEntry created in Push Registry");
        } else {
            testPassed = false;
            testerForm.append("\nCannot find the entry in Push Registry");
        }

        // check whether IOException is thrown
        testerForm.append("\nChecking IOException...");
        try {
            PushRegistry.registerConnection(connectionUrl, midletClassName, "*");
            testPassed = false;
        } catch(ClassNotFoundException e) {
            testerForm.append("\nERROR: ClassNotFoundException");
            testPassed = false;
        } catch(IOException e) {
            testerForm.append("\nOK");
        }
        ...

    }


    /** Performs the Dynamic Push Unregistration Test.
     */
    public void dynamicPushUnregistrationTest()
```

```
    {
        ...

        // check whether unregisterConnection() returns true when the connection
        // is registered
        if(unregisterTestConnection() == false) {
            testPassed = false;
        }

        // check whether the connection was deleted from Push Registry
        if(existsInRegistry()) {
            testPassed = false;
            testerForm.append("\nEntry exists already in Push Registry");
        } else {
            testerForm.append("\nEntry was deleted from Push Registry");
        }

        // check whether unregisterConnection() returns false when the connection
        // is already unregistered
        if(unregisterTestConnection() == true) {
            testPassed = true;
        }
        ...

    }

    public void useConnectionTest()
    {
        switch( testPhase )
        {
            case 0:
                ...

                try
                {
                    notifier =
                        (StreamConnectionNotifier)Connector.open( connectionUrl );
                }
                catch ( IOException e )
                {
                    testerForm.append( "\nConnector.open failed: IOException" );
                    testPassed = false;
                }

                showInformation( "Call AcceptAndOpen" );

                testPhase ++;
                break;
            case 1:

                testerForm.append( "\nWaiting on acceptAndOpen... " );
                AcceptAndOpenThread t = new AcceptAndOpenThread();
                t.start();
                ...

        }
    }

    public void connectionCloseTest()
    {
        switch( testPhase )
        {
            case 0:
                ...

                testPhase ++;
                startBluetoothServer();

                break;
            case 1:
```

```
                testPhase ++;

                testerForm.append( "\nWaiting on acceptAndOpen... " );
                AcceptAndOpenThread t = new AcceptAndOpenThread();
                t.start();
                break;
            case 2:
                testFinished();
        }
    }

    public void simpleBufferTest()
    {
        ...

        startBluetoothServer();
    }

    public void accumulationBufferTest()
    {
        ...

        startBluetoothServer();

        // start another acceptAndOpenThread
        testerForm.append( "\nWaiting on acceptAndOpen... " );
        AcceptAndOpenThread t = new AcceptAndOpenThread();
        t.start();
    }

    public void listConnectionsTest()
    {
        switch(testPhase)
        {
            case 0:
                ...

                // no incoming connection
                if((PushRegistry.listConnections(true)).length > 0)
                {
                    testPassed = false;
                    testerForm.append("\nConnection exists in PushRegistry:
                                       FAILED");
                }
                else
                {
                    testerForm.append("\nConnection does not exist in PushRegistry:
                                       OK");
                }

                if(testPassed)
                {
                    showInformation("Run Client Test");
                    testerForm.append("\nRun Client Test");
                }

                testPhase ++;
                break;
            case 1:
                // incoming connection
                if((PushRegistry.listConnections(true)).length > 0)
                {
                    testerForm.append("\nConnection exists in PushRegistry: OK");
                }
                else
                {
                    testPassed = false;
                    testerForm.append("\nConnection does not exist in PushRegistry:
                                       FAILED");
                }
```

```
                getDisplay().setCurrent(testerForm);
                testFinished();
        }
    }

    public void modifyServiceTest()
    {
        ServiceRecord localRecord;
        DataElement element;

        element = new DataElement( DataElement.STRING, serviceAttributeValue );
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nModify Service Test:\n" );

        try
        {
            notifier = ( StreamConnectionNotifier )Connector.open( connectionUrl );

            localRecord = local.getRecord( notifier );

            localRecord.setAttributeValue( serviceAttributeId, element );

            try
            {
                local.updateRecord( localRecord );
            }
            catch ( ServiceRegistrationException e )
            {
                testerForm.append( "\nSDDB could not be updated successfully" );
                testPassed = false;
            }

            notifier.close();
        }
        catch ( IOException e )
        {
            testerForm.append( "\nConnector failed: IOException" );
            testPassed = false;
        }

        testerForm.append( "\nRun Client Test" );

        testFinished();
    }

    public void notAllowedTest()
    {
        testPassed = true;
        String filter = "0a00112e62a7";  // aleatory BT address
        ...

