A) Glosario de acrónimos

Término	*Definición*
ACL	Asynchronous Connectionless
AMS	Application Management System
BCC	Bluetooth Control Center
BNEP	Bluetooth Network Encapsulation Protocol
CDC	Connected Device Configuration
CLDC	Connected, Limited Device Configuration
CRC	Cyclic Redundancy Check
DUN	Dial Up Networking
EDR	Enhanced Data Rate
FHS	Frequency Hop Synchronisation
FHSS	Frequency Hopping Spread Spectrum
FP	Foundation Profile
FTP	File Transfer Profile
GAP	Generic Access Profile
GFC	Generic Connection Framework
GFSK	Gaussian Frequency Shift Keying
GOEP	Generic Object Exchange Profile
HCI	Host Controller Interface
IEEE	Institute of Electronic and Electrical Engineers
IrDA	Infrared Data Association
ISM	Industrial, Scientific and Medical band
J2ME	Java 2, Micro Edition
JABWT	Java API for Bluetooth Wireless Technologie
JCP	Java Community Process
JSR	Java Specification Request
JVM	Java Virtual Machine
KVM	Kilobyte Virtual Machine
L2CAP	Logical Link Control And Adaptation Protocol
LAN	Local Area Network
LM	Link Manager
LMP	Link Manager Protocol
MAC	Media Access Control
MIDP	Mobile Information Device Profile
MTU	Maximum Transmission Unit
OBEX	Object Exchange Protocol
OSI	Open Systems Interconnect
OTA	Over The Air

Término	*Definición*
PAN	Personal Area Network
PBP	Personal Basis Profile
PCMCIA	Peripheral Component Microchannel Interconnect Architecture
PP	Personal Profile
PSM	Protocol and Service Multiplexor
RF	Radiofrequency
RFCOMM	Bluetooth Radio Frequency Communications Protocol
SCO	Synchronous Connection Oriented
SDAP	Service Discovery Application Profile
SDDB	Service Discovery Database
SDP	Service Discovery Protocol
SIG	Bluetooth Special Interest Group
SPP	Serial Port Profile
TDM	Time Division Multiplexing
UART	Universal Asynchronus Receiver-Transmitter
USB	Universal Serial Bus
VoIP	Voice over Internet Protocol
WLAN	Wireless Local Area Network

B) Bibliografía

Especificaciones

[E1] Bluetooth SIG. Specification of the Bluetooth System, Core, v1.1. www.bluetooth.org. 2000.

[E2] Bluetooth SIG. Specification of the Bluetooth System, v1.2. www.bluetooth.org. 2003.

[E3] Bluetooth SIG. Specification of the Bluetooth System, v2.0. www.bluetooth.org. 2004.

[E4] Java Community Process. J2ME Connected, Limited Device Configuration (JSR-30). www.jcp.org/jsr/detail/30.jsp. 2000.

[E5] Java Community Process. J2ME Connected Device Configuration (JSR-36). www.jcp.org/jsr/detail/36.jsp. 2001.

[E6] Java Community Process. Mobile Information Device Profile (JSR-37). www.jcp.org/jsr/detail/37.jsp. 2000.

[E7] Java Community Process. Java APIs for Bluetooth Wireless Technologie (JSR-82), www.jcp.org/jsr/detail/82.jsp. 2002.

Publicaciones

[L1] Jennifer Bray, Charles F. Sturman. "Bluetooth: Connect Without Cables". Prentice Hall. 2001.

[L2] C Bala Kumar, Paul J. Kline, Timothy J. Thomson. "Bluetooth Application Programming With The Java APIs". Morgan Kaufmann Publishers. 2004.

[L3] Bruce Hopkins, Ranjith Antony. "Bluetooth for Java". Apress. 2003.

[L4] S. Galvez Rojas, L. Ortega Díaz. "J2ME (Java 2 Micro Edition)". Universidad de Málaga. 2003.

[L5] Helmut Balzert. "Lehrbuch der Software-Technik I, Software-Entwicklung". Spektrum Akademischer Verlag. 2001.

[L6] Bart Broekman, Edwin Notenboom. "Testing Embedded Software". Addison Wesley. 2003.

[L7] Elfriede Dustin. "Effective software testing: 50 specific ways to improve your testing". Addison Wesley. 2003.

Internet

[I1] www.bluetooth.org

[I2] www.palowireless.com

[I3] www.zonabluetooth.com

[I4] www.techworld.com

[I5] www.todosymbian.com

[I6] www.java.sun.com

[I7] www.forum.nokia.com

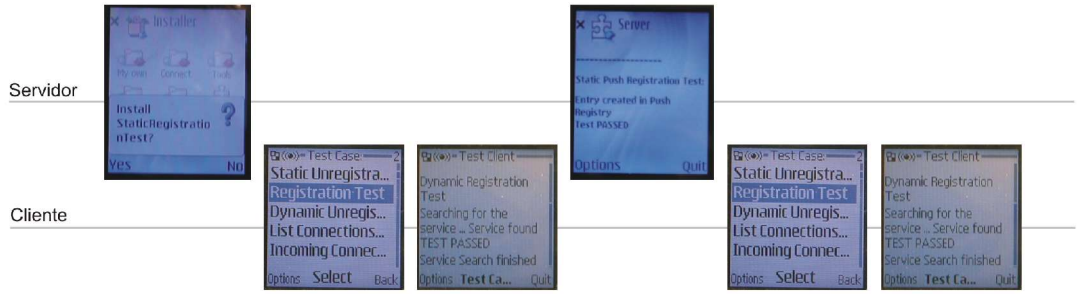
C) Indices de figuras y tablas

Indice de figuras

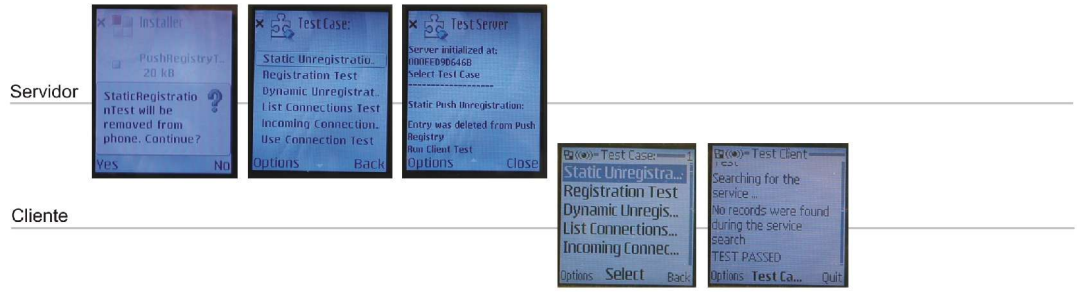
Indice de tablas

D) Fotografías de las pruebas

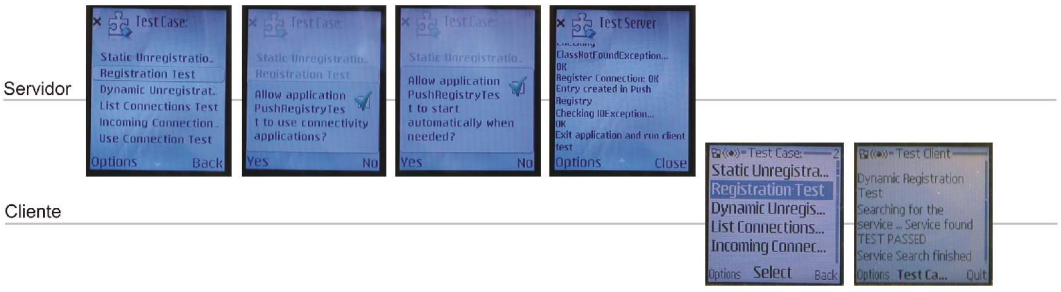
Registro estático



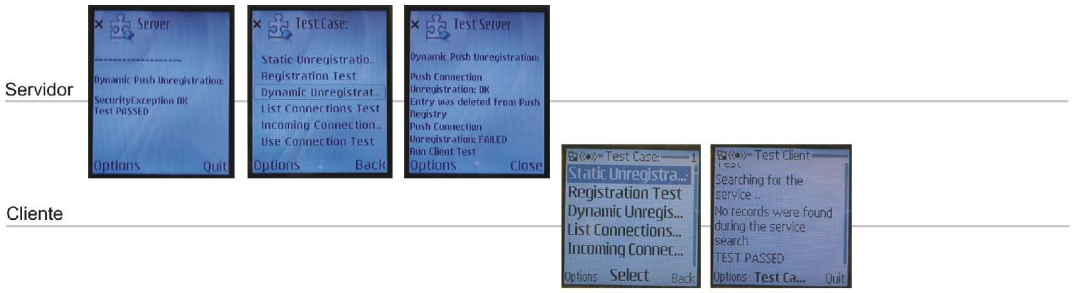
Eliminación estática de registro



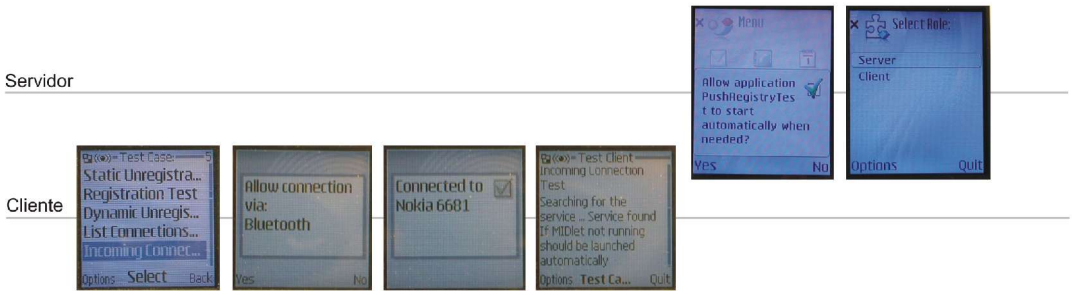
Registro dinámico



Eliminación dinámica de registro



MIDlet iniciado automáticamente por conexión entrante

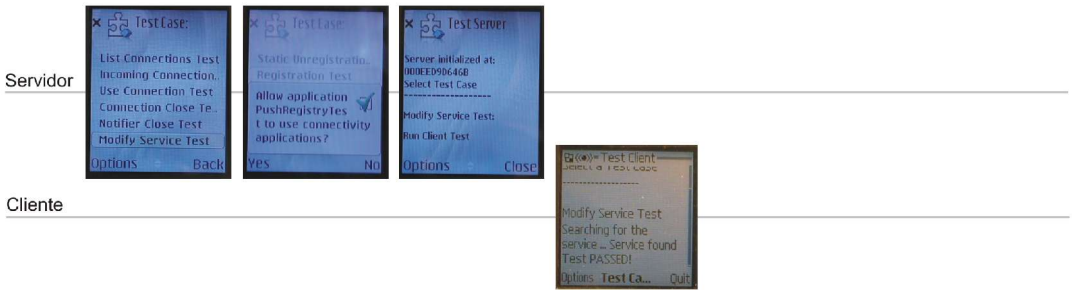


Uso de la conexión por MIDlet iniciado manualmente

Uso de la conexión por MIDlet iniciado automáticamente

Listar conexiones

Actualizar Service Record



Excepción en Connector.open

Cerrar conexión

Notifier.close

Clientes permitidos

E) Código de la aplicación

Clase PushRegistryTest (MIDlet PushRegistryTest):

```
/*
 * PushRegistryTest.java
 *
 * Created on 12. March 2005, 23:06
 */

//============================================================================
// Import Statements

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;


//============================================================================
// CLASS DECLARATIONS

/** Main class. Subclass of MIDlet class. Shows a list where the user can choose
 * server or client application
 *
 * @author Carlos Guerrero
 */

public class PushRegistryTest
    extends MIDlet
    implements CommandListener
{


//============================================================================
// Instance variables

    private Display display;


//============================================================================
// Constructors and miscellaneous (finalize method, initialization block,...)

    /** Constructs a PushRegistryTest object
     */
    public PushRegistryTest()
    {
        display = Display.getDisplay( this );
    }


//============================================================================
// Public Methods

    /** startApp() method of the MIDlet object
     */
    public void startApp()
    {
        SelectionList select = new SelectionList( this );
        display.setCurrent( select);
    }


    /** pauseApp() method of the MIDlet object
     */
    public void pauseApp()
    {
    }
```

```
    /** destroyApp() method of the MIDlet object
     * @param unconditional  */
    public void destroyApp( boolean unconditional )
    {
    }


    /** commandAction() method of the CommandListener interface implemented by
     * PushRegistryTest
     */
    public void commandAction( Command command, Displayable displayable )
    {
        if( command.getCommandType() == Command.EXIT )
        {
            // User chose to quit the application
            exitMIDlet();
        }
    }


    /** Calls destroyApp() and notifyDestroyed() when MIDlet exits
     */
    public void exitMIDlet()
    {
        destroyApp( false );
        notifyDestroyed();
    }


    /** Returns the MIDlet object of this application
     * @return the MIDlet object
     */
    public MIDlet getMIDlet()
    {
        return this;
    }


//=============================================================================
// Inner Classes

    /** List where the user can select Server/Client
     */
    public class SelectionList
    extends List
    implements CommandListener
    {

        private PushRegistryTest prObject;

        /** Constructs a SelectionList
         * @param p PushRegistry object unique to this MIDlet.
         */
        public SelectionList( PushRegistryTest p )
        {
            super( "Select Role:", List.IMPLICIT );
            append( "Server", null );
            append( "Client", null );
            addCommand( new Command( "Select", Command.OK, 1 ) );
            addCommand( new Command( "Quit", Command.EXIT, 1 ) );
            setCommandListener( this );

            prObject = p;
        }


        /** commandAction() method of the CommandListener interface implemented by
         * SelectionList
```

```
         */
        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();
                }

            }
            else if( c.getCommandType() == Command.EXIT )
            {
                exitMIDlet();
            }
        }
    }
}
```

