OSGI

PARTE II – Aprendiendo OSGI mediante Ejemplos

Roberto Montero Miguel

INDICE

I. Especificación OSGI, explicada mediante ejemplos.

En esta nueva sección del tutorial explicaremos mediante ejemplos la teoría descrita anteriormente, además entraremos en detalle en las especificaciones de algunos servicios OSGI (System Services).

Como comentamos en la primera entrega, intentaremos basar este tutorial en un caso real: Control Remoto de una Calefacción.

Antes de entrar en detalle con el caso real y los servicios predefinidos por OSGI, tenemos que tener muy claro algunos conceptos sobre los que se basará cualquier desarrollo sobre OSGI:

- **Classloader de OSGI**: Al igual que en cualquier entorno JAVA, en OSGI el tener claro el orden de carga de las clases, servirá de gran ayuda para cualquier desarrollador.
- **Servicios OSGI**: Es de vital importancia entender como se maneja la arquitectura orientada a servicios de OSGI.

1. Entendiendo los classloader de OSGI

En la anterior entrega de esta serie de artículos sobre OSGI, explicábamos teóricamente los pasos que realiza OSGI para cargar una clase. El orden de búsqueda de clases (classloader) lo plasmábamos con la siguiente figura:

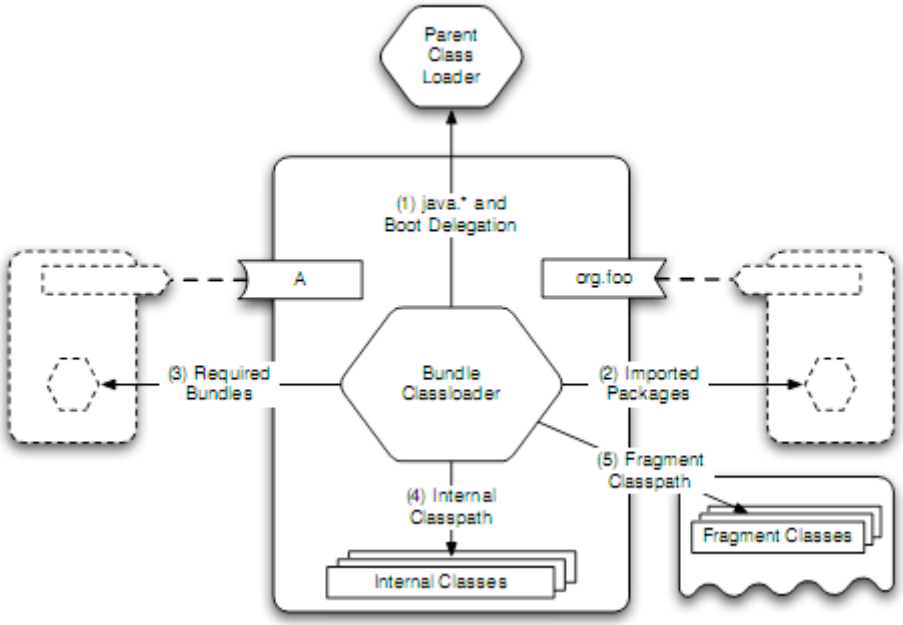


Figura 1.1 – Orden de Búsqueda de clases

A pesar que el orden de carga de las clases, lo explicábamos en el artículo anterior, vamos a repasarlo brevemente, para a continuación entender este proceso mediante un ejemplo:

1. Comprueba si el paquete a importar comienza por el patrón java.* o si el paquete aparece en la lista de la propiedad "org.osgi.framework.bootdelegation". En este caso el classloader delega al classloader superior (al igual que se realiza en las aplicaciones tradicionales). Con la propiedad "bootdelegation" le sugerimos al framework que busque las dependencias compuestas por clases y recursos, primeramente en el "boot classpath" antes de delegar en la carga normal de OSGI.

2. Comprobar si el paquete se encuentra importado en el archivo "manifiest" del bundle.

3. Comprobar si la clase a buscar se encuentra en el paquete importado a través de la cláusula Required-Bundle del archivo Manifiest.

4. El bundle comprueba si la clase buscada se encuentra entre sus clases o librerías internas.

5. Comprueba si las clases se encuentra entre las "Fragment Classes".

El ejemplo practico que hemos preparado se compone de varios bundles y consiste en que si colocamos la misma clase en varios bundles y cada uno de estos bundles se corresponde en un lugar del classloader como se muestra en la figura anterior (posiciones 2,3,4,5), podremos comprobar cual de las cuatro clases iguales prevalece sobre la otra. Por lo tanto tendremos los siguientes bundles:

- Bundle **org.javahispano.importpackage** que simplemente exporta en su archivo manifest la clase *org.javahispano.testclassloader.TestClassLoader*. Esta clase en su constructor tiene un System.out que muestra la traza: "SALUDO DESDE IMPORT PACKAGE".

- Bundle **org.javahispano.requiredbundle** que en su manifest exporta dos clases: *org.javahispano.testclassloader.TestClassLoader* y *org.javahispano.testclassloader2.TestClassLoaderRequiredBundle*.

- **Librería .jar** que solo contiene una clase: *org.javahispano.testclassloader.TestClassLoader* y al cargar su constructor nos mostrará la siguiente traza: "SALUDO DESDE INTERNAL CLASSPATH" .

- Fragment Bundle llamado **org.javahispano.fragmentbundle** que será anexado al Host principal y que solo contiene la clase *org.javahispano.testclassloader.TestClassLoader* que muestra la traza "SALUDO DESDE FRAGMENT BUNDLE".

- Bundle llamado **org.javahispano.classloader.test** que en su archivo manisfest tiene un import-package de la clase *org.javahispano.testclassloader.TestClassLoader* y un required-bundle del bundle **org.javahispano.requiredbunle**. Así mismo al classpath interno de este modulo le hemos añadido una librería que también contiene la clase *org.javahispano.testclassloader.TestClassLoader*. Por si fuera poco, a este bundle también le hemos anexado un "fragment bundle" llamado **org.javahispano.fragmentbunle**. En el activator del bundle Host, invocamos mediante reflection a las clases *org.javahispano.testclassloader.TestClassLoader* y *org.javahispano.testclassloader2.TestClassLoaderRequiredBundle*.