        try
        {
            PushRegistry.registerConnection(connectionUrl, midletClassName,
                                            filter);
        }
        catch (IOException e)
        {
            testerForm.append("\nConnection already registered or there are
                        insufficient resources to handle the registration");
            testPassed = false;
        }
        catch (ClassNotFoundException e)
        {
            testerForm.append("\nMIDlet class name can not be found in the current
                        MIDlet suite");
            testPassed = false;
        }
```

```
        if(testPassed == true) {
            testerForm.append("\nExit application and run client test");
        } else {
            testerForm.append("\nTest FAILED");
        }

        // test finished
        testFinished();
    }
```

Además de los métodos vistos hasta ahora, en la clase `PushRegistryTestServer` se definen los siguientes:

- `bluetoothServerInit()`: Inicializará el sistema Bluetooth del servidor, recuperando el objeto `LocalDevice`.
- `startBluetoothServer()`: Llamando a este método se consigue que el servidor comience a recibir conexiones por parte del cliente. Lo que hace es llamar a `Connector.open()` y `acceptAndOpen()` en un nuevo hilo.
- `existInRegistry()`: Comprueba si la conexión existe en el *Push Registry* mediante una llamada a `PushRegistry.listConnections()`.
- `unregisterTestConnection()`: Elimina el registro de la conexión llamando al método `PushRegistry.unregisterConnection()`. Además se captura la `SecurityException` que este puede lanzar.
- `registerTestConnection()`: Realiza el registro de la conexión dinámicamente llamando a `PushRegistry.registerConnection()`. Se capturan las excepciones del tipo `IOException` y `ClassNotFoundException` que este método puede lanzar.
- `showInformation()`: Muestra un mensaje en la pantalla de tipo `MessageInformation`.

```
    public boolean bluetoothServerInit()
    {
        ...

        try
        {
            local = LocalDevice.getLocalDevice();
        }
        catch ( BluetoothStateException e )
        {
            ...
        }
        ...
    }

    public void startBluetoothServer()
    {
        try
        {
            notifier = ( StreamConnectionNotifier )Connector.open( connectionUrl );
        }
        catch ( IOException e )
```

```
        {
            ...
        }

        AcceptAndOpenThread t = new AcceptAndOpenThread();
        t.start();
    }

    public boolean existsInRegistry()
    {
        ...

        connections = PushRegistry.listConnections(false);

        if(connections.length > 0)
        {
            for(int i = 0; i < connections.length; i++)
            {
                if(connections[i].equals(connectionUrl))
                {
                    existsInRegistry = true;
                }
                ...
            }
        }
        ...
    }

    public boolean unregisterTestConnection()
    {
        ...
        try
        {
            if(unregisterResult = PushRegistry.unregisterConnection(connectionUrl))
            {
                testerForm.append("\nPush Connection Unregistration: OK");
                unregisterResult = true;
            }
            ...
        }
        catch (SecurityException e)
        {
            ...
        }
        ...
    }

    public boolean registerTestConnection(boolean security)
    {
        ...
        try
        {
            if(security)
            {
                PushRegistry.registerConnection(connectionUrlSecurity,
                                                midletClassName, pushFilter);
            }
            else
            {
                PushRegistry.registerConnection(connectionUrl, midletClassName,
                                                pushFilter);
            }
            registerResult = true;
        }
        catch (IOException e)
        {
            ...
        }
        catch (ClassNotFoundException e)
        {
```

```
            ...
        }
        ...
    }

    protected void showInformation( String infoString )
    {
        info.setString( infoString );

        if( getDisplay().getCurrent() != info )
        {
            getDisplay().setCurrent( info );
        }
    }
```

Hasta aquí hemos visto todos los métodos que se incluyen en esta clase. A continuación se describen las dos clases internas que en ella también se definen:

- `MessageInformation`: Como anteriormente hemos citado esta clase hereda `Alert` por lo que será una pantalla de tipo alerta. Como constructor utiliza el de la superclase. Una vez creado el objeto, se le añaden dos comandos, de forma que la alerta se convierte en un dialogo modal, es decir que no desaparecerá hasta que el usuario seleccione alguno de los dos.
  En el método `commandAction()` se actúa según lo que el usuario haya seleccionado:
  - `CANCEL`: Se vuelve a la pantalla de tipo `Form`. Además se entiende que el usuario no desea continuar con el test, por lo que se llama a `testFinished()`.
  - `OK`: Se continúa con el test que se estaba ejecutando, que se puede saber llamando a `getTestCase()`.