Clase PushRegistryTester (MIDlet PushRegistryTest)

```
/*
 * PushRegistryTester.java
 *
 * Created on March 15, 2005, 3:22 PM
 */


//================================================================================
// Import Statements

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;


//================================================================================
// CLASS DECLARATIONS

/** Class to perform Push Registry Test. Both server and client have to be a
 * subclass of this class.
 *
 * @author Carlos Guerrero
 */

public class PushRegistryTester
    implements Runnable, CommandListener
{

//================================================================================
// Final variables (Class constants)

    /** 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;

    /** ID of the Static Push Unregistration Test. The value is 2
     */
    protected static final int STATIC_UNREGISTRATION_TEST = 2;

    /** ID of the Dynamic Push Registration Test. The value is 3
     */
    protected static final int DYNAMIC_REGISTRATION_TEST = 3;

    /** ID of the Dynamic Push Unregistration Test. The value is 4
     */
    protected static final int DYNAMIC_UNREGISTRATION_TEST = 4;

    /** ID of the incoming Connection Test. The value is 5
     */
    protected static final int INCOMING_CONNECTION_TEST = 5;

    /** ID of the Data BT Init Test. The value is 6
     */
    protected static final int BT_INIT_TEST = 6;

    /** ID of the List Connections Test. The value is 7
     */
    protected static final int LIST_CONNECTIONS_TEST = 7;

    /** ID of the Modify Service Test. The value is 8
     */
    protected static final int MODIFY_SERVICE_TEST = 8;

    /** ID of the Use Connection Test. The value is 9
     */
    protected static final int USE_CONNECTION_TEST = 9;

    /** ID of the Connection Close Test. The value is 10
     */
    protected static final int CONNECTION_CLOSE_TEST = 10;

    /** ID of the Data Buffer Test. The value is 11
     */
    protected static final int SIMPLE_BUFFER_TEST = 11;

    /** ID of the Data Notifier Close Test. The value is 12
     */
    protected static final int NOTIFIER_CLOSE_TEST = 12;

    /** ID of the Robustness Data Buffer Test. The value is 13
     */
    protected static final int ROBUSTNESS_BUFFER_TEST = 13;

    /** ID of the Accumulation Data Buffer Test. The value is 14
     */
    protected static final int ACCUMULATION_BUFFER_TEST = 14;

    /** ID of the discard Data Buffer Test. The value is 15
     */
    protected static final int DISCARD_BUFFER_TEST = 15;

    /** ID of the Discard Close Data Buffer Test. The value is 16
     */
    protected static final int DISC_CLOSE_BUFFER_TEST = 16;

    /** ID of the Not Allowed Test. The value is 17
     */
    protected static final int NOT_ALLOWED_TEST = 17;
```

```
//============================================================================
// Class (static) variables

    /** 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";

    /** Push filter to be used in the Push Registration. By default is *
     */
    public static String pushFilter = "*";

    /** Service Name to be registered in the Push Registry. By default is
     * PushService
     */
    public static String serviceName = "PushService";

    /** ID of a service attribute used for testing the Service Record
     */
    protected static int serviceAttributeId = 0x0200;

    /** Value of a service attribute used for testing the Service Record
     */
    protected static String serviceAttributeValue = "Test Attribute";

    /** Data to be sended
     */
    protected static String testData = "data";


//============================================================================
// Instance variables

    /** Variable used to know if the test is running
     */
    protected boolean testIsRunning;
    /** Some tests pause to allow the user perform some actions, so the test has
     * several phases. testPhase indicates which phase is the next to be performed.
     */
    protected int testPhase;

    /** PushRegistry object unique to this MIDlet.
     */
    protected PushRegistryTest prObject;
    /** Form to be used by both applications (server and client).
     */
    protected Form testerForm;
    private Display display;
    private int testCase;


//============================================================================
// Constructor

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

```
//============================================================================
// Public Methods

    /** run() method of the Thread.
     */
    public void run()
    {
    }


    /** Starts the test case to be performed.
     * @param tC Test case ID. NO_TEST_RUNNING, STATIC_REGISTRATION_TEST,
     * STATIC_UNREGISTRATION_TEST, DYNAMIC_REGISTRATION_TEST,
     * DYNAMIC_UNREGISTRATION_TEST, INCOMING_CONNECTION_TEST,
     * LIST_CONNECTIONS_TEST, MODIFY_SERVICE_TEST, USE_CONNECTION_TEST,
     * CONNECTION_CLOSE_TEST, SIMPLE_BUFFER_TEST, BT_INIT_TEST,
     * NOTIFIER_CLOSE_TEST, ROBUSTNESS_BUFFER_TEST, ACCUMULATION_BUFFER_TEST,
     * DISCARD_BUFFER_TEST, DISC_CLOSE_BUFFER_TEST
     */
    public void runTest(int tC)
    {
    }


    /** 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;
    }


    /** The same as CommandListener.commandAction(javax.microedition.lcdui.Command
     * command, javax.microedition.lcdui.Displayable displayable)
     * @param command a Command object identifying the command.
     * @param displayable the Displayable on which this event has occurred.
     */
    public void commandAction( Command command, Displayable displayable )
    {
    }


//============================================================================
// Protected Methods


    /** Gets the Display object that is unique to this MIDlet.
     * @return the display object that application can use for its user interface.
     */
    protected Display getDisplay()
```

```
    {
        return display;
    }


//==========================================================================
// Inner Classes

    /** TestSelector is a List which displays the test cases that can be selected
     * to be performed.
     */
    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 );
            append( "Registration Test", null );
            append( "Dynamic Unregistration Test", null );
            append( "List Connections Test", null );
            append( "Incoming Connection Test", null );
            append( "Use Connection Test", null );
            append( "Connection Close Test", null );
            append( "Notifier Close Test", null );
            append( "Modify Service Test", null );
            append( "Simple Buffer Test", null );
            append( "Robustness Buffer Test", null );
            append( "Accumulation Buffer Test", null );
            append( "Discard Buffer Test", null );
            append( "Disc Close Buffer Test", null );
            append( "Not Allowed Test", null );
            addCommand( new Command( "Select", Command.OK, 1 ) );
            addCommand( new Command( "Back", Command.BACK, 1 ) );
        }


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


        /** The same as CommandListener.commandAction(javax.microedition.lcdui.
         * Command command, javax.microedition.lcdui.Displayable displayable)
         * @param command a Command object identifying the command.
         * @param displayable the Displayable on which this event has occurred.
         */
        public void commandAction(Command command, Displayable displayable)
        {
            if( command.getCommandType() == Command.BACK )
            {
                display.setCurrent(testerForm);
            }
            else
            {
                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 );
                    }
                    if( s.equals( "Registration Test" ) )
                    {
                        runTest( DYNAMIC_REGISTRATION_TEST );
                    }
                    if( s.equals( "Dynamic Unregistration Test" ) )
                    {
                        runTest( DYNAMIC_UNREGISTRATION_TEST );
                    }
                    if( s.equals( "Incoming Connection Test" ) )
                    {
                        runTest( INCOMING_CONNECTION_TEST );
                    }
                    if( s.equals( "Use Connection Test" ) )
                    {
                        runTest( USE_CONNECTION_TEST );
                    }
                    if( s.equals( "Connection Close Test" ) )
                    {
                        runTest( CONNECTION_CLOSE_TEST );
                    }
                    if( s.equals( "List Connections Test" ) )
                    {
                        runTest( LIST_CONNECTIONS_TEST );
                    }
                    if( s.equals( "Modify Service Test" ) )
                    {
                        runTest( MODIFY_SERVICE_TEST );
                    }
                    if( s.equals( "Simple Buffer Test" ) )
                    {
                        runTest( SIMPLE_BUFFER_TEST );
                    }
                    if( s.equals( "Robustness Buffer Test" ) )
                    {
                        runTest( ROBUSTNESS_BUFFER_TEST );
                    }
                    if( s.equals("Accumulation Buffer Test" ) )
                    {
                        runTest( ACCUMULATION_BUFFER_TEST );
                    }
                    if( s.equals( "Discard Buffer Test" ) )
                    {
                        runTest( DISCARD_BUFFER_TEST );
                    }
                    if( s.equals( "Disc Close Buffer Test" ) )
                    {
                        runTest( DISC_CLOSE_BUFFER_TEST );
                    }
                    if( s.equals( "Notifier Close Test" ) )
                    {
                        runTest( NOTIFIER_CLOSE_TEST );
                    }
                    if( s.equals( "Not Allowed Test" ) )
                    {
                        runTest( NOT_ALLOWED_TEST );
                    }
                }
            }
        }
    }
}
```

Clase PushRegistryTestServer (MIDlet PushRegistryTest)

```
/*
 * PushRegTestServer.java
 *
 * Created on February 16, 2005, 11:45 AM
 */


//================================================================================
// Import Statements

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;
import javax.bluetooth.*;
import javax.microedition.rms.*;


//================================================================================
// CLASS DECLARATIONS

/** The class implements the server to perform the Push Registry Test
 *
 * @author Carlos Guerrero
 *
 */

public class PushRegistryTestServer
    extends PushRegistryTester
{

//================================================================================
// Instance variables:
//

    // displayables
    private Form testerForm;
    private TestSelector testSelector;

    // Alert which shows the user some information
    private MessageInformation info;

    // boolean which indicates whether the test have been passed or not
    private boolean testPassed;

    // URL to use for the connection
    private String connectionUrl;

    // Bluetooth variables
    private LocalDevice local;
    private StreamConnectionNotifier notifier;
    private StreamConnection connection;
    private ServiceRecord record;


//================================================================================
// Constructor

    /** Creates a PushRegistryTestServer.
     * @param p PushRegistryTest object unique to this MIDlet.
     */
    public PushRegistryTestServer( PushRegistryTest p)
    {
        super( p);
    }
```

```
//============================================================================
// Public Methods

    /** run() method of the Thread. This method is started when the thread starts.
    * Therefore the variables and objects are initialized in this method.
    */
    public void run()
    {
        // initialization of variables
        connectionUrl = "btspp://localhost:20000000000010008000006057028C19;name="
                        + serviceName;

        testPhase = 0;

        // 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();

        // add Commands to the Form
        testerForm.addCommand( new Command( "Test Case", Command.ITEM, 1 ) );
        testerForm.addCommand( new Command( "Clear Log", Command.ITEM, 1 ) );
        testerForm.append( "\nSelect Test Case" );

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

        // initialization of test state variables
        testPassed = true;
    }


    /** Same as CommandListener.commandAction(javax.microedition.lcdui.Command command,
     * javax.microedition.lcdui.Displayable displayable)
     * @param c a Command object identifying the command.
     * @param d the Displayable on which this event has occurred.
     */
    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();
            }
        }
    }


    /** Starts the test case to be performed. Depending on the test case passed
     * as parameter, runTest() starts the method which performs the specified test.
     * @param tC Constant which indicates the test case to be performed.
     * NO_TEST_RUNNING, STATIC_UNREGISTRATION_TEST, STATIC_REGISTRATION_TEST,
     * DYNAMIC_UNREGISTRATION_TEST, DYNAMIC_REGISTRATION_TEST.
     */
```