En la siguiente imagen podremos ver la arquitectura de los bundles de nuestro ejemplo:

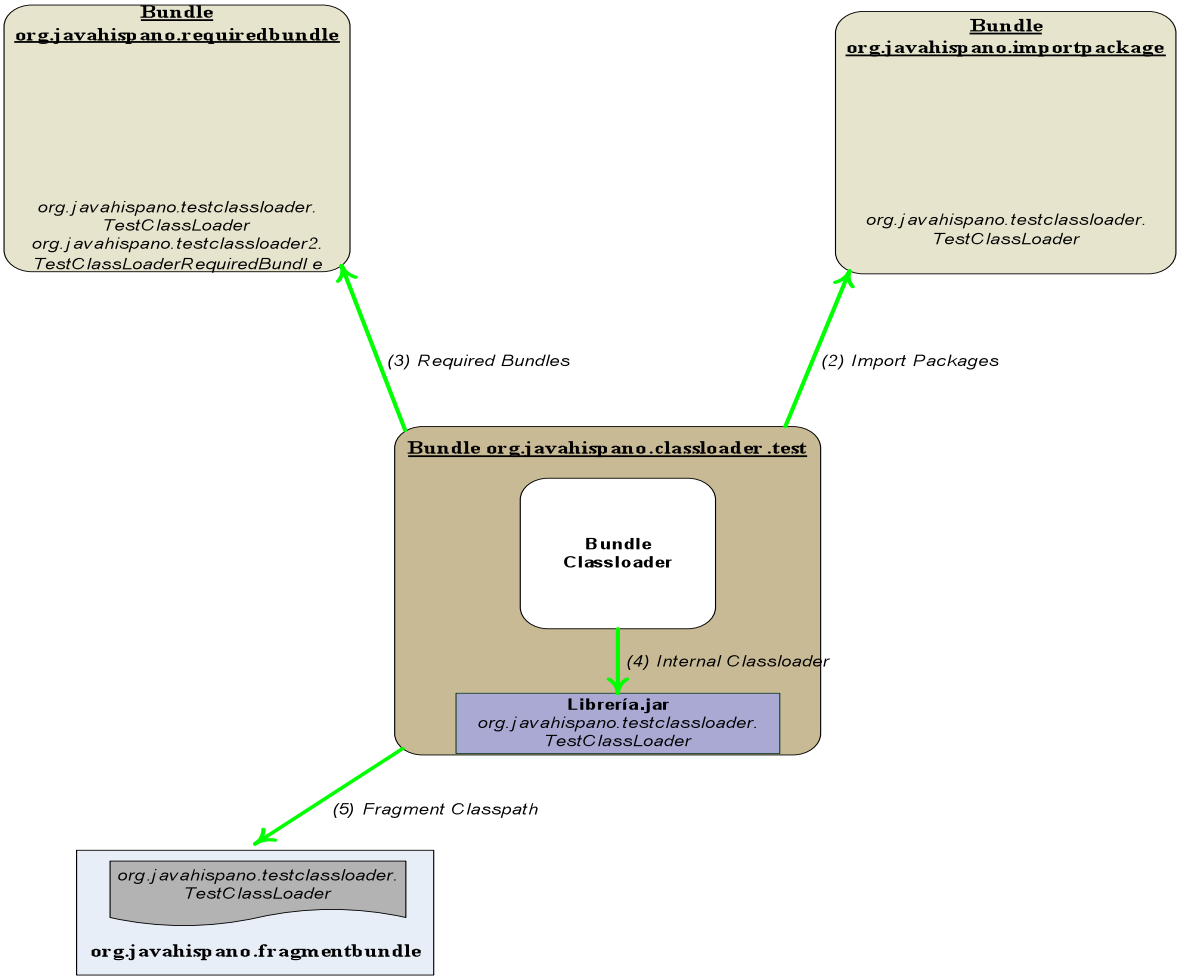
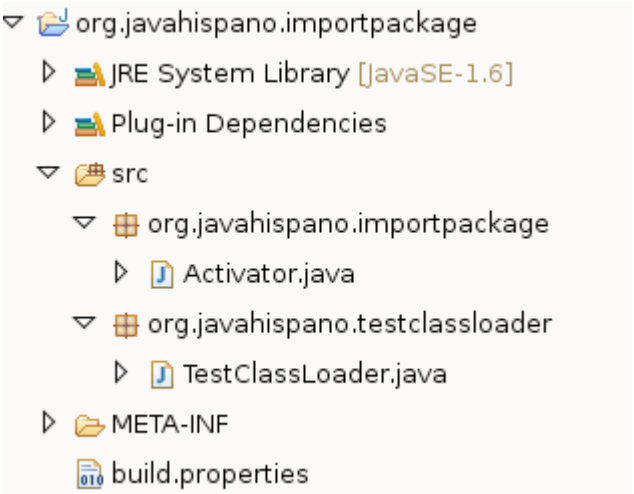


Figura 1.2 – Orden de Búsqueda de clases Ejemplo

Todos estos bundles los he construido usando Eclipse como IDE de desarrollo (Están subidos XXXXX), desde la vista "package explorer" de eclipse tendremos la siguiente estructura:



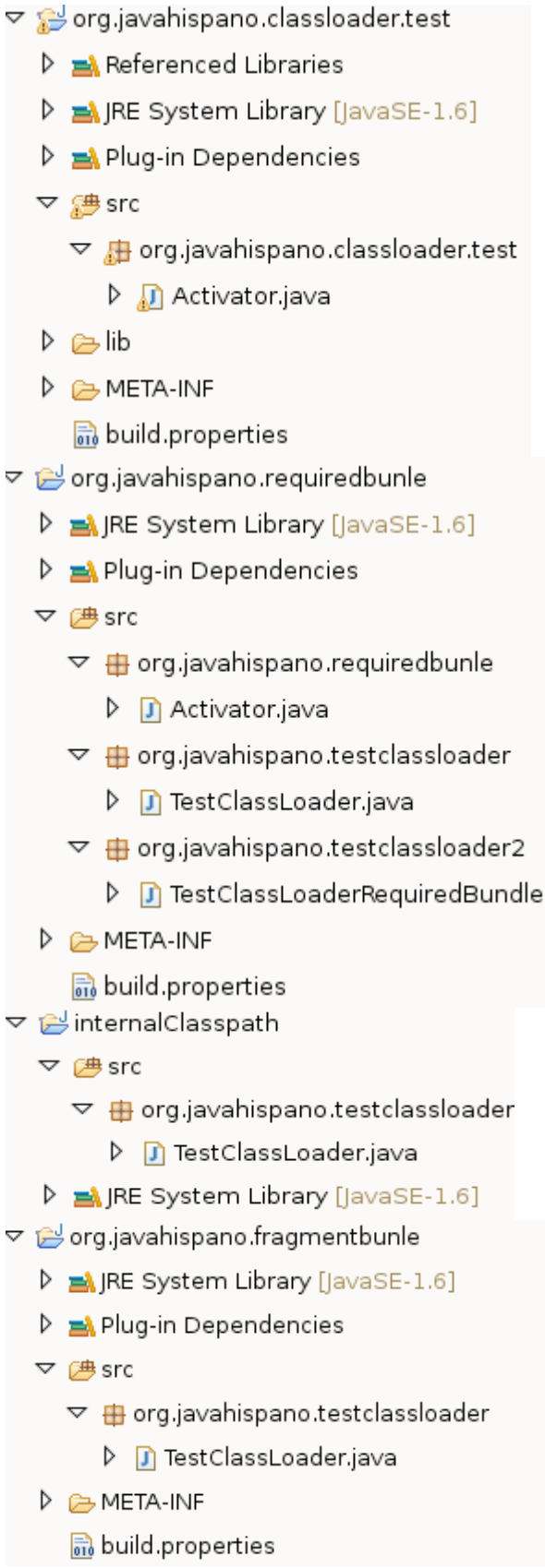


Figura 1.3 – Ejemplo Classloader. Estructura Eclipse

A continuación presentaremos brevemente las clases mas importantes de todos estos bundles:

Bundle: org.javahispano.importpackage

TestClassLoader.java

```
package org.javahispano.testclassloader;
public class TestClassLoader {
    public TestClassLoader(){
            System.out.println("SALUDO DESDE IMPORT PACKAGE");
    }
}
```

Manifest.mf

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Importpackage
Bundle-SymbolicName: org.javahispano.importpackage
Bundle-Version: 1.0.0
Bundle-Activator: org.javahispano.importpackage.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: org.javahispano.testclassloader
Import-Package: org.osgi.framework
```

Listado 1.1 - Bundle org.javahispano.importpackage

Bundle: org.javahispano.requiredbundle

TestClassLoader.java

```
package org.javahispano.testclassloader;

public class TestClassLoader {
    public TestClassLoader(){
            System.out.println("SALUDO DESDE REQUIRED BUNDLE");
    }
}
```

TestClassLoaderRequiredBundle.java

```
package org.javahispano.testclassloader2;

public class TestClassLoaderRequiredBundle {
    public TestClassLoaderRequiredBundle(){
```

```
        System.out.println("Clase TestClassLoaderRequiredBundle:::SALUDO
DESDE REQUIRED BUNDLE");
    }
}
```

Manifest.mf

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Requiredbunle
Bundle-SymbolicName: org.javahispano.requiredbunle
Bundle-Version: 1.0.0
Bundle-Activator: org.javahispano.requiredbunle.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.framework
Export-Package: org.javahispano.testclassloader,
 org.javahispano.testclassloader2
```

Listado 1.2 - Bundle org.javahispano.requiredbundle

Librería: internalClasspath

Se trata de un proyecto JAVA de Eclipse, que contiene la clase *TestClassLoader*, que será exportada como una librería JAR al classpath interno del bundle *org.javahispano.classloader.test*.

```
package org.javahispano.testclassloader;

public class TestClassLoader {
    public TestClassLoader(){
        System.out.println("SALUDO DESDE INTERNAL CLASSPATH");
    }
}
```

Listado 1.3 - Librería InternalClasspath

Fragment Bundle: org.javahispano.fragmentbunle

Cuando en el artículo anterior explicábamos los principios básicos de OSGI, también explicamos teóricamente en que consisten este tipo de bundles, por definirlos en pocas palabras podríamos decir que son "trozos" de bundle que se anexan a un bundle principal o Host. Este tipo de bundles nos ayudan cuando nuestro bundle principal incorpora código nativo dependiente del entorno. En vez de tener que crear un bundle para cada sistema operativo donde queramos que funcione, sacaremos las dependencias del código nativo a un fragment bundle y lo anexaremos al bundle principal.

En la siguiente figura podremos ver la arquitectura de un desarrollo típico multiplataforma con código nativo dependiente del entorno:

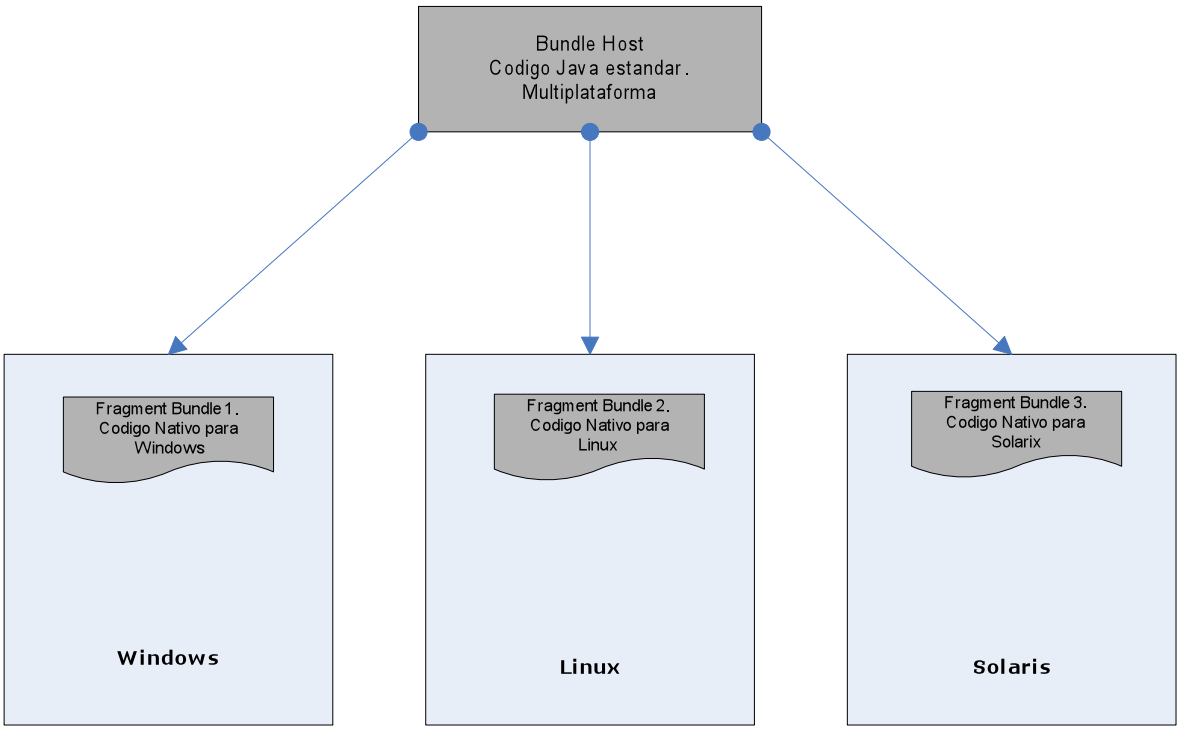


Figura 1.4 – Ejemplo Classloader. Fragment Bundles

TestClassLoader.java

```
package org.javahispano.testclassloader;

public class TestClassLoader {
    public TestClassLoader(){
            System.out.println("SALUDO DESDE FRAGMENT BUNDLE");
    }
}
```

Manifest.mf

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Fragmentbunle
Bundle-SymbolicName: org.javahispano.fragmentbunle
Bundle-Version: 1.0.0
Fragment-Host: org.javahispano.classloader.test;bundle-version="1.0.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Listado 1.4 - Fragment Bundle

Una de las características de los fragment bundles es que en tiempo de compilación, desde el bundle Host no se puede acceder a las clases del fragment-bundle. Por eso las invocaciones desde el bundle Host se hacen mediante el mecanismo de reflexión. En cambio, desde el fragment bundle se puede hacer uso de todas las clases del bundle Host.

org.javahispano.classloader.test

Este será el bundle que invocará a las clases TestClassLoader y TestClassLoaderRequiredBundle, a través del mecanismo de reflexión.

Activator.java

```
package org.javahispano.classloader.test;

import java.lang.reflect.Constructor;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator{
   public void start(BundleContext context) throws Exception {
    try{
        Class cls =
        Class.forName("org.javahispano.testclassloader.TestClassLoader");
        Constructor ct  = cls.getConstructor();
         ct.newInstance();
    }catch(ClassNotFoundException e){
          System.out.println("No existe la clase
            org.javahispano.testclassloader.TestClassLoader");
    }
    try{
        Class clsRequiredBundle =
Class.forName("org.javahispano.testclassloader2.TestClassLoaderRequiredBundle");
                        Constructor ct  = clsRequiredBundle.getConstructor();
                        ct.newInstance();
     }catch(ClassNotFoundException e){
            System.out.println("No existe la clase
             org.javahispano.testclassloader2.TestClassLoaderRequiredBundle");
    }
   }
   public void stop(BundleContext context) throws Exception {
   }
}
```

Manifest.mf

```
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.javahispano.testclassloader,
```

```
org.osgi.framework
Bundle-ClassPath: lib/internalClasspath.jar,
.
Require-Bundle: org.javahispano.requiredbunle;bundle-version="1.0.0"
```

Listado 1.5 - Bundle de Test

1.1 Desplegando el juego de pruebas

Para realizar esta práctica, he utilizado la plataforma de Equinox, copiando todos estos bundles a la carpeta EQUINOX_HOME/classloader/plugins y editando el fichero config.ini de la siguiente manera:

```
osgi.clean=true
eclipse.ignoreApp=true

org.osgi.framework.bootdelegation=javax.jms
osgi.bundles=../classloader/plugins/org.javahispano.importpackage_1.0.0@start, \
../classloader/plugins/org.javahispano.requiredbunle_1.0.0.jar@start, \
../classloader/plugins/org.javahispano.classloader.test_1.0.0.jar@start, \
../classloader/plugins/org.javahispano.fragmentbunle_1.0.0.jar, \
```

Listado 1.6 - config.ini

Cuando ejecutemos la plataforma, obtendremos la siguiente salida:

```
osgi> SALUDO DESDE IMPORT PACKAGE
Clase TestClassLoaderRequiredBundle:::SALUDO DESDE REQUIRED BUNDLE
```

Con la salida anteriormente mostrada, podremos deducir **que ha predominado el classpath del import-package sobre todos los demás classpath**. Ha tomado la cabecera import-package para importar la clase TestClassLoader (Ha predominado sobre la cabecera required-bundle e internal-classpath), como en ningún otro bundle existe la clase TestClassLoaderRequiredBundle, la ha cogido de la cláusula required-bundle, nótese que el paquete de esta clase (org.javahispano.testclassloader2) no esta importado mediante la cabecera import-package, esto quiere decir, que cuando indicamos en nuestro manifest una cabecera requiered-bundle: nombre-bundle, se importarán todos los paquetes del bundle señalado en dicha cláusula.