```
    protected class MessageInformation
        extends Alert
        implements CommandListener
    {
        public MessageInformation( String msg )
        {
            super( "", msg, null, AlertType.INFO );
            addCommand( new Command( "Cancel", Command.CANCEL, 1 ) );
            addCommand( new Command( "OK", Command.OK, 1 ) );
            setCommandListener( this );
        }

        public void commandAction( Command c, Displayable d )
        {

            if( c.getCommandType() == Command.CANCEL )
            {
                getDisplay().setCurrent( testerForm );
                testFinished();
            }
            if( c.getCommandType() == Command.OK )
            {
                switch( getTestCase() )
                {
                    case LIST_CONNECTIONS_TEST:
                        listConnectionsTest();
                        break;
                    ...
                }
```

```
            }
        }
    }
```

- `AcceptAndOpenThread()`: El método `acceptAndOpen()` bloquea la aplicación hasta que se reciba la conexión por parte de un cliente. Por eso se decide llamarlo desde este nuevo objeto que se ejecuta en un hilo independiente. Como único método aparte del constructor, posee el método `run()` heredado de la clase `Thread`. Aquí lo que se hace es llamar a `acceptAndOpen()` y una vez recibida la conexión del cliente, continuar según el test que se esté ejecutando.

```
    protected class AcceptAndOpenThread
        extends Thread
    {
        ...
        public void run()
        {
            ...
            try
            {
                connection = ( StreamConnection )notifier.acceptAndOpen();

                testerForm.append( "Connected!" );

                switch( getTestCase() )
                {
                    case USE_CONNECTION_TEST:
                        ...
                        break;
                    case CONNECTION_CLOSE_TEST:
                        ...
                        break;
                    ...
        }
    }
```

7.3.4. Clase PushRegistryTestClient

Esta clase es la que utiliza el cliente, que al igual que la del servidor heredará `PushRegistryTester`. De este modo también tendrá que implementar las interfaces `Runnable` y `CommandListener`.

El método `run()` de la primera de éstas se aprovecha para obtener el objeto `Display` del `MIDlet`, y para llamar a `bluetoothClientInit()` que inicializa el cliente Bluetooth. Además si esto se ha realizado con éxito se comenzará el proceso de selección del servidor a utilizar llamando a `selectTestServer()`.

```
public void run() {

        display = getDisplay();

        if( bluetoothClientInit() ) {
```

```
            selectTestServer();
        }
    }
```

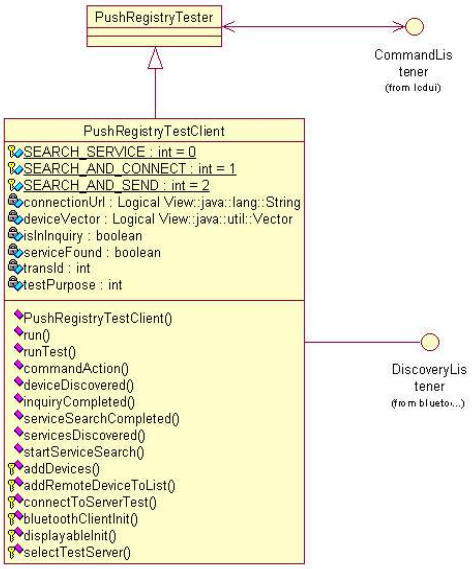


Figura 7.5 Clase PushRegistryTestClient

El método `selectTestServer()`, llamado desde `run()` nada más comenzar la ejecución del cliente, mostrará una lista donde aparecerán los diferentes dispositivos del entorno para poder seleccionar entre ellos al servidor. Primeramente se crea la pantalla de tipo `List` donde se muestran los dispositivos al usuario, y un objeto de tipo `Vector` para ir almacenando los dispositivos como como objetos `RemoteDevice` para después poder acceder a ellos. La adición de dispositivos tanto a la lista como al vector lo realiza el método `addRemoteDeviceToList()`. A continuación se llama al método `addDevices()` que añade los dispositivos preconocidos y de la *caché*. Por último se inicia un *inquiry* para comenzar el descubrimiento de dispositivos del entorno.