```
    public void runTest( int tC )
    {
        getDisplay().setCurrent( testerForm );

        setTestCase( tC );
        testPhase = 0;

        switch( tC )
        {
            case STATIC_UNREGISTRATION_TEST:
                staticPushUnregistrationTest();
                break;
            case DYNAMIC_REGISTRATION_TEST:
                dynamicPushRegistrationTest();
                break;
            case DYNAMIC_UNREGISTRATION_TEST:
                dynamicPushUnregistrationTest();
                break;
            case INCOMING_CONNECTION_TEST:
                incomingConnectionTest();
                break;
            case USE_CONNECTION_TEST:
                useConnectionTest();
                break;
            case SIMPLE_BUFFER_TEST:
                simpleBufferTest();
                break;
            case ROBUSTNESS_BUFFER_TEST:
                robustnessBufferTest();
                break;
            case ACCUMULATION_BUFFER_TEST:
                accumulationBufferTest();
                break;
            case DISCARD_BUFFER_TEST:
                discardBufferTest();
                break;
            case DISC_CLOSE_BUFFER_TEST:
                discCloseBufferTest();
                break;
            case LIST_CONNECTIONS_TEST:
                listConnectionsTest();
                break;
            case MODIFY_SERVICE_TEST:
                modifyServiceTest();
                break;
            case CONNECTION_CLOSE_TEST:
                connectionCloseTest();
                break;
            case NOTIFIER_CLOSE_TEST:
                notifierCloseTest();
                break;
            case NOT_ALLOWED_TEST:
                notAllowedTest();
                break;
        }
    }


    /** Performs the Static Push Unregistration Test.
     */
    public void staticPushUnregistrationTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append("\n-------------------");
        testerForm.append("\n\nStatic Push Unregistration:\n");

        // 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();
    }


    /** Performs the Dynamic Push Registration Test:
     * 1. Try to register the connection with a wrong name to check whether
     *      ClassNotFoundException is thrown.
     * 2. Register the connection.
     * 3. Check whether the connection exists in PushRegistry
     * 4. Try to register the connection again and an IOException should be thrown.
     */
    public void dynamicPushRegistrationTest()
    {
        testPassed = true;

        testerForm.append("\n--------------------");
        testerForm.append("\n\nDynamic Push Registration:\n");

        // 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");
        }

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

```
        } else {
            testerForm.append("\nTest FAILED");
        }

        // test finished
        testFinished();
    }


    /** Performs the Dynamic Push Unregistration Test.
     */
    public void dynamicPushUnregistrationTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append("\n-------------------");
        testerForm.append("\n\nDynamic Push Unregistration:\n");

        // 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;
        }

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

        // test finished
        testFinished();
    }


    /** Performs the Incoming Connection Test. Does nothing because this test is
     * suposed to be performed only on the client side.
     */
    public void incomingConnectionTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nIncoming connection test:\n" );
    }


    /** Performs the Use Connection Test. Calls connector.open() and acceptAndOpen()
     * if the user accept.
     */
    public void useConnectionTest()
    {
        switch( testPhase )
        {
```

```
            case 0:
                // initialization of variables
                testPassed = true;

                testerForm.append( "\n--------------------" );
                testerForm.append( "\n\nUse connection test:\n" );

                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();

                getDisplay().setCurrent( testerForm );
                // test finished
                testFinished();
        }
    }


    /** Performs the Connection Close Test. Calls connector.open() and acceptAndOpen().
     *
     */
    public void connectionCloseTest()
    {
        switch( testPhase )
        {
            case 0:
                // initialization of variables
                testPassed = true;

                testerForm.append( "\n--------------------" );
                testerForm.append( "\n\nConnection close test:\n" );
                getDisplay().setCurrent( testerForm );

                testPhase ++;
                startBluetoothServer();

                break;
            case 1:
                testPhase ++;

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


    /** Performs the Simple Buffering Test. Calls connector.open() and acceptAndOpen().
     */
    public void simpleBufferTest()
```

```
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nSimple buffer test:\n" );

        startBluetoothServer();
    }


    /** Performs the Robustness Buffering Test. Calls connector.open() and acceptAndOpen().
     */
    public void robustnessBufferTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nRobustness buffer test:\n" );

        startBluetoothServer();
    }


    /** Performs the Accumulation Buffering Test. Calls connector.open() and acceptAndOpen().
     */
    public void accumulationBufferTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nAccumulation buffer test:\n" );

        // start a acceptAndOpen Thread
        startBluetoothServer();

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


    /** Performs the Discard Buffering Test. Calls connector.open() and acceptAndOpen().
     */
    public void discardBufferTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nDiscard buffer test:\n" );

        startBluetoothServer();
    }


    /** Performs the Discard Close Buffering Test. Test calls connector.open()
     * and acceptAndOpen().
     */
    public void discCloseBufferTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n-------------------" );
        testerForm.append( "\n\nDisc Close buffer test:\n" );
```

```
        startBluetoothServer();
    }


    /** Performs the List Connections Test.
     */
    public void listConnectionsTest()
    {
        switch(testPhase)
        {
            case 0:
                // initialization of variables
                testPassed = true;

                testerForm.append("\n-------------------");
                testerForm.append("\n\nList Connections:\n");

                // 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();
        }
    }


    /** Performs the List Connections Test. Creates a new attribute for the Service
     * Record and update it.
     */
    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();
    }


    /** Performs the Notifier Close Test. Test calls connector.open()
     * and acceptAndOpen().
     */
    public void notifierCloseTest()
    {
        // initialization of variables
        testPassed = true;

        testerForm.append( "\n--------------------" );
        testerForm.append( "\n\nNotifier Close test:\n" );

        startBluetoothServer();
    }


    /** Performs the Not Allowed Test
     */
    public void notAllowedTest()
    {
        testPassed = true;
        String filter = "0a00112e62a7";  // aleatory BT address

        testerForm.append("\n--------------------");
        testerForm.append("\n\nNot Allowed Test:\n");

        // register Connection
        testerForm.append("\nRegistering connection...");

        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();
    }


    /** Initializes the server bluetooth system by retrieving the LocalDevice object.
     * @return True if the initialization was sucesfully or false in other case.
     */
    public boolean bluetoothServerInit()
    {
        boolean bluetoothResult = true;

        try
        {
            local = LocalDevice.getLocalDevice();
        }
        catch ( BluetoothStateException e )
        {
            bluetoothResult = false;
            testerForm.append( "\nBluetooth system could not be initialized" );
        }

        testerForm.append( "\nServer initialized at: " + local.getBluetoothAddress() );

        return bluetoothResult;
    }


    /** Starts the server by calling Connector.open and acceptAndOpen().
     */
    public void startBluetoothServer()
    {

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

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

    }


    /** Check whether the MIDlet exists in Push Registry.
     * @return True if the MIDlet entry exists in the Push Registry and false if not.
     */
    public boolean existsInRegistry()
    {
        String[] connections = null;
        boolean existsInRegistry = false;

        connections = PushRegistry.listConnections(false);
```

```
        if(connections.length > 0)
        {
            for(int i = 0; i < connections.length; i++)
            {
                if(connections[i].equals(connectionUrl))
                {
                    existsInRegistry = true;
                }
                else
                {
                    testerForm.append("\nAnother connection found: " + connections[i]);
                }
            }
        }

        return existsInRegistry;
    }


    /** Performs a dynamic Unregister Connection scenario by calling
     * PushRegistry.unregisterConnection().
     * @return True if the unregistration was sucesfully done or false in other case.
     */
    public boolean unregisterTestConnection()
    {
        boolean unregisterResult = false;

        try
        {
            if(unregisterResult = PushRegistry.unregisterConnection(connectionUrl))
            {
                testerForm.append("\nPush Connection Unregistration: OK");
                unregisterResult = true;
            }
            else
            {
                testerForm.append("\nPush Connection Unregistration: FAILED");
            }
        }
        catch (SecurityException e)
        {
            testerForm.append("\nPush Connection already registered by another MIDlet");
            unregisterResult = false;
        }

        return unregisterResult;
    }


    /** Performs a dynamic Unregister Connection scenario by calling
     * PushRegistry.registerConnection().
     * @return True if the registration was sucesfully done or false in other case.
     * @param security if true the connection require security
     */
    public boolean registerTestConnection(boolean security)
    {
        boolean registerResult = false;
        String connectionUrlSecurity = connectionUrl +
                                       ";authenticate=true;authorize=true";

        try
        {
            if(security)
            {
                PushRegistry.registerConnection(connectionUrlSecurity,
                                                midletClassName, pushFilter);
            }
            else
            {
```

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

        if(registerResult == false)
        {
            testerForm.append("\nRegister Connection: FAILED");
        }
        else
        {
            testerForm.append("\nRegister Connection: OK");
        }

        return registerResult;
    }


 //============================================================================
 // Protected Methods

    /** Shows an Alert window.
     * @param infoString the String to be shown on the Alarm window
     */
    protected void showInformation( String infoString )
    {
        info.setString( infoString );

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


 //============================================================================
 // Inner Classes

    /** This class is an Alert which is shown to the user when he has to perform some
     * action. After it the user is suposed to press the OK button and the test can
     * continue.
     */
    protected class MessageInformation
        extends Alert
        implements CommandListener
    {

        /** Contructs a MessageInformation object
         * @param msg Message to be shown.
         */
        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 );
        }
```

```
        /** commandAction of the CommandListener interface which is implemented by the
         * MessageInformation class.
         * @param c a Command object identifying the command.
         * @param d the Displayable on which this event has occurred.
         */
        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;
                    case USE_CONNECTION_TEST:
                        useConnectionTest();
                        break;
                }
            }
        }
    }


    /** This Thread class call acceptAndOpen method.
     */
    protected class AcceptAndOpenThread
        extends Thread
    {
        /**
         * Constructor
         **/
        public AcceptAndOpenThread()
        {
        }

        /**
         * run method
         * Start AcceeptAndOpen and wait on Exception or connection. When the
         * connection is established several actions should be performed
         * depending on the test case.
         */
        public void run()
        {
            InputStream input;
            byte[] data = new byte[10];
            int length = 0;
            String inputStr = "";

            try
            {
                connection = ( StreamConnection )notifier.acceptAndOpen();

                testerForm.append( "Connected!" );

                switch( getTestCase() )
                {
                    case USE_CONNECTION_TEST:
                        testerForm.append( "\nTest PASSED" );
                        // test finished
                        testFinished();

                        connection.close();
                        notifier.close();
                        break;
```

```
                    case CONNECTION_CLOSE_TEST:
                        connection.close();

                        connectionCloseTest();

                        break;
                    case SIMPLE_BUFFER_TEST:
                        input = connection.openInputStream();

                        while( ( length = input.read( data ) ) != -1 )
                        {
                            inputStr = new String( data, 0, length );
                        }

                        testerForm.append( "\nReceived: " + inputStr );

                        if( inputStr.equals( testData ) )
                        {
                            testerForm.append( "\nData received: Test PASSED" );
                        }
                        else
                        {
                            testerForm.append( "\nData does not match: Test FAILED" );
                        }

                        // test finished
                        testFinished();
                        break;

                    case ROBUSTNESS_BUFFER_TEST:
                        input = connection.openInputStream();

                        while( ( length = input.read( data ) ) != -1 )
                        {
                            inputStr = new String( data, 0, length );
                        }

                        testerForm.append( "\nReceived " + length + " bytes");
                        testerForm.append( "\nTest PASSED");

                        // test finished
                        testFinished();
                        break;

                    case ACCUMULATION_BUFFER_TEST:
                        input = connection.openInputStream();

                        while( ( length = input.read( data ) ) != -1 )
                        {
                            inputStr = new String( data, 0, length );
                        }

                        testerForm.append( "\n" + inputStr );

                        testerForm.append( "\nIf numbers are 123 456: Test PASSED" );

                        // test finished
                        testFinished();
                        break;

                    case DISCARD_BUFFER_TEST:
                        input = connection.openInputStream();

                        if( input.read( data ) == -1 )
                        {
                            testerForm.append( "\nNo buffered data: Test PASSED" );
                        }
                        else
                        {
                            testerForm.append( "\nBuffered data: Test FAILED" );
```

```
                        }

                        // test finished
                        testFinished();
                        break;

                    case DISC_CLOSE_BUFFER_TEST:
                        input = connection.openInputStream();

                        notifier.close();

                        if( input.read( data ) == -1 )
                        {
                            testerForm.append( "\nNo buffered data: Test PASSED" );
                        }
                        else
                        {
                            testerForm.append( "\nBuffered data: Test FAILED" );
                        }

                        // test finished
                        testFinished();
                        break;

                    case NOTIFIER_CLOSE_TEST:
                        testerForm.append( "\nClosing Notifier object" );
                        notifier.close();
                        break;
                }
            }
            catch( IOException e )
            {
                testerForm.append( "\nIOException while starting server" );
            }
        }
    }
}
```

Clase PushRegistryTestClient (MIDlet PushRegistryTest)

```
/*
 * PushRegTestClient.java
 *
 * Created on February 16, 2005, 3:47 PM
 */


//================================================================================
// Import Statements