```
    protected void selectTestServer() {
        deviceVector = new Vector();
        deviceList = new List( "Select server", List.IMPLICIT );
        ...

        addDevices();

        try {
            agent.startInquiry( DiscoveryAgent.GIAC, this );
        }
        catch ( BluetoothStateException e ) {
            ...
        }

        isInInquiry = true;
    }

    protected void addDevices() {
        RemoteDevice[] list = agent.retrieveDevices( DiscoveryAgent.PREKNOWN );
```

```
        if( list != null ) {
            for( int i = 0; i < list.length; i++ ) {
                addRemoteDeviceToList( list[i] );
            }
        }

        list = agent.retrieveDevices( DiscoveryAgent.CACHED );

        if( list != null ) {
            for( int i = 0; i < list.length; i++ ) {
                addRemoteDeviceToList( list[i] );
            }
        }
    }

    protected void addRemoteDeviceToList( RemoteDevice d ) {
        String address = d.getBluetoothAddress();

        deviceList.insert( 0, address, null );
        deviceVector.insertElementAt( d, 0 );
    }
```

Para responder a los eventos producidos como resultado de la búsqueda de dispositivos y de servicios que se realizará más tarde, `PushRegistryTestClient` implementará la interfaz `DiscoveryListener`. De este modo cuando se encuentre un nuevo dispositivo se producirá una llamada al método `deviceDiscovered()`, donde se añadirá el dispositivo a la lista. Cuando el *inquiry* finalice se llamará a `inquiryCompleted()` donde se verá por qué ha finalizado.

```
    public void deviceDiscovered( RemoteDevice remoteDevice, DeviceClass deviceClass ) {
        addRemoteDeviceToList( remoteDevice );
    }

    public void inquiryCompleted( int type ) {
        isInInquiry = false;
        ...

        if( type != DiscoveryListener.INQUIRY_COMPLETED ) {
            if( type == DiscoveryListener.INQUIRY_TERMINATED ) {
                return;
            }
            else {
                // Inquiry failed
                ...
            }
        }
        else {
            ...
        }
        ...
    }
```

La interfaz `CommandListener` proporciona el método `commandAction()` que se deberá implementar, actuando según los eventos producidos por el usuario. Habrá cuatro tipos de comandos que pueden recibirse:

- `Command.EXIT` que podrá tener lugar en cualquier momento.

- Comandos con etiqueta "Test Case" y "Clear Log" que se podrán recibir cuando la pantalla activa sea testerForm.

- List.SELECT_COMMAND que se podrá recibir cuando la pantalla activa en el momento sea una lista. Esto se producirá durante el proceso de descubrimiento de dispositivos.

La tabla 7.2 muestra las distintas alternativas que existen y las acciones que se llevan a cabo en commandAction().

Comando	**Acción**		
EXIT	Si se están buscando dispositivos	Se detiene la búsqueda	
	Si no	Se cierra el MIDlet	
"Test Case"	Se muestra menú de selección de tests		
"Clear Log"	Se borra el contenido de TesterForm		
SELECT_COMMAND	Si se están buscando dispositivos	Se detiene la búsqueda	- Se inicializa y se muestra la pantalla TesterForm - Se obtiene dispositivo seleccionado
	Si no		

Tabla 7.2 commandAction() en PushRegistryTestClient

```
    public void commandAction( Command command, Displayable displayable ) {
        if( command.getCommandType() == Command.EXIT ) {
            if( isInInquiry ) {
                agent.cancelInquiry( this );
            }
            else {
                prObject.exitMIDlet();
            }
        }
        else {
            if( command.getLabel() == "Test Case" ) {
                testSelector.setThis();
            }
            if( command.getLabel() == "Clear Log" )  {
                testerForm.deleteAll();
            }
        }

        if( command == List.SELECT_COMMAND ) {

            display.setCurrent( testerForm );

            if( isInInquiry ) {
                agent.cancelInquiry( this );
            }

            serverDevice = (RemoteDevice)deviceVector.elementAt(deviceList.getSelectedIndex());
            displayableInit();
            ...

        }
    }
```

Como vemos si el usuario selecciona “Test Case” se mostrará el menú de selección de tests donde se elegirá el que se desea ejecutar. Es entonces cuando se inicia el test mediante el método `runTest()`, que será sobreescrito en esta clase.

El cliente lo que realiza en cada test es buscar el servicio en el dispositivo servidor. En algunos además se conecta a él y en otros incluso le envía datos. Esto es lo indicará la variable `testPurpose` que podrá tomar los valores siguientes:

- `SEARCH_SERVICE`: El test buscará el servicio en el servidor.
- `SEARCH_AND_CONNECT`: Buscará el servicio y se conectará a él.
- `SEARCH_AND_SEND`: Buscará el servicio, se conectará a él y enviará datos.

En la tabla 7.3 se muestra cuales son los tests de cada tipo.

Tipo	*Tests*
SEARCH_SERVICE	STATIC_UNREGISTRATION_TEST DYNAMIC_REGISTRATION_TEST DYNAMIC_UNREGISTRATION_TEST MODIFY_SERVICE_TEST
SEARCH_AND_CONNECT	INCOMING_CONNECTION_TEST USE_CONNECTION_TEST CONNECTION_CLOSE_TEST LIST_CONNECTIONS_TEST NOTIFIER_CLOSE_TEST NOT_ALLOWED_TEST
SEARCH_AND_SEND	SIMPLE_BUFFER_TEST ROBUSTNESS_BUFFER_TEST ACCUMULATION_BUFFER_TEST DISCARD_BUFFER_TEST DISC_CLOSE_BUFFER_TEST

Tabla 7.3 Tipos de tests en el cliente

El método `runTest()` lo que hace es actualizar la variable `testPurpose` según el indicador de test recibido como parámetro, para seguidamente comoenzar la búsqueda del servicio.

```
public void runTest( int tC ) {

    setTestCase( tC );
    ...
    switch( getTestCase() ) {
        case STATIC_UNREGISTRATION_TEST:
            testerForm.append( "\n\nStatic Unregistration Test\n\n" );
            testPurpose = SEARCH_SERVICE;
            break;
        ...
        case INCOMING_CONNECTION_TEST:
            testerForm.append( "\n\nIncoming Connection Test\n\n" );
            testPurpose = SEARCH_AND_CONNECT;
            break;
        ...
        case SIMPLE_BUFFER_TEST:
            testerForm.append( "\n\nSimple Buffer Test\n\n" );
            testPurpose = SEARCH_AND_SEND;
```

```
                break;
            ...
        }

        startServiceSearch();
    }
```

En el método `startServiceSearch()` se configura la búsqueda de servicios y después se inicia con el método `searchServices()`. Normalmente no se recuperará ningún atributo del *Service Record* en especial, excepto en el caso del test "Actualización del *Service Record*", en el que se recuperará el atributo previamente añadido por el servidor.

```
    public void startServiceSearch() {
        try {
            UUID[] uuidList = new UUID[1];
            uuidList[0] = new UUID( serviceUUID, false );

            if(getTestCase() == MODIFY_SERVICE_TEST) {
                int[] attrList = new int[1];
                attrList[0] = serviceAttributeId;

                transId = agent.searchServices( attrList, uuidList, serverDevice, this );
            } else {
                transId = agent.searchServices( null, uuidList, serverDevice, this );
            }
        }
        catch ( BluetoothStateException e ) {
            testerForm.append( "\nUnable to start service search" );
        }
    }
```

Al igual que en la búsqueda de dispositivos, los servicios que se van encontrando se devuelven como eventos, en los que se llamará al método `servicesDiscovered()`. En éste se guardará el `ServiceRecord` recuperado y se obtendrá a partir de él la cadena de conexión del servidor. Aquí habrá tests que finalicen su ejecución, pues su objetivo será el de encontrar o no el servicio en el dispositivo remoto. Por último se detendrá la búsqueda de servicios mediante el método `cancelServiceSearch()`.