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import java.util.*;
import javax.microedition.io.*;
import java.io.*;


//================================================================================
// CLASS DECLARATIONS

/** This class implements the client to perform the Push Registry Tests.
 *
 * @author Carlos Guerrero
 */
```

```
public class PushRegistryTestClient
    extends PushRegistryTester
    implements DiscoveryListener {


//================================================================================
// Final variables (Class constants)

    /** Specifies that the test on the client will try to find the service on the server.
     */
    static protected final int SEARCH_SERVICE = 0;
    /** Specifies that the test on the client will try to find the service on the server,
     * and if it is found, connect to it.
     */
    static protected final int SEARCH_AND_CONNECT = 1;
    /** Specifies that the test on the client will try to find the service on the server,
     * and if it is found, connect to it and send data. The data to send depend on the
     * test case.
     */
    static protected final int SEARCH_AND_SEND = 2;


//================================================================================
// Instance variables

    //displayables and Display object
    private List deviceList;
    private Form testerForm;
    private TestSelector testSelector;
    private Display display;

    // Bluetooth variables
    private LocalDevice local;
    private RemoteDevice serverDevice;
    private DiscoveryAgent agent;
    private StreamConnection connection;
    private ServiceRecord remoteRecord;

    private String connectionUrl;
    private Vector deviceVector;
    private boolean isInInquiry;
    private boolean serviceFound;
    private int transId;
    private int testPurpose;


//================================================================================
// Constructor

    /** Creates a PushRegistryTestClient.
     * @param p PushRegistryTest object unique to this MIDlet.
     */
    public PushRegistryTestClient( PushRegistryTest p)
    {
        super( p);
    }


//================================================================================
// Public Methods


    /** run() method of the Thread. This method is started when the thread starts.
     * Initializes the BT client and calls startTestServer() which shows a list of
     * devices to select the server.
     */
    public void run()
    {
        display = getDisplay();
```

```
        if( bluetoothClientInit() )
        {
            selectTestServer();
        }
    }


    /** Starts the test case to be performed. Specifies what has to do the client
     * depending on the test case.
     * @param tC Constant which indicates the test case to be performed. NO_TEST_RUNNING,
STATIC_UNREGISTRATION_TEST, STATIC_REGISTRATION_TEST, DYNAMIC_UNREGISTRATION_TEST,
DYNAMIC_REGISTRATION_TEST.
     */
    public void runTest( int tC )
    {
        display.setCurrent( testerForm );

        setTestCase( tC );

        serviceFound = false;

        testerForm.append( "\n-------------------" );

        switch( getTestCase() ) {
            case STATIC_UNREGISTRATION_TEST:
                testerForm.append( "\n\nStatic Unregistration Test\n\n" );
                testPurpose = SEARCH_SERVICE;
                break;
            case DYNAMIC_REGISTRATION_TEST:
                testerForm.append( "\n\nDynamic Registration Test\n\n" );
                testPurpose = SEARCH_SERVICE;
                break;
            case DYNAMIC_UNREGISTRATION_TEST:
                testerForm.append( "\n\nDynamic 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 USE_CONNECTION_TEST:
                testerForm.append( "\n\nUse Connection Test\n\n" );
                testPurpose = SEARCH_AND_CONNECT;
                break;
            case CONNECTION_CLOSE_TEST:
                testerForm.append( "\n\nConnection Close 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;
            case ROBUSTNESS_BUFFER_TEST:
                testerForm.append( "\n\nRobustness Buffer Test\n\n" );
                testPurpose = SEARCH_AND_SEND;
                break;
            case ACCUMULATION_BUFFER_TEST:
                testerForm.append( "\n\nAccumulation Buffer Test\n\n" );
                testPurpose = SEARCH_AND_SEND;
                break;
            case DISCARD_BUFFER_TEST:
                testerForm.append( "\n\nDiscard Buffer Test\n\n" );
                testPurpose = SEARCH_AND_SEND;
                break;
            case DISC_CLOSE_BUFFER_TEST:
                testerForm.append( "\n\nDisc Close Buffer Test\n\n" );
                testPurpose = SEARCH_AND_SEND;
                break;
            case LIST_CONNECTIONS_TEST:
                testerForm.append( "\n\nList Connecions Test\n\n" );
                testPurpose = SEARCH_AND_CONNECT;
```

```
                break;
            case MODIFY_SERVICE_TEST:
                testerForm.append( "\n\nModify Service Test\n\n" );
                testPurpose = SEARCH_SERVICE;
                break;
            case NOTIFIER_CLOSE_TEST:
                testerForm.append( "\n\nNotifier Close Test\n\n" );
                testPurpose = SEARCH_AND_CONNECT;
                break;
            case NOT_ALLOWED_TEST:
                testerForm.append( "\n\nNot Allowed Test\n\n" );
                testPurpose = SEARCH_AND_CONNECT;
                break;
        }

        startServiceSearch();
    }


    /** The same as CommandListener.commandAction(javax.microedition.lcdui.Command command,
     * javax.microedition.lcdui.Displayable displayable)
     * @param command  a Command object identifying the command.
     * @param displayable the Displayable on which this event has occurred.
     */
    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();
            testerForm.append( "\nServer: " + serverDevice.getBluetoothAddress() );
            testerForm.append( "\nSelect a Test Case\n" );

        }
    }


    /** The same as DiscoveryListener.deviceDiscovered(RemoteDevice btDevice,
```

```
     * DeviceClass cod). When a device is discovered, is added to the list of remote
     * devices.
     * @param remoteDevice the device that was found during the inquiry
     * @param deviceClass the service classes, major device class, and minor device class of
the remote device
     */
    public void deviceDiscovered( RemoteDevice remoteDevice, DeviceClass deviceClass )
    {
        addRemoteDeviceToList( remoteDevice );
    }


    /** The same as DiscoveryListener.inquiryCompleted( int discType ). The inquiry could
     * be terminated by the user or ends normally.
     * @param type the type of request that was completed; either INQUIRY_COMPLETED,
INQUIRY_TERMINATED, or INQUIRY_ERROR
     */
    public void inquiryCompleted( int type )
    {
        isInInquiry = false;

        Alert dialog = null;

        if( type != DiscoveryListener.INQUIRY_COMPLETED )
        {
            if( type == DiscoveryListener.INQUIRY_TERMINATED )
            {
                return;
            }
            else
            {
                dialog = new Alert( "Bluetooth Error", "Inquiry failed", null,
AlertType.INFO );
                dialog.setTimeout( Alert.FOREVER );
            }
        }
        else
        {
            dialog = new Alert( "Inquiry completed", "inquiry completed normally", null,
AlertType.INFO );
            dialog.setTimeout( 2000 );
        }

        display.setCurrent( dialog );
    }


    /** The same as DiscoveryListener.serviceSearchCompleted(int transID,
     * int respCode ). Depending on the test case the client should have found the service
     * or not, has to connect to the service, or has to check the retrieved Service Record.
     * @param transID the transaction ID identifying the request which initiated the service
search
     * @param type the response code that indicates the status of the transaction
     */
    public void serviceSearchCompleted( int transID, int type )
    {

        switch( type )
        {
            case SERVICE_SEARCH_COMPLETED:
                if( !serviceFound )
                {
                    testerForm.append( "\nService not found" );

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

```
                            break;
                        case DYNAMIC_REGISTRATION_TEST:
                            testerForm.append( "\nTEST FAILED\n" );
                            break;
                        case DYNAMIC_UNREGISTRATION_TEST:
                            testerForm.append( "\nTEST PASSED\n" );
                            break;
                        default:
                            testerForm.append( "\nService not found: Test incompleted\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" );
                                            }
                                            else
                                            {
                                                testerForm.append( "\nAttribute is not updated:
Test FAILED!\n" );
                                            }
                                        }
                                        else
                                        {
                                            testerForm.append( "\nAttribute is empty: Test
FAILED!\n" );
                                        }
                                    }
                                    else
                                    {
                                        testerForm.append( "\nData Element is null" );
                                    }
                                }
                                else
                                {
                                    testerForm.append( "\nService Record is null" );
                                }
                            }
                            else
                            {
                                testerForm.append( "\nService Search finished" );
                            }
```

```
                            break;

                        case SEARCH_AND_CONNECT:
                            connectToServerTest();
                            testerForm.append( "\nIf MIDlet not running should be launched
automatically" );

                            if( getTestCase() == LIST_CONNECTIONS_TEST )
                            {
                                testerForm.append( "\nClient test finished\n" );
                            }

                            break;
                        case SEARCH_AND_SEND:
                            connectToServerTest();
                            testerForm.append( "\nIf MIDlet not running should be launched
automatically" );
                            break;
                    }
                }
                break;
            case SERVICE_SEARCH_ERROR:
                testerForm.append( "\nAn error occurred while processing the request" );
                if( !serviceFound )
                {
                    testerForm.append( "\nService not found" );
                }
                testFinished();
                break;
            case SERVICE_SEARCH_NO_RECORDS:
                testerForm.append( "\nNo records were found during the service search" );

                switch( getTestCase() )
                {
                    case STATIC_REGISTRATION_TEST:
                        testerForm.append( "\nTEST FAILED\n" );
                        break;
                    case STATIC_UNREGISTRATION_TEST:
                        testerForm.append( "\nTEST PASSED\n" );
                        break;
                    case DYNAMIC_REGISTRATION_TEST:
                        testerForm.append( "\nTEST FAILED\n" );
                        break;
                    case DYNAMIC_UNREGISTRATION_TEST:
                        testerForm.append( "\nTEST PASSED\n" );
                        break;
                    default:
                        testerForm.append( "\nTest incompleted\n" );
                        break;
                }
                testFinished();
                break;
            case SERVICE_SEARCH_DEVICE_NOT_REACHABLE:
                testerForm.append( "\nCould not establish a connection to the remote device" );
                if( !serviceFound )
                {
                    testerForm.append( "\nService not found" );
                }
                testerForm.append( "\nTest incompleted\n" );
                testFinished();
                break;
        }
    }


    /** The same as DiscoveryListener.servicesDiscovered(int transID,
     * ServiceRecord[] servRecord). When the service is found, cancels the search to
     * continue with the test.
     * @param transID the transaction ID of the service search that is posting the result
     * @param record a list of services found during the search request
     */
```

```
    public void servicesDiscovered( int transID, ServiceRecord[] record )
    {

        remoteRecord = record[0];
        serviceFound = true;
        connectionUrl = remoteRecord.getConnectionURL( 0, false );

        testerForm.append( "Service found\n" );

        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 );
    }


    /** Set up the Service Record attributes to be retrieved and starts the service search
     */
    public void startServiceSearch()
    {
        testerForm.append( "\nSearching for the service ..." );

        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" );
        }
    }


//================================================================================
    // Protected Methods

    /** Retrieves preknown and cached devices, and insert it in the list of remote
     * devices.
     */
    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] );
            }
        }
    }


    /** Adds devices to the List which shows the devices
     * @param d the device to be added
     */
    protected void addRemoteDeviceToList( RemoteDevice d )
    {
        String address = d.getBluetoothAddress();

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


    /** Establishes a connection to the server and, depending on the test, send data to it.
     */
    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:
                        int numberOfBytes = 1000000;
                        data = new byte[numberOfBytes];

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

                        break;

                    case ACCUMULATION_BUFFER_TEST:
                        testData = "123";
                        data = testData.getBytes();

                        // send some data
                        output.write( data );
```

```
                        output.flush();

                        // close connection
                        output.close();
                        connection.close();

                        // open a new connection
                        connection = ( StreamConnection )Connector.open( connectionUrl );
                        output = connection.openOutputStream();

                        testData = "456";
                        data = testData.getBytes();
                        // send some data again
                        output.write( data );
                        output.flush();

                        break;

                    case DISCARD_BUFFER_TEST:
                        data = testData.getBytes();

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

                        break;

                    case DISC_CLOSE_BUFFER_TEST:
                        data = testData.getBytes();

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

                        break;
                }

                output.close();
            }

            // close connection
            connection.close();

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


    /** Initializes the client bluetooth system by retrieving the LocalDevice and
     * DiscoveryAgent objects
     * @return True if the initialization was sucesfully or false in other case.
     */
    protected boolean bluetoothClientInit()
    {
        try {
            local = LocalDevice.getLocalDevice();
            agent = local.getDiscoveryAgent();
            return true;
        }
        catch ( BluetoothStateException e ) {
            Alert error = new Alert( "Error", "Bluetooth system could not be initialized",
null, AlertType.ERROR );
            error.setTimeout( Alert.FOREVER );
            display.setCurrent( error );
            return false;
        }
    }

    /** Initializes the Displayable objects to be used.
     */
```

```
    protected void displayableInit()
    {
        testerForm = new Form( "Test Client" );
        testerForm.addCommand( new Command( "Quit", Command.EXIT, 1 ) );
        testerForm.addCommand( new Command( "Test Case", Command.ITEM, 2 ) );
        testerForm.addCommand( new Command( "Clear Log", Command.ITEM, 2 ) );

        testerForm.setCommandListener( this );
        display.setCurrent( testerForm );

        testSelector = new TestSelector();
    }

    /** Shows a list of devices to select the server. Retrieve preknown and cached
     * devices and start an inquiry.
     */
    protected void selectTestServer()
    {
        deviceVector = new Vector();
        isInInquiry = false;

        deviceList = new List( "Select server", List.IMPLICIT );
        deviceList.addCommand( new Command( "Quit", Command.EXIT, 1 ) );
        deviceList.setCommandListener( this );
        display.setCurrent( deviceList );

        addDevices();

        try {
            agent.startInquiry( DiscoveryAgent.GIAC, this );
        }
        catch ( BluetoothStateException e ) {
            Alert error = new Alert( "Error", "Device does not allow now an inquiry ", null,
AlertType.ERROR );
            error.setTimeout( Alert.FOREVER );
            display.setCurrent( error, deviceList );
        }

        isInInquiry = true;
    }
}
```