```
public void servicesDiscovered( int transID, ServiceRecord[] record )
    {
        remoteRecord = record[0];
        serviceFound = true;
        connectionUrl = remoteRecord.getConnectionURL( 0, false );

        switch( getTestCase() )
        {
            case STATIC_REGISTRATION_TEST:
                testerForm.append( "\nTEST PASSED\n" );
                testFinished();
                break;
            case STATIC_UNREGISTRATION_TEST:
                testerForm.append( "\nTEST FAILED\n" );
                break;
            case DYNAMIC_REGISTRATION_TEST:
                testerForm.append( "\nTEST PASSED\n" );
                testFinished();
                break;
            case DYNAMIC_UNREGISTRATION_TEST:
                testerForm.append( "\nTEST FAILED\n" );
                testFinished();
```

```
                break;
        }

        agent.cancelServiceSearch( transId );
    }
```

Al detener la búsqueda de servicios se producirá un evento que llamará a `serviceSearchCompleted()`. En este método también finalizarán los tests cuyo objetivo era el determinar si el servicio era visible o no. Además en el caso del test "Actualización del *Service Record*" se comprobará si el atributo a recuperar se ha obtenido correctamente. Por otro lado en el caso de los tests de tipo `SEARCH_AND_CONNECT` y `SEARCH_AND_SEND` se procede a establecer la conexión con el servidor, para lo que se utiliza el método `connectToServer()`.

```
    public void serviceSearchCompleted( int transID, int type )
    {
        switch( type )
        {
            case SERVICE_SEARCH_COMPLETED:
                if( !serviceFound )
                {
                    ...
                    switch( getTestCase() )
                    {
                        case STATIC_REGISTRATION_TEST:
                            testerForm.append( "\nTEST FAILED\n" );
                            break;
                        ...
                    }

                    testFinished();
                }
                break;
            case SERVICE_SEARCH_TERMINATED:
                if( !serviceFound )
                {
                    testerForm.append( "\nService not found" );
                    testFinished();
                }
                else
                {
                    switch( testPurpose )
                    {
                        case SEARCH_SERVICE:
                            if( getTestCase() == MODIFY_SERVICE_TEST )
                            {
                                String attributeString;
                                DataElement attributeElement;

                                if( remoteRecord != null )
                                {
                                    if( ( attributeElement = remoteRecord.getAttributeValue
                                         ( serviceAttributeId ) ) != null )
                                    {
                                        if( ( attributeString = ( String )
                                         attributeElement.getValue() ).length() > 0 )
                                        {
                                            if(attributeString.equals(serviceAttributeValue))
                                            {
                                                testerForm.append( "\nTest PASSED!\n" );
                                            }
                                            ...
                                break;

                            case SEARCH_AND_CONNECT:
                                connectToServerTest();
                                ...
```

```
                        break;
                    case SEARCH_AND_SEND:
                        connectToServerTest();
                        ...
                        break;
                }
            }
            break;
        case SERVICE_SEARCH_ERROR:
            testerForm.append( "\nAn error occurred while processing the request" );
            ...
            break;
        case SERVICE_SEARCH_NO_RECORDS:
            switch( getTestCase() )
            {
                case STATIC_REGISTRATION_TEST:
                    testerForm.append( "\nTEST FAILED\n" );
                    break;
                ...
            }
            testFinished();
            break;
        case SERVICE_SEARCH_DEVICE_NOT_REACHABLE:
            testerForm.append( "\nCould not establish a connection to the remote device" );
            ...
            testFinished();
            break;
    }
}
```

El método `connectToServer()` será el que se encargue de establecer la conexión con el servidor. Para ello utilizará el método `Connector.open()` con la cadena de conexión previamente obtenida. Además en el caso de los tests de tipo `SEARCH_AND_SEND` se enviarán los datos que sean necesarios. Por último se cerrarán los flujos de datos y las conexiones.