MIDlet StaticRegistrationTest

```
/*
 * StaticRegistrationTest.java
 *
 * Created on 21. Mai 2005, 12:31
 */

//=============================================================================
// Import Statements

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;


//=============================================================================
// CLASS (OR INTERFACE) DECLARATIONS

/** This MIDlet performs the Static Registration Test. The MIDlet checks whether
 * the connection exists in the PushRegistry. The connection should be statically
 * registered when the MIDlet was installed.
 *
 * @author Carlos Guerrero
 */
```

```
public class StaticRegistrationTest
    extends MIDlet
    implements CommandListener
{


//============================================================================
// Class (static) variables

    /** Service UUID to be used. By default is 2000000000001000800000006057028c19
     */
    public static String serviceUUID = "2000000000001000800000006057028c19";


//============================================================================
// Instance variables

    private Display display;
    private Form form;
    // URL to use for the connection
    private String connectionUrl;


//============================================================================
// Constructor

    /** Creates a StaticRegistrationTest object.
     */
    public StaticRegistrationTest()
    {
        display = Display.getDisplay( this );
        form = new Form( "Server" );
        display.setCurrent( form );

        form.addCommand( new Command( "Quit", Command.EXIT, 1 ) );
        form.setCommandListener( this );
    }


//============================================================================
// Public Methods

    /** startApp() method of the MIDlet object
     */
    public void startApp()
    {
        String[] connections = null;
        connectionUrl = "btspp://:" + serviceUUID;

        form.append( "\n------------------" );
        form.append( "\n\nStatic Push Registration Test:\n" );

        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");
                }
                else
                {
                    form.append("\nAnother connection found: " + connections[i]);
                }
            }
```

```
        }
        else
        {
            form.append("\nCannot find the entry in Push Registry");
            form.append("\nTest FAILED");
        }
    }


    /** pauseApp() method of the MIDlet object
     */
    public void pauseApp()
    {
    }


    /** destroyApp() method of the MIDlet object
     * @param unconditional  */
    public void destroyApp( boolean unconditional )
    {
    }


    /** commandAction() method of the CommandListener interface implemented by
     * PushConnectorOpenTest
     */
    public void commandAction( Command command, Displayable displayable )
    {
        if( command.getCommandType() == Command.EXIT ) {
            destroyApp( false );
            notifyDestroyed();
        }
    }
}
```

MIDlet PushUnregistrationTest

```
/*
 * PushUnregistrationTest.java
 *
 * Created on April 21, 2005, 1:00 PM
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;

/**
 *
 * @author  caguerre
 * @version
 */
public class PushUnregistrationTest
    extends MIDlet
    implements CommandListener {

    /** Service Name to be registered in the Push Registry. By default is PushService
     */
    public static String serviceName = "PushService";
    /** Service UUID to be used. By default is 20000000000010008000006057028C19
     */
    public static String serviceUUID = "20000000000010008000006057028C19";

    private Display display;
    private Form form;
    // URL to use for the connection
```

```
    private String connectionUrl;
    boolean testPassed;

    public PushUnregistrationTest() {
        display = Display.getDisplay(this);
        form = new Form("Server");
        display.setCurrent(form);

        form.addCommand( new Command( "Quit", Command.EXIT, 1 ) );
        form.setCommandListener( this );
    }

    public void startApp() {
        // initialization of variables
        connectionUrl = "btspp://localhost:" + serviceUUID + ";name=" + serviceName;
        testPassed = true;

        form.append("\n-------------------");
        form.append("\n\nDynamic Push Unregistration:\n");

        try {
            PushRegistry.unregisterConnection(connectionUrl);
            testPassed = false;
        } catch(SecurityException e) {
            form.append("\nSecurityException OK");
            testPassed = true;
        }

        if(testPassed) {
            form.append("\nTest PASSED");
        } else {
            form.append("\nTest FAILED");
        }
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    /** commandAction() method of the CommandListener interface implemented by
     * PushUnregistrationTest
     */
    public void commandAction( Command command, Displayable displayable )
    {
        if( command.getCommandType() == Command.EXIT ) {
            destroyApp( false );
            notifyDestroyed();
        }
    }

}
```

MIDlet PushConnectorOpenTest

```
/*
 * PushConnectorOpenTest.java
 *
 * Created on April 20, 2005, 3:14 PM
 */

//================================================================================
// Import Statements

import javax.microedition.midlet.*;
```

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


//============================================================================
// CLASS (OR INTERFACE) DECLARATIONS

/** This MIdlet performs the Connector Open Test. This test only checks wheter an
 * IOException is thrown when a MIDlet tries to open an already registered (by
 * another MIDlet) push conection.
 *
 * @author Carlos Guerrero
 */

public class PushConnectorOpenTest
    extends MIDlet
    implements CommandListener
{


//============================================================================
// Class (static) variables

    /** Service Name to be registered in the Push Registry. By default is PushService
     */
    public static String serviceName = "PushService";
    /** Service UUID to be used. By default is 2000000000001000800000606057028C19
     */
    public static String serviceUUID = "2000000000001000800000606057028C19";


//============================================================================
// Instance variables

    private Display display;
    private Form form;
    // URL to use for the connection
    private String connectionUrl;
    private StreamConnectionNotifier notifier;


//============================================================================
// Constructor

    /** Creates a PushConnectorOpenTest object.
     */
    public PushConnectorOpenTest()
    {
        display = Display.getDisplay( this );
        form = new Form( "Server" );
        display.setCurrent( form );

        form.addCommand( new Command( "Quit", Command.EXIT, 1 ) );
        form.setCommandListener( this );
    }


//============================================================================
// Public Methods

    /** startApp() method of the MIDlet object
     */
    public void startApp()
    {
        connectionUrl = "btspp://localhost:" + serviceUUID + ";name=" + serviceName;

        form.append( "\n-------------------" );
        form.append( "\n\nPush Connector Open Test:\n" );
```

```
        try
        {
            notifier = ( StreamConnectionNotifier )Connector.open( connectionUrl );
            form.append( "\nNo Exception: Test FAILED" );
            notifier.close();
        }
        catch( IOException e )
        {
            form.append( "\nIOException: Test PASSED" );
        }
    }


    /** pauseApp() method of the MIDlet object
     */
    public void pauseApp()
    {
    }


    /** destroyApp() method of the MIDlet object
     * @param unconditional  */
    public void destroyApp( boolean unconditional )
    {
    }


    /** commandAction() method of the CommandListener interface implemented by
     * PushConnectorOpenTest
     */
    public void commandAction( Command command, Displayable displayable )
    {
        if( command.getCommandType() == Command.EXIT ) {
            destroyApp( false );
            notifyDestroyed();
        }
    }
}
```

F) Documentación Java

Class PushRegistryTest

```
java.lang.Object
  └ javax.microedition.midlet.MIDlet
      └ PushRegistryTest
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener

---

public class **PushRegistryTest**
extends javax.microedition.midlet.MIDlet
implements javax.microedition.lcdui.CommandListener
Main class. Subclass of MIDlet class. Shows a list where the user can choose server or client application

---

Nested Class Summary	
class	**PushRegistryTest.SelectionList** List where the user can select Server/Client

Constructor Summary	
PushRegistryTest() Constructs a PushRegistryTest object	

Method Summary	
void	**commandAction**(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable) commandAction() method of the CommandListener interface implemented by PushRegistryTest
void	**destroyApp**(boolean unconditional) destroyApp() method of the MIDlet object
void	**exitMIDlet**() Calls destroyApp() and notifyDestroyed() when MIDlet exits
javax.microedition.midlet.MIDlet	**getMIDlet**() Returns the MIDlet object of this application
void	**pauseApp**() pauseApp() method of the MIDlet object
void	**startApp**() startApp() method of the MIDlet object

Methods inherited from class javax.microedition.midlet.MIDlet
checkPermission, getAppProperty, notifyDestroyed, notifyPaused, platformRequest, resumeRequest

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PushRegistryTest

```
public PushRegistryTest()
```

Constructs a PushRegistryTest object

Method Detail

startApp

```
public void startApp()
```

startApp() method of the MIDlet object

---

pauseApp

```
public void pauseApp()
```

pauseApp() method of the MIDlet object

---

destroyApp

```
public void destroyApp(boolean unconditional)
```

destroyApp() method of the MIDlet object

Parameters:

`unconditional` -

---

commandAction

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

commandAction() method of the CommandListener interface implemented by PushRegistryTest

Specified by:

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

---

exitMIDlet

```
public void exitMIDlet()
```

Calls destroyApp() and notifyDestroyed() when MIDlet exits

---

getMIDlet

```
public javax.microedition.midlet.MIDlet getMIDlet()
```

Returns the MIDlet object of this application

Returns:

the MIDlet object

Class PushRegistryTest.SelectionList

```
java.lang.Object
  └ javax.microedition.lcdui.Displayable
      └ javax.microedition.lcdui.Screen
          └ javax.microedition.lcdui.List
              └ PushRegistryTest.SelectionList
```

All Implemented Interfaces:
javax.microedition.lcdui.Choice, javax.microedition.lcdui.CommandListener
Enclosing class:
PushRegistryTest

---

public class **PushRegistryTest.SelectionList**
extends javax.microedition.lcdui.List
implements javax.microedition.lcdui.CommandListener
List where the user can select Server/Client

Field Summary

Fields inherited from class javax.microedition.lcdui.List
SELECT_COMMAND

Fields inherited from interface javax.microedition.lcdui.Choice
EXCLUSIVE, IMPLICIT, MULTIPLE, POPUP, TEXT_WRAP_DEFAULT, TEXT_WRAP_OFF, TEXT_WRAP_ON

Constructor Summary

PushRegistryTest.SelectionList(PushRegistryTest p) Constructs a SelectionList	

Method Summary

void	**commandAction**(javax.microedition.lcdui.Command c, javax.microedition.lcdui.Displayable d) commandAction() method of the CommandListener interface implemented by SelectionList

Methods inherited from class javax.microedition.lcdui.List
append, delete, deleteAll, getFitPolicy, getFont, getImage, getSelectedFlags, getSelectedIndex, getString, insert, isSelected, removeCommand, set, setFitPolicy, setFont, setSelectCommand, setSelectedFlags, setSelectedIndex, size

Methods inherited from class javax.microedition.lcdui.Displayable
addCommand, getHeight, getTicker, getTitle, getWidth, isShown, setCommandListener, setTicker,

setTitle, sizeChanged

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PushRegistryTest.SelectionList

```
public PushRegistryTest.SelectionList(PushRegistryTest p)
```

Constructs a SelectionList

Parameters:

`p` - PushRegistry object unique to this MIDlet.