```
protected void connectToServerTest()
{
    OutputStream output;
    byte[] data;

    try
    {
        connection = ( StreamConnection )Connector.open( connectionUrl );

        if( testPurpose == SEARCH_AND_SEND )
        {
            output = connection.openOutputStream();

            switch( getTestCase() )
            {
                case SIMPLE_BUFFER_TEST:
                    data = testData.getBytes();

                    output.write( data );
                    output.flush();

                    break;

                case ROBUSTNESS_BUFFER_TEST:
                    ...
            output.close();
        }

        // close connection
```

```
            connection.close();

        } catch( IOException e ) {
            testerForm.append( "Connector.open: IOException\n" );
        }
    }
```

Hasta aquí hemos visto en que consiste la clase `PushRegistryTestClient` y con ello las cuatro clases que componen el MIDlet `PushRegistryTest`. A continuación se describen los otros tres sencillos MIDlets que se han desarrollado.

7.3.5. MIDlet StaticRegistrationTest

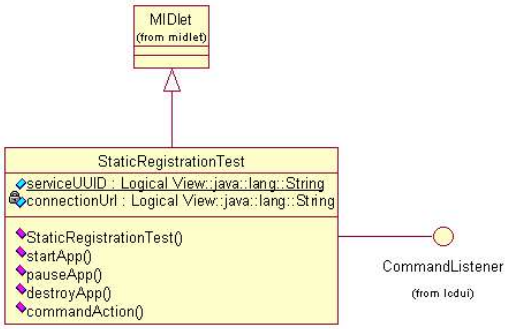


Figura 7.6 Diagrama de StaticRegistrationTest

Es un MIDlet muy sencillo que comprueba que haya una entrada en el *Push Registry*. Lo más interesante de este MIDlet sería el archivo JAD que es donde se realiza el registro *push* de forma estática, gracias al atributo `MIDlet-Push` que posee. Su contenido sería el siguiente:

```
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
MIDlet-Jar-URL: StaticRegistrationTest.jar
MIDlet-Name: StaticRegistrationTest
MIDlet-Vendor: cgr
MIDlet-Version: 1.0
MIDlet-Jar-Size: 3305
MIDlet-Push-1: btspp://:2000000000001000800000006057028c19, StaticRegistrationTest, *
MIDlet-1: StaticRegistrationTest, , StaticRegistrationTest
```

En el código vemos como se comprueba la existencia de la conexión en el *Push Registry* mediante el método `PushRegistry.listConnections(false)`.

```
public class StaticRegistrationTest
    extends MIDlet
    implements CommandListener
{

    public static String serviceUUID = "2000000000001000800000006057028c19";
    ...

    public void startApp()
    {
        String[] connections = null;
```

```
        connectionUrl = "btspp://:" + serviceUUID;

        connections = PushRegistry.listConnections(false);

        // check whether an entry in the Push Registry was created
        if(connections.length > 0)
        {
            for(int i = 0; i < connections.length; i++)
            {
                if(connections[i].equals(connectionUrl))
                {
                    form.append("\nEntry created in Push Registry");
                    form.append("\nTest PASSED");
                }
        ...
```

7.3.6. MIDlet PushUnregistrationTest

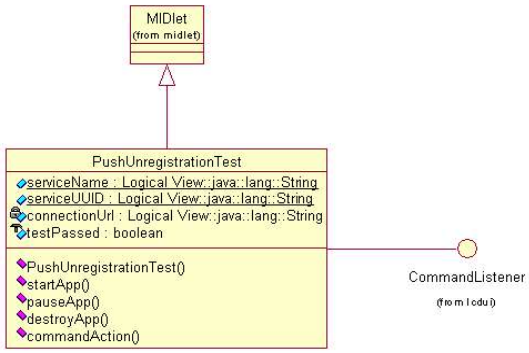


Figura 7.7 Diagrama de PushUnregistrationTest

Este MIDlet lo que realiza es tratar de eliminar la conexión que está registrada. Como el registro fue realizado por otro MIDlet debe de lanzarse una excepción del tipo `SecurityException`.