Method Detail

commandAction

```
public void commandAction(javax.microedition.lcdui.Command c,
                          javax.microedition.lcdui.Displayable d)
```

commandAction() method of the CommandListener interface implemented by SelectionList

Specified by:

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

Class PushRegistryTester

```
java.lang.Object
  └ PushRegistryTester
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener, java.lang.Runnable
Direct Known Subclasses:
PushRegistryTestClient, PushRegistryTestServer

---

public class **PushRegistryTester**
extends java.lang.Object
implements java.lang.Runnable, javax.microedition.lcdui.CommandListener
Class to perform Push Registry Test. Both server and client have to be a subclass of this class.

Nested Class Summary

protected class	**PushRegistryTester.TestSelector** TestSelector is a List which displays the test cases that can be selected to be performed.

Field Summary

protected static int	**ACCUMULATION_BUFFER_TEST** ID of the Accumulation Data Buffer Test.
protected static int	**BT_INIT_TEST** ID of the Data BT Init Test.
protected static int	**CONNECTION_CLOSE_TEST** ID of the Connection Close Test.
protected static int	**DISC_CLOSE_BUFFER_TEST** ID of the Discard Close Data Buffer Test.
protected static int	**DISCARD_BUFFER_TEST** ID of the discard Data Buffer Test.
protected static int	**DYNAMIC_REGISTRATION_TEST** ID of the Dynamic Push Registration Test.
protected static int	**DYNAMIC_UNREGISTRATION_TEST** ID of the Dynamic Push Unregistration Test.
protected static int	**INCOMING_CONNECTION_TEST** ID of the incoming Connection Test.
protected static int	**LIST_CONNECTIONS_TEST** ID of the List Connections Test.
static java.lang.String	**midletClassName** Name of the MIDlet to be registered in the Push Registry.
protected static int	**MODIFY_SERVICE_TEST** ID of the Modify Service Test.
protected static int	**NO_TEST_RUNNING** Indicates that no test is running at the moment.
protected static int	**NOT_ALLOWED_TEST** ID of the Not Allowed Test.
protected static int	**NOTIFIER_CLOSE_TEST** ID of the Data Notifier Close Test.
protected PushRegistryTest	**prObject** PushRegistry object unique to this MIDlet.
static java.lang.String	**pushFilter** Push filter to be used in the Push Registration.
protected static int	

	ROBUSTNESS_BUFFER_TEST ID of the Robustness Data Buffer Test.
protected static int	**serviceAttributeId** ID of a service attribute used for testing the Service Record
protected static java.lang.String	**serviceAttributeValue** Value of a service attribute used for testing the Service Record
static java.lang.String	**serviceName** Service Name to be registered in the Push Registry.
static java.lang.String	**serviceUUID** Service UUID to be used.
protected static int	**SIMPLE_BUFFER_TEST** ID of the Data Buffer Test.
protected static int	**STATIC_REGISTRATION_TEST** ID of the Static Push Registration Test.
protected static int	**STATIC_UNREGISTRATION_TEST** ID of the Static Push Unregistration Test.
protected static java.lang.String	**testData** Data to be sended
protected javax.microedition.lcdui.Form	**testerForm** Form to be used by both applications (server and client).
protected boolean	**testIsRunning** Variable used to know if the test is running
protected int	**testPhase** Some tests pause to allow the user perform some actions, so the test has several phases. testPhase indicates which phase is the next to be performed.
protected static int	**USE_CONNECTION_TEST** ID of the Use Connection Test.

Constructor Summary

PushRegistryTester(PushRegistryTest p) Creates a PushRegistryTester.	

Method Summary

void	**commandAction**(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable) The same as CommandListener.commandAction (javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)
protected javax.microedition.lcdui.Display	**getDisplay**() Gets the Display object that is unique to this MIDlet.
int	**getTestCase**() Get the test case running at the moment.
void	**run**() run() method of the Thread.
void	**runTest**(int tC) Starts the test case to be performed.
void	**setTestCase**(int testCaseID) Set the test case to be performed.
void	**testFinished**() Indicates that the test case has finished.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

NO_TEST_RUNNING

```
protected static final int NO_TEST_RUNNING
```

Indicates that no test is running at the moment. The value is 0.

See Also:
Constant Field Values

STATIC_REGISTRATION_TEST

```
protected static final int STATIC_REGISTRATION_TEST
```

ID of the Static Push Registration Test. The value is 1

See Also:
Constant Field Values

STATIC_UNREGISTRATION_TEST

```
protected static final int STATIC_UNREGISTRATION_TEST
```

ID of the Static Push Unregistration Test. The value is 2

See Also:
Constant Field Values

DYNAMIC_REGISTRATION_TEST

```
protected static final int DYNAMIC_REGISTRATION_TEST
```

ID of the Dynamic Push Registration Test. The value is 3

See Also:
Constant Field Values

DYNAMIC_UNREGISTRATION_TEST

```
protected static final int DYNAMIC_UNREGISTRATION_TEST
```

ID of the Dynamic Push Unregistration Test. The value is 4

See Also:
Constant Field Values

INCOMING_CONNECTION_TEST

```
protected static final int INCOMING_CONNECTION_TEST
```

ID of the incoming Connection Test. The value is 5

See Also:
Constant Field Values

BT_INIT_TEST

```
protected static final int BT_INIT_TEST
```

ID of the Data BT Init Test. The value is 6

See Also:

Constant Field Values

LIST_CONNECTIONS_TEST

```
protected static final int LIST_CONNECTIONS_TEST
```

ID of the List Connections Test. The value is 7

See Also:

Constant Field Values

MODIFY_SERVICE_TEST

```
protected static final int MODIFY_SERVICE_TEST
```

ID of the Modify Service Test. The value is 8

See Also:

Constant Field Values

USE_CONNECTION_TEST

```
protected static final int USE_CONNECTION_TEST
```

ID of the Use Connection Test. The value is 9

See Also:

Constant Field Values

CONNECTION_CLOSE_TEST

```
protected static final int CONNECTION_CLOSE_TEST
```

ID of the Connection Close Test. The value is 10

See Also:

Constant Field Values

SIMPLE_BUFFER_TEST

```
protected static final int SIMPLE_BUFFER_TEST
```

ID of the Data Buffer Test. The value is 11

See Also:

Constant Field Values

NOTIFIER_CLOSE_TEST

```
protected static final int NOTIFIER_CLOSE_TEST
```

ID of the Data Notifier Close Test. The value is 12

See Also:

Constant Field Values

ROBUSTNESS_BUFFER_TEST

```
protected static final int ROBUSTNESS_BUFFER_TEST
```

ID of the Robustness Data Buffer Test. The value is 13

See Also:

Constant Field Values

ACCUMULATION_BUFFER_TEST

```
protected static final int ACCUMULATION_BUFFER_TEST
```

ID of the Accumulation Data Buffer Test. The value is 14

See Also:

Constant Field Values

DISCARD_BUFFER_TEST

```
protected static final int DISCARD_BUFFER_TEST
```

ID of the discard Data Buffer Test. The value is 15

See Also:

Constant Field Values

DISC_CLOSE_BUFFER_TEST

```
protected static final int DISC_CLOSE_BUFFER_TEST
```

ID of the Discard Close Data Buffer Test. The value is 16

See Also:

Constant Field Values

NOT_ALLOWED_TEST

```
protected static final int NOT_ALLOWED_TEST
```

ID of the Not Allowed Test. The value is 17

See Also:

Constant Field Values

---

midletClassName

```
public static java.lang.String midletClassName
```

Name of the MIDlet to be registered in the Push Registry. By default is PushRegistryTest.

---

serviceUUID

```
public static java.lang.String serviceUUID
```

Service UUID to be used. By default is 2000000000001000800000057028C19

---

pushFilter

```
public static java.lang.String pushFilter
```

Push filter to be used in the Push Registration. By default is *

---

serviceName

```
public static java.lang.String serviceName
```

Service Name to be registered in the Push Registry. By default is PushService

---

serviceAttributeId

```
protected static int serviceAttributeId
```

ID of a service attribute used for testing the Service Record

---

serviceAttributeValue

```
protected static java.lang.String serviceAttributeValue
```

Value of a service attribute used for testing the Service Record

---

testData

```
protected static java.lang.String testData
```

Data to be sended

---

testIsRunning

```
protected boolean testIsRunning
```

Variable used to know if the test is running

---

testPhase

```
protected int testPhase
```

Some tests pause to allow the user perform some actions, so the test has several phases. testPhase indicates which phase is the next to be performed.

---

prObject

```
protected PushRegistryTest prObject
```

PushRegistry object unique to this MIDlet.

---

testerForm

```
protected javax.microedition.lcdui.Form testerForm
```

Form to be used by both applications (server and client).

Constructor Detail

PushRegistryTester

```
public PushRegistryTester(PushRegistryTest p)
```

Creates a PushRegistryTester.

Parameters:

`p` - PushRegistry object unique to this MIDlet.

Method Detail

run

```
public void run()
```

run() method of the Thread.

Specified by:

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

---

runTest

```
public void runTest(int tC)
```

Starts the test case to be performed.

Parameters:

`tC` - Test case ID. NO_TEST_RUNNING, STATIC_REGISTRATION_TEST, STATIC_UNREGISTRATION_TEST, DYNAMIC_REGISTRATION_TEST, DYNAMIC_UNREGISTRATION_TEST, INCOMING_CONNECTION_TEST, LIST_CONNECTIONS_TEST, MODIFY_SERVICE_TEST, USE_CONNECTION_TEST, CONNECTION_CLOSE_TEST, SIMPLE_BUFFER_TEST, BT_INIT_TEST, NOTIFIER_CLOSE_TEST, ROBUSTNESS_BUFFER_TEST, ACCUMULATION_BUFFER_TEST, DISCARD_BUFFER_TEST, DISC_CLOSE_BUFFER_TEST

---

getTestCase

```
public int getTestCase()
```

Get the test case running at the moment.

Returns:

Test case ID.

---

setTestCase

```
public void setTestCase(int testCaseID)
```

Set the test case to be performed.

Parameters:

`testCaseID` - The test case ID.

---

testFinished

```
public void testFinished()
```

Indicates that the test case has finished. This method has to be called at the end of every test case.

---

commandAction

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

The same as CommandListener.commandAction(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)

Specified by:

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

Parameters:

`command` - a Command object identifying the command.

`displayable` - the Displayable on which this event has occurred.

---

getDisplay

```
protected javax.microedition.lcdui.Display getDisplay()
```

Gets the Display object that is unique to this MIDlet.

Returns:

the display object that application can use for its user interface.

Class PushRegistryTester.TestSelector

```
java.lang.Object
  └ javax.microedition.lcdui.Displayable
      └ javax.microedition.lcdui.Screen
          └ javax.microedition.lcdui.List
              └ PushRegistryTester.TestSelector
```

All Implemented Interfaces:
javax.microedition.lcdui.Choice, javax.microedition.lcdui.CommandListener
Enclosing class:
PushRegistryTester

---

protected class **PushRegistryTester.TestSelector**
extends javax.microedition.lcdui.List
implements javax.microedition.lcdui.CommandListener
TestSelector is a List which displays the test cases that can be selected to be performed.

Field Summary

Fields inherited from class javax.microedition.lcdui.List
SELECT_COMMAND

Fields inherited from interface javax.microedition.lcdui.Choice
EXCLUSIVE, IMPLICIT, MULTIPLE, POPUP, TEXT_WRAP_DEFAULT, TEXT_WRAP_OFF, TEXT_WRAP_ON

Constructor Summary

PushRegistryTester.TestSelector() Creates a TestSelector with the test cases that can be performed.	

Method Summary

void	**commandAction**(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable) The same as CommandListener.commandAction(javax.microedition.lcdui.
void	**setThis**() Sets the TestSelector as the current Displayable and CommandListener.

Methods inherited from class javax.microedition.lcdui.List
append, delete, deleteAll, getFitPolicy, getFont, getImage, getSelectedFlags, getSelectedIndex, getString, insert, isSelected, removeCommand, set, setFitPolicy, setFont, setSelectCommand, setSelectedFlags, setSelectedIndex, size

Methods inherited from class javax.microedition.lcdui.Displayable
addCommand, getHeight, getTicker, getTitle, getWidth, isShown, setCommandListener, setTicker, setTitle, sizeChanged

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PushRegistryTester.TestSelector

```
public PushRegistryTester.TestSelector()
```

Creates a TestSelector with the test cases that can be performed.

Method Detail

setThis

```
public void setThis()
```

Sets the TestSelector as the current Displayable and CommandListener.

---

commandAction

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

The same as CommandListener.commandAction(javax.microedition.lcdui. Command command, javax.microedition.lcdui.Displayable displayable)

Specified by:

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

Parameters:

`command` - a Command object identifying the command.
`displayable` - the Displayable on which this event has occurred.

Class PushRegistryTestServer

```
java.lang.Object
  └ PushRegistryTester
      └ PushRegistryTestServer
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener, java.lang.Runnable

---

public class **PushRegistryTestServer**
extends PushRegistryTester
The class implements the server to perform the Push Registry Test

---

Nested Class Summary

protected class	**PushRegistryTestServer.AcceptAndOpenThread** This Thread class call acceptAndOpen method.
protected class	**PushRegistryTestServer.MessageInformation** This class is an Alert which is shown to the user when he has to perform some action.

Nested classes inherited from class PushRegistryTester

PushRegistryTester.TestSelector

Field Summary

Fields inherited from class PushRegistryTester

ACCUMULATION_BUFFER_TEST, BT_INIT_TEST, CONNECTION_CLOSE_TEST, DISC_CLOSE_BUFFER_TEST, DISCARD_BUFFER_TEST, DYNAMIC_REGISTRATION_TEST, DYNAMIC_UNREGISTRATION_TEST, INCOMING_CONNECTION_TEST, LIST_CONNECTIONS_TEST, midletClassName, MODIFY_SERVICE_TEST, NO_TEST_RUNNING, NOT_ALLOWED_TEST, NOTIFIER_CLOSE_TEST, prObject, pushFilter, ROBUSTNESS_BUFFER_TEST, serviceAttributeId, serviceAttributeValue, serviceName, serviceUUID, SIMPLE_BUFFER_TEST, STATIC_REGISTRATION_TEST, STATIC_UNREGISTRATION_TEST, testData, testIsRunning, testPhase, USE_CONNECTION_TEST

Constructor Summary

PushRegistryTestServer(PushRegistryTest p) Creates a PushRegistryTestServer.	

Method Summary

void	**accumulationBufferTest**() Performs the Accumulation Buffering Test.
boolean	**bluetoothServerInit**() Initializes the server bluetooth system by retrieving the LocalDevice object.
void	**commandAction**(javax.microedition.lcdui.Command c, javax.microedition.lcdui.Displayable d) Same as CommandListener.commandAction(javax.microedition.lcdui.Command command,

	javax.microedition.lcdui.Displayable displayable)
void	**connectionCloseTest**() Performs the Connection Close Test.
void	**discardBufferTest**() Performs the Discard Buffering Test.
void	**discCloseBufferTest**() Performs the Discard Close Buffering Test.
void	**dynamicPushRegistrationTest**() Performs the Dynamic Push Registration Test: 1.
void	**dynamicPushUnregistrationTest**() Performs the Dynamic Push Unregistration Test.
boolean	**existsInRegistry**() Check whether the MIDlet exists in Push Registry.
void	**incomingConnectionTest**() Performs the Incoming Connection Test.
void	**listConnectionsTest**() Performs the List Connections Test.
void	**modifyServiceTest**() Performs the List Connections Test.
void	**notAllowedTest**() Performs the Not Allowed Test
void	**notifierCloseTest**() Performs the Notifier Close Test.
boolean	**registerTestConnection**(boolean security) Performs a dynamic Unregister Connection scenario by calling PushRegistry.registerConnection().
void	**robustnessBufferTest**() Performs the Robustness Buffering Test.
void	**run**() run() method of the Thread.
void	**runTest**(int tC) Starts the test case to be performed.
protected void	**showInformation**(java.lang.String infoString) Shows an Alert window.
void	**simpleBufferTest**() Performs the Simple Buffering Test.
void	**startBluetoothServer**() Starts the server by calling Connector.open and acceptAndOpen().
void	**staticPushUnregistrationTest**() Performs the Static Push Unregistration Test.
boolean	**unregisterTestConnection**() Performs a dynamic Unregister Connection scenario by calling PushRegistry.unregisterConnection().
void	**useConnectionTest**() Performs the Use Connection Test.

Methods inherited from class PushRegistryTester
getDisplay, getTestCase, setTestCase, testFinished

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PushRegistryTestServer

```
public PushRegistryTestServer(PushRegistryTest p)
```

Creates a PushRegistryTestServer.

Parameters:
`p` - PushRegistryTest object unique to this MIDlet.

Method Detail

run

```
public void run()
```

run() method of the Thread. This method is started when the thread starts. Therefore the variables and objects are initialized in this method.

Specified by:
`run` in interface `java.lang.Runnable`

Overrides:
`run` in class `PushRegistryTester`

---

commandAction

```
public void commandAction(javax.microedition.lcdui.Command c,
                          javax.microedition.lcdui.Displayable d)
```

Same as CommandListener.commandAction(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)

Specified by:
`commandAction` in interface `javax.microedition.lcdui.CommandListener`

Overrides:
`commandAction` in class `PushRegistryTester`

Parameters:
`c` - a Command object identifying the command.
`d` - the Displayable on which this event has occurred.

---

runTest

```
public void runTest(int tC)
```

Starts the test case to be performed. Depending on the test case passed as parameter, runTest() starts the method which performs the specified test.

Overrides:
`runTest` in class `PushRegistryTester`

Parameters:
`tC` - Constant which indicates the test case to be performed. NO_TEST_RUNNING, STATIC_UNREGISTRATION_TEST, STATIC_REGISTRATION_TEST, DYNAMIC_UNREGISTRATION_TEST, DYNAMIC_REGISTRATION_TEST.

---

staticPushUnregistrationTest

```
public void staticPushUnregistrationTest()
```

Performs the Static Push Unregistration Test.

---

dynamicPushRegistrationTest

```
public void dynamicPushRegistrationTest()
```

Performs the Dynamic Push Registration Test: 1. Try to register the connection with a wrong name to check whether ClassNotFoundException is thrown. 2. Register the connection. 3. Check whether the connection exists in PushRegistry 4. Try to register the connection again and an IOException should be thrown.

---

dynamicPushUnregistrationTest

```
public void dynamicPushUnregistrationTest()
```

Performs the Dynamic Push Unregistration Test.

---

incomingConnectionTest

```
public void incomingConnectionTest()
```

Performs the Incoming Connection Test. Does nothing because this test is suposed to be performed only on the client side.

---

useConnectionTest

```
public void useConnectionTest()
```

Performs the Use Connection Test. Calls connector.open() and acceptAndOpen() if the user accept.

---

connectionCloseTest

```
public void connectionCloseTest()
```

Performs the Connection Close Test. Calls connector.open() and acceptAndOpen().

---

simpleBufferTest

```
public void simpleBufferTest()
```

Performs the Simple Buffering Test. Calls connector.open() and acceptAndOpen().

---

robustnessBufferTest

```
public void robustnessBufferTest()
```

Performs the Robustness Buffering Test. Calls connector.open() and acceptAndOpen().

---

accumulationBufferTest

```
public void accumulationBufferTest()
```

Performs the Accumulation Buffering Test. Calls connector.open() and acceptAndOpen().

---

discardBufferTest

```
public void discardBufferTest()
```

Performs the Discard Buffering Test. Calls connector.open() and acceptAndOpen().

---

discCloseBufferTest

```
public void discCloseBufferTest()
```

Performs the Discard Close Buffering Test. Test calls connector.open() and acceptAndOpen().

---

listConnectionsTest

```
public void listConnectionsTest()
```

Performs the List Connections Test.

---

modifyServiceTest

```
public void modifyServiceTest()
```

Performs the List Connections Test. Creates a new attribute for the Service Record and update it.

---

notifierCloseTest

```
public void notifierCloseTest()
```

Performs the Notifier Close Test. Test calls connector.open() and acceptAndOpen().

---

notAllowedTest

```
public void notAllowedTest()
```

Performs the Not Allowed Test

---

bluetoothServerInit

```
public boolean bluetoothServerInit()
```

Initializes the server bluetooth system by retrieving the LocalDevice object.

Returns:

True if the initialization was sucesfully or false in other case.

---

startBluetoothServer

```
public void startBluetoothServer()
```

Starts the server by calling Connector.open and acceptAndOpen().

---

existsInRegistry

```
public boolean existsInRegistry()
```

Check whether the MIDlet exists in Push Registry.

Returns:

True if the MIDlet entry exists in the Push Registry and false if not.

---

unregisterTestConnection

```
public boolean unregisterTestConnection()
```

Performs a dynamic Unregister Connection scenario by calling PushRegistry.unregisterConnection().

Returns:

True if the unregistration was sucesfully done or false in other case.

---

registerTestConnection

```
public boolean registerTestConnection(boolean security)
```

Performs a dynamic Unregister Connection scenario by calling PushRegistry.registerConnection().

Parameters:

`security` - if true the connection require security

Returns:

True if the registration was sucesfully done or false in other case.

---

showInformation

```
protected void showInformation(java.lang.String infoString)
```

Shows an Alert window.

Parameters:

`infoString` - the String to be shown on the Alarm window

Class PushRegistryTestServer.AcceptAndOpenThread

```
java.lang.Object
  └ java.lang.Thread
      └ PushRegistryTestServer.AcceptAndOpenThread
```

All Implemented Interfaces:
java.lang.Runnable
Enclosing class:
PushRegistryTestServer

---

protected class **PushRegistryTestServer.AcceptAndOpenThread**
extends java.lang.Thread
This Thread class call acceptAndOpen method.

Field Summary

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary

PushRegistryTestServer.AcceptAndOpenThread() Constructor	

Method Summary

void	**run**() run method Start AcceeptAndOpen and wait on Exception or connection.

Methods inherited from class java.lang.Thread
activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getContextClassLoader, getName, getPriority, getThreadGroup, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setName, setPriority, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

PushRegistryTestServer.AcceptAndOpenThread

```
public PushRegistryTestServer.AcceptAndOpenThread()
```

Constructor

Method Detail

run

```
public void run()
```

run method Start AcceeptAndOpen and wait on Exception or connection. When the connection is established several actions should be performed depending on the test case.

Class PushRegistryTestServer.MessageInformation

```
java.lang.Object
  └ javax.microedition.lcdui.Displayable
      └ javax.microedition.lcdui.Screen
          └ javax.microedition.lcdui.Alert
              └ PushRegistryTestServer.MessageInformation
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener
Enclosing class:
PushRegistryTestServer

---

protected class **PushRegistryTestServer.MessageInformation**
extends javax.microedition.lcdui.Alert
implements javax.microedition.lcdui.CommandListener
This class is an Alert which is shown to the user when he has to perform some action. After it the user is suposed to press the OK button and the test can continue.

Field Summary

Fields inherited from class javax.microedition.lcdui.Alert
DISMISS_COMMAND, FOREVER

Constructor Summary

PushRegistryTestServer.MessageInformation(java.lang.String msg) Contructs a MessageInformation object	

Method Summary

void	**commandAction**(javax.microedition.lcdui.Command c, javax.microedition.lcdui.Displayable d) commandAction of the CommandListener interface which is implemented by the MessageInformation class.

Methods inherited from class javax.microedition.lcdui.Alert
addCommand, getDefaultTimeout, getImage, getIndicator, getString, getTimeout, getType, removeCommand, setCommandListener, setImage, setIndicator, setString, setTimeout, setType

Methods inherited from class javax.microedition.lcdui.Displayable
getHeight, getTicker, getTitle, getWidth, isShown, setTicker, setTitle, sizeChanged

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PushRegistryTestServer.MessageInformation

```
public PushRegistryTestServer.MessageInformation(java.lang.String msg)
```

Contructs a MessageInformation object

Parameters:

`msg` - Message to be shown.

Method Detail

commandAction

```
public void commandAction(javax.microedition.lcdui.Command c,
                          javax.microedition.lcdui.Displayable d)
```

commandAction of the CommandListener interface which is implemented by the MessageInformation class.

Specified by:

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

Parameters:

`c` - a Command object identifying the command.

`d` - the Displayable on which this event has occurred.

Class PushRegistryTestClient

```
java.lang.Object
  └ PushRegistryTester
      └ PushRegistryTestClient
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener, javax.bluetooth.DiscoveryListener, java.lang.Runnable

---

public class **PushRegistryTestClient**
extends PushRegistryTester
implements javax.bluetooth.DiscoveryListener
This class implements the client to perform the Push Registry Tests.

Nested Class Summary

Nested classes inherited from class PushRegistryTester
PushRegistryTester.TestSelector

Field Summary

protected static int	**SEARCH_AND_CONNECT** Specifies that the test on the client will try to find the service on the server, and if it is found, connect to it.
protected static int	**SEARCH_AND_SEND** Specifies that the test on the client will try to find the service on the server, and if it is found, connect to it and send data.
protected static int	**SEARCH_SERVICE** Specifies that the test on the client will try to find the service on the server.

Fields inherited from class PushRegistryTester
ACCUMULATION_BUFFER_TEST, BT_INIT_TEST, CONNECTION_CLOSE_TEST, DISC_CLOSE_BUFFER_TEST, DISCARD_BUFFER_TEST, DYNAMIC_REGISTRATION_TEST, DYNAMIC_UNREGISTRATION_TEST, INCOMING_CONNECTION_TEST, LIST_CONNECTIONS_TEST, midletClassName, MODIFY_SERVICE_TEST, NO_TEST_RUNNING, NOT_ALLOWED_TEST, NOTIFIER_CLOSE_TEST, prObject, pushFilter, ROBUSTNESS_BUFFER_TEST, serviceAttributeId, serviceAttributeValue, serviceName, serviceUUID, SIMPLE_BUFFER_TEST, STATIC_REGISTRATION_TEST, STATIC_UNREGISTRATION_TEST, testData, testIsRunning, testPhase, USE_CONNECTION_TEST

Fields inherited from interface javax.bluetooth.DiscoveryListener
INQUIRY_COMPLETED, INQUIRY_ERROR, INQUIRY_TERMINATED, SERVICE_SEARCH_COMPLETED, SERVICE_SEARCH_DEVICE_NOT_REACHABLE, SERVICE_SEARCH_ERROR, SERVICE_SEARCH_NO_RECORDS, SERVICE_SEARCH_TERMINATED

Constructor Summary

PushRegistryTestClient(PushRegistryTest p) Creates a PushRegistryTestClient.	