```
public class PushUnregistrationTest
    extends MIDlet
    implements CommandListener {

    public static String serviceName = "PushService";
    public static String serviceUUID = "2000000000001000800000606057028C19";
    ...
    public void startApp() {
        // initialization of variables
        connectionUrl = "btspp://localhost:" + serviceUUID + ";name=" + serviceName;
        testPassed = true;
        ...
        try {
            PushRegistry.unregisterConnection(connectionUrl);
            testPassed = false;
        } catch(SecurityException e) {
            form.append("\nSecurityException OK");
            testPassed = true;
        }
        ...
```

7.3.7. MIDlet PushConnectorOpenTest

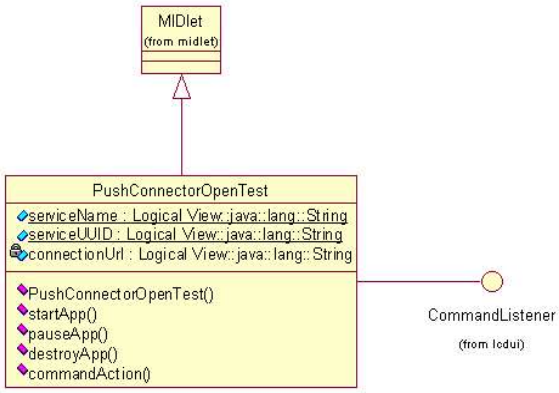


Figura 7.8 Diagrama de PushConnectorOpenTest

Este MIDlet al igual que los dos anteriores es bastante sencillo. Su función será la de abrir la conexión que ha sido previamente registrada por otro MIDlet para comprobar que se lanza una `IOException`. Por lo demás es bastante parecido al anterior.

```
public class PushConnectorOpenTest
    extends MIDlet
    implements CommandListener
{

    public static String serviceName = "PushService";
    public static String serviceUUID = "2000000000001000800000606057028C19";
    ...
    public void startApp()
    {
        connectionUrl = "btspp://localhost:" + serviceUUID + ";name=" + serviceName;
        ...
        try
        {
            notifier = ( StreamConnectionNotifier )Connector.open( connectionUrl );
            form.append( "\nNo Exception: Test FAILED" );
            notifier.close();
        }
        catch( IOException e )
        {
            form.append( "\nIOException: Test PASSED" );
        }
    }
    ...
```

7.4. Pruebas

Una vez desarrollada, se decidió hacer algunas pruebas de la aplicación. Debido a que la implementación a la que está destinada no está terminada ha sido necesario ejecutarla en otra plataforma. En particular se ha utilizado el modelo 6681 de la Serie 60. La característica *PushRegistry* es algo propio del servidor por lo la elección del cliente es algo más libre. El que se ha utilizado para las pruebas ha sido el modelo 6021 de la Serie 40.

Figura 7.9 Nokia 6881 y Nokia 6021

Se podría pensar que la ejecución en esta plataforma de prueba debiera llevar a los mismos resultados que en la Serie 40, para la que se está desarrollando la implmentación. En general se podría responder que sí, pero el diseño de los tests se hace ajustándose a la arquitectura de la implementación a la que están destinados, por lo que puede haber algunas diferencias.

En nuestro caso particular las ha habido, porque el modelo 6881 no soporta el *buffering*. Hasta ahora se ha hablado de que el cliente puede establecer una conexión con el servidor e inmediatamente después enviar datos. Esto es así porque el servidor va almacenando los datos que recibe aunque no haya terminado de iniciar el MIDlet. Así viene descrito en las especificaciones de la implementación que se está realizando para la Serie 40, pero no lo es para el modelo 6881 donde he realizado estas pruebas. Por este motivo se ha tenido que prescindir de las pruebas de los tests destinados al *buffering*.

En cuanto al resto de los tests las pruebas se han realizado con éxito, excepto en dos casos particulares. Por un lado en los tests que llaman a `acceptAndOpen()` una vez que el MIDlet ha sido iniciado por una conexión entrante. Para que la conexión sea realmente aceptada ha sido necesario sustituir la cadena de conexión del servidor de la forma `btspp://localhost:`{UUID} por una del tipo `btspp://:`{UUID}.

Además en el test “Clientes permitidos” tampoco se han obtenido los resultados esperados. Según se comentó en la explicación de este test, se especifica un filtro en el registro de la conexión de forma que se imposibilita la conexión de determinados clientes. Este filtro sólo debe aplicarse a la hora iniciar la aplicación mediante la conexión, pero no cuando el servidor ya está iniciado y espera a conexiones mediante `acceptAndOpen()`. En las pruebas realizadas sin embargo, el filtro se aplicaba en los dos casos, siendo imposible la conexión por parte del cliente, aún cuando el MIDlet ya está iniciado.

En el apéndice D se pueden ver las fotografías de las pruebas realizadas.