Method Summary

protected void	**addDevices**() Retrieves preknown and cached devices, and insert it in the list of remote devices.
protected void	**addRemoteDeviceToList**(javax.bluetooth.RemoteDevice d) Adds devices to the List which shows the devices
protected boolean	**bluetoothClientInit**() Initializes the client bluetooth system by retrieving the LocalDevice and DiscoveryAgent objects
void	**commandAction**(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable) The same as CommandListener.commandAction(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)
protected void	**connectToServerTest**() Establishes a connection to the server and, depending on the test, send data to it.
void	**deviceDiscovered**(javax.bluetooth.RemoteDevice remoteDevice, javax.bluetooth.DeviceClass deviceClass) The same as DiscoveryListener.deviceDiscovered(RemoteDevice btDevice, DeviceClass cod).
protected void	**displayableInit**() Initializes the Displayable objects to be used.
void	**inquiryCompleted**(int type) The same as DiscoveryListener.inquiryCompleted(int discType).
void	**run**() run() method of the Thread.
void	**runTest**(int tC) Starts the test case to be performed.
protected void	**selectTestServer**() Shows a list of devices to select the server.
void	**servicesDiscovered**(int transID, javax.bluetooth.ServiceRecord[] record) The same as DiscoveryListener.servicesDiscovered(int transID, ServiceRecord[] servRecord).
void	**serviceSearchCompleted**(int transID, int type) The same as DiscoveryListener.serviceSearchCompleted(int transID, int respCode).
void	**startServiceSearch**() Set up the Service Record attributes to be retrieved and starts the service search

Methods inherited from class PushRegistryTester

getDisplay, getTestCase, setTestCase, testFinished

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

SEARCH_SERVICE

```
protected static final int SEARCH_SERVICE
```

Specifies that the test on the client will try to find the service on the server.

See Also:

Constant Field Values

SEARCH_AND_CONNECT

```
protected static final int SEARCH_AND_CONNECT
```

Specifies that the test on the client will try to find the service on the server, and if it is found, connect to it.

See Also:

Constant Field Values

SEARCH_AND_SEND

```
protected static final int SEARCH_AND_SEND
```

Specifies that the test on the client will try to find the service on the server, and if it is found, connect to it and send data. The data to send depend on the test case.

See Also:

Constant Field Values

Constructor Detail

PushRegistryTestClient

```
public PushRegistryTestClient(PushRegistryTest p)
```

Creates a PushRegistryTestClient.

Parameters:

`p` - PushRegistryTest object unique to this MIDlet.

Method Detail

run

```
public void run()
```

run() method of the Thread. This method is started when the thread starts. Initializes the BT client and calls startTestServer() which shows a list of devices to select the server.

Specified by:

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

Overrides:

`run` in class `PushRegistryTester`

---

runTest

```
public void runTest(int tC)
```

Starts the test case to be performed. Specifies what has to do the client depending on the test case.

Overrides:

`runTest` in class `PushRegistryTester`

Parameters:

`tC` - Constant which indicates the test case to be performed. NO_TEST_RUNNING, STATIC_UNREGISTRATION_TEST, STATIC_REGISTRATION_TEST, DYNAMIC_UNREGISTRATION_TEST, DYNAMIC_REGISTRATION_TEST.

---

commandAction

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

The same as CommandListener.commandAction(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)

Specified by:
`commandAction` in interface `javax.microedition.lcdui.CommandListener`
Overrides:
`commandAction` in class `PushRegistryTester`
Parameters:
`command` - a Command object identifying the command.
`displayable` - the Displayable on which this event has occurred.

---

deviceDiscovered

```
public void deviceDiscovered(javax.bluetooth.RemoteDevice remoteDevice,
                             javax.bluetooth.DeviceClass deviceClass)
```

The same as DiscoveryListener.deviceDiscovered(RemoteDevice btDevice, DeviceClass cod). When a device is discovered, is added to the list of remote devices.
Specified by:
`deviceDiscovered` in interface `javax.bluetooth.DiscoveryListener`
Parameters:
`remoteDevice` - the device that was found during the inquiry
`deviceClass` - the service classes, major device class, and minor device class of the remote device

---

inquiryCompleted

```
public void inquiryCompleted(int type)
```

The same as DiscoveryListener.inquiryCompleted(int discType). The inquiry could be terminated by the user or ends normally.
Specified by:
`inquiryCompleted` in interface `javax.bluetooth.DiscoveryListener`
Parameters:
`type` - the type of request that was completed; either INQUIRY_COMPLETED, INQUIRY_TERMINATED, or INQUIRY_ERROR

---

serviceSearchCompleted

```
public void serviceSearchCompleted(int transID,
                                   int type)
```

The same as DiscoveryListener.serviceSearchCompleted(int transID, int respCode). Depending on the test case the client should have found the service or not, has to connect to the service, or has to check the retrieved Service Record.
Specified by:
`serviceSearchCompleted` in interface `javax.bluetooth.DiscoveryListener`
Parameters:
`transID` - the transaction ID identifying the request which initiated the service search
`type` - the response code that indicates the status of the transaction

---

servicesDiscovered

```
public void servicesDiscovered(int transID,
                               javax.bluetooth.ServiceRecord[] record)
```

The same as DiscoveryListener.servicesDiscovered(int transID, ServiceRecord[] servRecord). When the service is found, cancels the search to continue with the test.
Specified by:
`servicesDiscovered` in interface `javax.bluetooth.DiscoveryListener`
Parameters:
`transID` - the transaction ID of the service search that is posting the result
`record` - a list of services found during the search request

---

startServiceSearch

```
public void startServiceSearch()
```

Set up the Service Record attributes to be retrieved and starts the service search

---

addDevices

```
protected void addDevices()
```

Retrieves preknown and cached devices, and insert it in the list of remote devices.

---

addRemoteDeviceToList

```
protected void addRemoteDeviceToList(javax.bluetooth.RemoteDevice d)
```

Adds devices to the List which shows the devices

Parameters:

`d` - the device to be added

---

connectToServerTest

```
protected void connectToServerTest()
```

Establishes a connection to the server and, depending on the test, send data to it.

---

bluetoothClientInit

```
protected boolean bluetoothClientInit()
```

Initializes the client bluetooth system by retrieving the LocalDevice and DiscoveryAgent objects

Returns:

True if the initialization was sucesfully or false in other case.

---

displayableInit

```
protected void displayableInit()
```

Initializes the Displayable objects to be used.

---

selectTestServer

```
protected void selectTestServer()
```

Shows a list of devices to select the server. Retrieve preknown and cached devices and start an inquiry.

Class StaticRegistrationTest

```
java.lang.Object
  └ javax.microedition.midlet.MIDlet
      └ StaticRegistrationTest
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener

---

public class **StaticRegistrationTest**
extends javax.microedition.midlet.MIDlet
implements javax.microedition.lcdui.CommandListener
This MIDlet performs the Static Registration Test. The MIDlet checks whether the connection exists in the PushRegistry. The connection should be statically registered when the MIDlet was installed.

Field Summary

static java.lang.String	**serviceUUID** Service UUID to be used.

Constructor Summary

StaticRegistrationTest() Creates a StaticRegistrationTest object.	

Method Summary

void	**commandAction**(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable) commandAction() method of the CommandListener interface implemented by PushConnectorOpenTest
void	**destroyApp**(boolean unconditional) destroyApp() method of the MIDlet object
void	**pauseApp**() pauseApp() method of the MIDlet object
void	**startApp**() startApp() method of the MIDlet object

Methods inherited from class javax.microedition.midlet.MIDlet

checkPermission, getAppProperty, notifyDestroyed, notifyPaused, platformRequest, resumeRequest

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

serviceUUID

```
public static java.lang.String serviceUUID
```

Service UUID to be used. By default is 2000000000001000800000057028c19

Constructor Detail

StaticRegistrationTest

```
public StaticRegistrationTest()
```

Creates a StaticRegistrationTest object.

Method Detail

startApp

```
public void startApp()
```

startApp() method of the MIDlet object

---

pauseApp

```
public void pauseApp()
```

pauseApp() method of the MIDlet object

---

destroyApp

```
public void destroyApp(boolean unconditional)
```

destroyApp() method of the MIDlet object

Parameters:

`unconditional` -

---

commandAction

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

commandAction() method of the CommandListener interface implemented by PushConnectorOpenTest

Specified by:

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

Class PushUnregistrationTest

```
java.lang.Object
  └ javax.microedition.midlet.MIDlet
      └ PushUnregistrationTest
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener

---

public class **PushUnregistrationTest**
extends javax.microedition.midlet.MIDlet
implements javax.microedition.lcdui.CommandListener

Field Summary	
`static java.lang.String`	**`serviceName`** Service Name to be registered in the Push Registry.
`static java.lang.String`	**`serviceUUID`** Service UUID to be used.

Constructor Summary	
`PushUnregistrationTest``()`	

Method Summary	
`void`	**`commandAction`**`(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)` commandAction() method of the CommandListener interface implemented by PushUnregistrationTest
`void`	**`destroyApp`**`(boolean unconditional)`
`void`	**`pauseApp`**`()`
`void`	**`startApp`**`()`

Methods inherited from class javax.microedition.midlet.MIDlet
`checkPermission, getAppProperty, notifyDestroyed, notifyPaused, platformRequest, resumeRequest`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

serviceName

```
public static java.lang.String serviceName
```

Service Name to be registered in the Push Registry. By default is PushService

---

serviceUUID

```
public static java.lang.String serviceUUID
```

Service UUID to be used. By default is 2000000000001000800000057028C19

Constructor Detail

PushUnregistrationTest

```
public PushUnregistrationTest()
```

Method Detail

startApp

```
public void startApp()
```

---

pauseApp

```
public void pauseApp()
```

---

destroyApp

```
public void destroyApp(boolean unconditional)
```

---

commandAction

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

commandAction() method of the CommandListener interface implemented by PushUnregistrationTest

Specified by:

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

Class PushConnectorOpenTest

```
java.lang.Object
  └ javax.microedition.midlet.MIDlet
      └ PushConnectorOpenTest
```

All Implemented Interfaces:
javax.microedition.lcdui.CommandListener

---

public class **PushConnectorOpenTest**
extends javax.microedition.midlet.MIDlet
implements javax.microedition.lcdui.CommandListener
This MIdlet performs the Connector Open Test. This test only checks wheter an IOException is thrown when a MIDlet tries to open an already registered (by another MIDlet) push conection.

Field Summary	
`static java.lang.String`	**`serviceName`** Service Name to be registered in the Push Registry.
`static java.lang.String`	**`serviceUUID`** Service UUID to be used.

Constructor Summary	
`PushConnectorOpenTest()` Creates a PushConnectorOpenTest object.	

Method Summary	
`void`	**`commandAction`**`(javax.microedition.lcdui.Command command, javax.microedition.lcdui.Displayable displayable)` commandAction() method of the CommandListener interface implemented by PushConnectorOpenTest
`void`	**`destroyApp`**`(boolean unconditional)` destroyApp() method of the MIDlet object
`void`	**`pauseApp`**`()` pauseApp() method of the MIDlet object
`void`	**`startApp`**`()` startApp() method of the MIDlet object

Methods inherited from class javax.microedition.midlet.MIDlet
`checkPermission, getAppProperty, notifyDestroyed, notifyPaused, platformRequest, resumeRequest`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

serviceName

```
public static java.lang.String serviceName
```

Service Name to be registered in the Push Registry. By default is PushService

---

serviceUUID

```
public static java.lang.String serviceUUID
```

Service UUID to be used. By default is 2000000000001000800000006057028C19

Constructor Detail

PushConnectorOpenTest

```
public PushConnectorOpenTest()
```

Creates a PushConnectorOpenTest object.

Method Detail

startApp

```
public void startApp()
```

startApp() method of the MIDlet object

---

pauseApp

```
public void pauseApp()
```

pauseApp() method of the MIDlet object

---

destroyApp

```
public void destroyApp(boolean unconditional)
```

destroyApp() method of the MIDlet object

Parameters:

`unconditional` -

---

commandAction

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

commandAction() method of the CommandListener interface implemented by PushConnectorOpenTest

Specified by:

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