La siguiente prueba veremos como predomina la clausula required-bundle sobre la clausula Bundle-ClassPath, para ello modificaremos el meta-inf del bundle org.javahispano.classloader.test, quitando la clausula import-package:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Test
Bundle-SymbolicName: org.javahispano.classloader.test
Bundle-Version: 1.0.0
Bundle-Activator: org.javahispano.classloader.test.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
~~Import-Package: org.osgi.framework, org.javahispano.testclassloader~~
Bundle-ClassPath: lib/internalClasspath.jar, .
Require-Bundle: org.javahispano.requiredbunle;bundle-version="1.0.0"
```

Listado 1.7 - manifest.mf

También eliminaremos el bundle org.javahispano.importpackage del config.ini:

```
osgi.clean=true
eclipse.ignoreApp=true

osgi.bundles=../classloader/plugins/org.javahispano.requiredbunle_1.0.0.jar@start, \
#../classloader/plugins/org.javahispano.importpackage_1.0.0@start, \
../classloader/plugins/org.javahispano.classloader.test_1.0.0.jar@start, \
../classloader/plugins/org.javahispano.fragmentbunle_1.0.0.jar, \
```

Listado 1.8 - config.ini

La salida que obtendremos es la siguiente:

```
osgi> SALUDO DESDE REQUIRED BUNDLE
Clase TestClassLoaderRequiredBundle:::SALUDO DESDE REQUIRED BUNDLE
```

Vemos como **ha predominado la cabecera Required-bundle sobre el classpath interno y el fragment bundle**.

Nuestra última prueba será demostrar **como predomina el classpath interno del bundle sobre el classpath de un fragment bundle**. Para ello eliminaremos la cláusula requiered-bundle del bundle de test:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Test
Bundle-SymbolicName: org.javahispano.classloader.test
Bundle-Version: 1.0.0
Bundle-Activator: org.javahispano.classloader.test.Activator
```

```
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.framework
Bundle-ClassPath: lib/internalClasspath.jar, .
Require-Bundle: org.javahispano.requiredbunle;bundle-version="1.0.0"
```

Listado 1.9 - manifest.mf

La salida que obtendremos:

```
osgi> SALUDO DESDE INTERNAL CLASSPATH
No existe la clase
org.javahispano.testclassloader2.TestClassLoaderRequiredBundle
```

Vemos como se ha ejecutado la clase de la librería interna del bundle. Finalmente, para demostrar que "no hay trampa ni cartón", cambiaremos el meta-inf del bundle de test para que se ejecute la clase del fragment-bundle. Para ello eliminamos la directiva Bundle-Classpath:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Test
Bundle-SymbolicName: org.javahispano.classloader.test
Bundle-Version: 1.0.0
Bundle-Activator: org.javahispano.classloader.test.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-ClassPath: lib/internalClasspath.jar, .
Import-Package: org.osgi.framework
```

Listado 1.10 - manifest.mf

La salida que obtendremos es la siguiente:

```
osgi> SALUDO DESDE FRAGMENT BUNDLE
No existe la clase
org.javahispano.testclassloader2.TestClassLoaderRequiredBundle
```

2. Trabajando con Servicios OSGI

En la entrega anterior vimos como podemos crear fácilmente un servicio y consumirlo desde otro bundle. En esta ocasión veremos como también es posible publicar y consumir servicios, utilizando algunos mecanismos como por ejemplo:

- **Eventos**: La plataforma OSGI genera eventos cada vez que un servicio es publicado o eliminado. Podremos usar este mecanismo para que nuestro bundle consumidor comience a usar un servicio, cuando este haya sido desplegado y al contrario, dejar de consumirlo cuando este sea detenido o eliminado.
- **Tracker**: Es un mecanismo para tener encapsulado el consumo de un servicio.
- **Service Factory**: Service Factory es un mecanismo especial, por el cual el bundle que expone el servicio puede tener controlado cuando y como se crean las instancias de los servicios. Si no utilizamos este mecanismo siempre se devolverá la misma instancia del servicio.
- **Servicios Declarativos**: OSGI nos permite registrar servicios de forma declarativa, editando ficheros de metadatos. Para quien este familiarizado con la inyección de dependencias de SPRING, podríamos decir que es algo similar.
- **Extensión de la línea de comandos de la consola:** Podremos crear nuevos comandos personalizados ejecutables desde consola. Con estos comandos podremos, por ejemplo, activar y desactivar nuestros servicios.

2.1 Eventos en los Servicios

Para ver claramente como funcionan los eventos y los listenners en OSGI, trataremos de crear un ejemplo sencillo basado en nuestro futuro controlador de calefacción, para ello necesitaremos al menos dos bundles:

- **org.javahispano.sensorreader** : Este bundle se encargará mas adelante de leer desde el puerto COM los datos que llegan desde el sensor de temperatura, así mismo publicará un servicio que de momento tendrá un único método getTemperatura.
- **org.javahispano.sensortest**: Solo se encargará de consumir el servicio expuesto por org.javahispano.sensorreader. En un futuro un bundle como este, podría almacenar el histórico de valores del sensor en una BD o tomar decisiones conforme a la temperatura leída etc...

2.1.1 Servicio org.javahispano.sensorreader

Este bundle será similar al servicio que construimos en la primera entrega de este tutorial, salvo que en vez de saludarnos, nos dará la temperatura leída del sensor.
De momento la temperatura que retornará este bundle será un numero aleatorio, mas adelante implementaremos la lectura del puerto COM - RS232.

Clase Activator:

```
package org.javahispano.sensorreader;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class Activator implements BundleActivator {
	private ServiceRegistration registration;

	public void start(BundleContext context) throws Exception {
		registration =
		 context.registerService(SensorTemperatura.class.getName(),
					new SensorTemperaturaImpl(), null);
		System.out.println("REGISTRADO SERVICIO");
	}
	public void stop(BundleContext context) throws Exception {
		registration.unregister();
	}
}
```

Listado 2.1.1-1 - Activator

Interfaz del Servicio:

```
package org.javahispano.sensorreader;

public interface SensorTemperatura {
	public double getTemp();
}
```

Listado 2.1.1-2 - Interfaz SensorTemperatura

Implementación del Servicio:

```
package org.javahispano.sensorreader;

public class SensorTemperaturaImpl implements SensorTemperatura{
	public double getTemp(){
		return  ((double) Math.random() * 100.5);
	}
}
```

Listado 2.1.1-3 - Implementación SensorTemperaturaImpl

No debemos olvidar exportar el paquete org.javahispano.sensorreader en nuestro MANIFEST.MF.

2.1.2 Consumidor org.javahispano.sensortest

Este bundle declarará un listener de eventos de los servicios de la plataforma y aplicará un filtro para seleccionar solo los eventos del servicio del sensor de temperatura. Una vez detectado que se ha registrado el servicio que estábamos esperando, se ejecutará un thread que pedirá cada X segundos el valor del sensor.

Clase Activator y Listenner:

```
package org.javahispano.sensortest;

import org.javahispano.sensorreader.SensorTemperatura;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceEvent;
import org.osgi.framework.ServiceListener;
import org.osgi.framework.ServiceReference;

public class Activator implements BundleActivator, ServiceListener{
private SensorTemperatura sensorTemperatura;
private BundleContext context;
private Thread testThread = null;

        public void start(BundleContext context) throws Exception {
           this.context = context;
    String filter = "(objectclass=" + SensorTemperatura.class.getName() + ")";
           context.addServiceListener(this, filter);
        }

        public void stop(BundleContext context) throws Exception {
                context.removeServiceListener(this);
        }

        @Override
        public void serviceChanged(ServiceEvent ev) {
            ServiceReference sr = ev.getServiceReference();
            switch(ev.getType()) {
            case ServiceEvent.REGISTERED:
             {
                   sensorTemperatura = (SensorTemperatura)context.getService(sr);
              testThread = new Thread( new TestSensorThread(sensorTemperatura));
```

```
                                testThread.start();
                        break;
                    }
                    case ServiceEvent.UNREGISTERING:
                    {
                            if(testThread != null){
                                testThread.interrupt();
                            }
                            break;
                    }
                    default:
                        break;
                    }
        }
}
```

Listado 2.1.2-1 - Activator

Clase TestSensorThread:

```
package org.javahispano.sensortest;

import org.javahispano.sensorreader.SensorTemperatura;

public class TestSensorThread implements Runnable {
        SensorTemperatura sensorTemperatura;
        public TestSensorThread(SensorTemperatura sensorTemperatura){
                this.sensorTemperatura = sensorTemperatura;
        }
        @Override
        public void run() {
                while (! Thread.interrupted() ){
                System.out.println("Temp: " + sensorTemperatura.getTemp());
                        try {
                                Thread.sleep(5000);
                        } catch (InterruptedException e) {
                                System.out.println("Thread Detenido");
                                sensorTemperatura = null;
                                break;
                        }
                }
        }
}
```

Listado 2.1.2-2 - TestSensorThread

Lo primero que nos llamará la atención al comenzar a revisar la clase *Activator*, será que además de implementar a la interfaz *BundleActivator*, hemos incluido la implementación a *org.osgi.framework.ServiceListener*.

Implementar la interfaz *ServiceListener* nos obliga a sobrescribir el método *serviceChanged*. Esté método será invocado por la plataforma cada vez que se cambie el estado de un servicio.

Para que el método serviceChanged no sea invocado cada vez que cambie de estado cualquier bundle de la plataforma, le hemos aplicado un filtro:

```
String filter = "(objectclass=" + SensorTemperatura.class.getName() + ")";
            context.addServiceListener(this, filter);
```

En la plataforma OSGI, los filtros son utilizados para muchos otros casos. Estos filtros se corresponden con la especificación RFC 1960.

2.1.3 Ejecutando nuestro ejemplo

Cuando vayamos a instalar sobre la plataforma estos dos bundles, tenemos que tener en cuenta que primeramente deberemos desplegar el bundle cliente que contiene el listener, a continuación cuando despleguemos el servicio, el listenner detectará el servicio y se empezarán a mostrar las trazas de la temperatura.

2.2 Tracker

Como he mencionado antes, un *ServiceTracker* puede ser un mecanismo de encapsular el consumo del servicio, pero además de eso puede funcionar de forma muy parecida al ServiceListenner, indicándonos cuando se ha arrancado o detenido el servicio que estamos consumiendo.

Para verlo mediante un ejemplo, tomaremos el bundle *org.javahispano.sensortest* y crearemos una nueva clase *TestSensorTracker* que implementara a la interfaz *org.osgi.util.tracker.ServiceTracker* (no olvidéis importar este paquete en el archivo MANIFEST.MF):

```
package org.javahispano.sensortest;

import org.javahispano.sensorreader.SensorTemperatura;
import org.osgi.framework.BundleContext;
import org.osgi.framework.BundleException;
import org.osgi.framework.ServiceReference;
```

```
import org.osgi.util.tracker.ServiceTracker;

public class TestSensorTracker extends ServiceTracker{

        private BundleContext context;
   public TestSensorTracker(BundleContext context) {
      super(context, SensorTemperatura.class.getName(),null);
      this.context = context;
   }
   public Object addingService(ServiceReference reference) {
      System.out.println("Añadiendo servicio SensorTemperatura");
      return super.addingService(reference);

   }
   public void removedService(ServiceReference reference, Object service) {
      System.out.println("Eliminando Servicio SensorTemperatura");
      super.removedService(reference, service);
      try {
                        this.context.getBundle().stop();
                } catch (BundleException e) {
                        //Si ya lo estamos deteniendo se producirá una excepción (en
caso de que paremos el bundle sensortest)
                        //En cambio si paramos el servicio, este bundle también se
detendrá
                }
   }
}
```

Listado 2.2-1 - TestSensorTracker

Implementar a la interfaz ServiceTracker, nos obliga a sobrescribir dos métodos addingService y removedService. Estos dos métodos serán invocados cuando:

- Se arranque o se pare el servicio consumido (addingService o removedService).
- Se arranque o se pare el bundle consumidor.

Para que esto funcione tendremos que modificar el Activator de nuestro bundle y obtener la referencia al servicio que queremos consumir a través del nuevo ServiceTracker:

```
package org.javahispano.sensortest;

import org.javahispano.sensorreader.SensorTemperatura;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
public class Activator implements BundleActivator{

	TestSensorTracker testSensorTracker;
	private Thread testThread = null;
	public void start(BundleContext context) throws Exception {
		testSensorTracker = new TestSensorTracker(context);
		testSensorTracker.open();
		SensorTemperatura sensorTemperatura =
(SensorTemperatura)testSensorTracker.getService();
		testThread = new Thread( new
TestSensorThread(sensorTemperatura));
		testThread.start();
	}

	public void stop(BundleContext context) throws Exception {
		 testThread.interrupt();
		 testSensorTracker.close();
	}
}
```

Listado 2.2-2 - Activator

Una vez que tenemos el código compilado, volveremos a ejecutar ambos bundles sobre la plataforma OSGI.

Para probar nuestro ejemplo, solo tenéis que :

- Parar el bundle que contiene el servicio: La plataforma invocará al TestSensorTracker y se detendrá el bundle consumidor.

- Parar el bundle consumidor: El método stop del Activator detendrá el Thread (*testThread.interrupt()*) e invocará al TestSensorTracker (*testSensorTracker.close()*) que eliminará la referencia al servicio (*super.removedService(reference, service)*).

2.3 Service Factory

El registrar un servicio utilizando la interfaz ServiceFactory, nos proporciona la posibilidad de registrar una instancia del servicio para cada bundle que lo consuma.

Para intentar ver de modo práctico el uso del ServiceFactory, primeramente veremos como si no utilizamos esta interfaz, la instancia del servicio será única para todos los bundles que lo consuman.

Tomaremos de punto de partida el ejemplo de los eventos (Apartado 2.1.1), para esta

primera prueba lo único que modificaremos es la clase que implementa el servicio org.javahispano.sensorreader, añadiéndole a la clase una variable global que nos indique el valor anterior a la nueva lectura:

Implementación del Servicio:

```
package org.javahispano.sensorreader;

public class SensorTemperaturaImpl implements SensorTemperatura{
        private double valorAnterior = 0;

        public double getTemp(){
                System.out.println(":::VALOR ANTERIOR::: " +  valorAnterior);
                double valor = ((double) Math.random() * 100.5);
                valorAnterior = valor;
                return  valor;
        }
}
```

Listado 2.3-1 - Implementación del Servicio SensorTemperaturaImpl

Cada vez que llamemos al método getTemp(), nos mostrará el valor retornado en la llamada anterior (En la primera invocación el valor será 0).

La salida que podríamos obtener al ejecutar este ejemplo sería:

```
:::VALOR ANTERIOR::: 0.0
Temp: 52.87979155742252
:::VALOR ANTERIOR::: 52.87979155742252
Temp: 58.95909051040939
:::VALOR ANTERIOR::: 58.95909051040939
Temp: 93.85129930571783
```

¿Que ocurriría si otro bundle consume también este servicio? Pues que la variable global "valor anterior" estaría compartida por ambos bundles.

Para comprobar esto lo único que he hecho es duplicar el bundle *org.javahispano.sensortest*, llamándole *org.javahispano.sensortest2*. De este nuevo bundle he simplificado la forma de capturar el servicio, quitando la gestión de los eventos, el activator quedaría así:

```
package org.javahispano.sensortest;

public class Activator implements BundleActivator{
private SensorTemperatura sensorTemperatura;
private Thread testThread = null;
```

```
ServiceReference ref;
        public void start(BundleContext context) throws Exception {
      ref = context.getServiceReference(SensorTemperatura.class.getName());
           sensorTemperatura = (SensorTemperatura)context.getService(ref);
           testThread = new Thread( new TestSensorThread(sensorTemperatura));
           testThread.start();
        }
        public void stop(BundleContext context) throws Exception {
                 if(testThread != null)
                          testThread.interrupt();
                context.ungetService(ref);
                ref = null;
        }
}
```

Listado 2.3-2 - Clase Activator – org.javahispano.sensortest2

De la clase thread de este nuevo bundle solamente cambio la traza:

System.out.println("Temp2: " + sensorTemperatura.getTemp());

La idea es desplegar este nuevo bundle en la plataforma, pero sin arrancarlo, es decir solo dejamos arrancados los dos anteriores. La salida que veríamos sería igual a la anterior:

```
:::VALOR ANTERIOR::: 0.0
Temp: 52.87979155742252
:::VALOR ANTERIOR::: 52.87979155742252
Temp: 58.95909051040939
:::VALOR ANTERIOR::: 58.95909051040939
Temp: 93.85129930571783

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.javahispano.sensortest_1.0.0
2       ACTIVE      org.javahispano.sensorreader_1.0.0
3       RESOLVED    org.javahispano.sensortest2_1.0.0

osgi>
```

Si ahora arrancamos el nuevo bundle, tendremos que ver que según arranca coge el "valor anterior" que le ha dado el primer bundle:

```
:::VALOR ANTERIOR::: 0.0
Temp: 52.87979155742252
:::VALOR ANTERIOR::: 52.87979155742252
Temp: 58.95909051040939
:::VALOR ANTERIOR::: 58.95909051040939
Temp: 93.85129930571783

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.javahispano.sensortest_1.0.0
2       ACTIVE      org.javahispano.sensorreader_1.0.0
3       RESOLVED    org.javahispano.sensortest2_1.0.0

osgi> start 3
:::VALOR ANTERIOR::: 93.85129930571783
Temp2: 70.08322764090255

:::VALOR ANTERIOR::: 70.08322764090255
Temp: 77.22543075701803

:::VALOR ANTERIOR::: 77.22543075701803
Temp2: 58.410518560918476

:::VALOR ANTERIOR::: 58.410518560918476
Temp: 65.04992703195114

:::VALOR ANTERIOR::: 65.04992703195114
Temp2: 72.50096517242912
```

Si lo que queremos es que cada consumidor tenga su propia instancia del servicio, crearemos una nueva clase en el bundle del servicio *org.javahispano.sensorreader* que implementará a *ServiceFactory*.

SensorTemperaturaFactory:

```
package org.javahispano.sensorreader;

import org.osgi.framework.Bundle;
import org.osgi.framework.ServiceFactory;
import org.osgi.framework.ServiceRegistration;

public class SensorTemperaturaFactory implements ServiceFactory{
   SensorTemperatura sensorTemperaturaService = null;

   public Object getService(Bundle bundle, ServiceRegistration registration) {
      sensorTemperaturaService = new SensorTemperaturaImpl();
      return sensorTemperaturaService;
   }
   public void ungetService(Bundle bundle, ServiceRegistration registration,
Object service) {

   }
}
```

Listado 2.3-3 - Clase SensorTemperaturaFactory

Implementar la interfaz ServiceFactory nos obliga a sobrescribir dos métodos: getService y ungetService.

Cada vez que un bundle consumidor haga get o unget de nuestro servicio de temperatura, internamente esta clase será invocada.

Finalmente para que este ServiceFactory entre en acción tendremos que retocar el Activator que registra el servicio:

```
package org.javahispano.sensorreader;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class Activator implements BundleActivator {
        private ServiceRegistration registration;

        public void start(BundleContext context) throws Exception {
                //ANTES
```

```
            //registration =
context.registerService(SensorTemperatura.class.getName(),
                        new SensorTemperaturaImpl(), null);
            //AHORA
            SensorTemperaturaFactory sensorTemperaturaServiceFactory =
new SensorTemperaturaFactory();
            registration =
context.registerService(SensorTemperatura.class.getName(),
                        sensorTemperaturaServiceFactory, null);
        }
        public void stop(BundleContext context) throws Exception {
            registration.unregister();
        }
}
```

Listado 2.3-4 - Clase Activator – Ejemplo ServiceFactory

Si ahora hacemos la prueba anterior veremos que al arrancar el segundo bundle de test, su valor anterior empieza por 0 y no por el último valor registrado por el primer bundle.

```
:::VALOR ANTERIOR::: 0.0
Temp: 52.87979155742252
:::VALOR ANTERIOR::: 52.87979155742252
Temp: 93.85129930571783
osgi> ss
Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.javahispano.sensortest_1.0.0
2       ACTIVE      org.javahispano.sensorreader_1.0.0
3       RESOLVED    org.javahispano.sensortest2_1.0.0

osgi> start 3
:::VALOR ANTERIOR::: 0.0
Temp2: 70.08322764090255

:::VALOR ANTERIOR::: 93.85129930571783
Temp: 77.22543075701803

:::VALOR ANTERIOR::: 70.08322764090255
Temp2: 65.04992703195114

:::VALOR ANTERIOR::: 77.22543075701803
Temp: 58.410518560918476
```

2.4 Servicios Declarativos

Usando la especificación *Declarative Services* de OSGI, podremos registrar servicios configurando de forma declarativa las descripción de los mismos en un XML.
Estos servicios declarados mediante mecanismos declarativos, forman el modelo de componentes de servicios. El "*Service Component model*" usa un modelo declarativo para publicar, buscar y conectar servicios OSGI. Este modelo simplifica la programacion de servicios y la gestión de sus dependencias. Además de reducir la cantidad de código a programar también permite que los componentes de servicio sean cargados solo cuando sean necesarios. Este tipo de bundles/componentes no necesitan de una clase Activadora.
Básicamente un bundle necesita de cuatro cosas para llegar a ser un componente:

- Un fichero XML donde describir al servicio y las dependencias de este con otros servicios.
- Modificar el fichero manifest añadiéndole una nueva cabecera que indicará que este paquete se comportará como un componente.
- Implementar la clase donde se definan los métodos *Activate* o *Desactivate* (O los métodos *bind* y *unbind*)
- Añadir a la plataforma la implementación correspondiente de la especificación Declaratives Services, bien sea por ejemplo la de Equinox o Apache Felix SCR.

2.4.1 Construyendo un servicio de forma declarativa

Para ver de forma práctica los principios básicos de la especificación Declarative Services, realizaremos el mismo ejemplo que en los apartados anteriores: registraremos el servicio SensorTemperatura y desde otro bundle lo consumiremos.
Lo primero que nos indica que este bundle se trata de un componente, es su fichero XML donde se describen los servicios y dependencias. Nuestro fichero XML será el siguiente:

```
<scr:component name="sensorreader.component"
   immediate="true"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.osgi.org/xmlns/scr/v1.0.0
   http://www.osgi.org/xmlns/scr/v1.0.0/scr.xsd ">
   <implementation
class="org.javahispano.sensorreader.SensorTemperaturaImpl"/>

   <service>
       <provide interface="org.javahispano.sensorreader.SensorTemperatura" />
   </service>
</scr:component>
```

Listado 2.4.1-1 - XML Servicio Declarativo

En el fichero anterior hemos declarado un servicio bajo la interfaz org.javahispano.sensorreader.SensorTemperatura. La clase que implementará esta interfaz será: org.javahispano.sensorreader.SensorTemperaturaImpl.

Nuestra interfaz será idéntica a los ejemplos de los apartados anteriores:

```
package org.javahispano.sensorreader;

public interface SensorTemperatura {
   public double getTemp();
}
```

Listado 2.4.1-2 - Interfaz SensorTemperatura

La clase que implementa a esta interfaz, de momento, también será idéntica a la de los apartados anteriores:

```
package org.javahispano.sensorreader;

import org.osgi.service.component.ComponentContext;

public class SensorTemperaturaImpl implements SensorTemperatura{
   public double getTemp(){
         return  ((double) Math.random() * 100.5);
   }
}
```

Listado 2.4.1-3 - SensorTemperaturaImpl

Por último modificaremos el manifest.mf para indicarle al framework donde se encuentra ubicado el fichero XML que describe el componente:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: org.javahispano.sensorreader Plug-in
Service-Component: OSGI-INF/component.xml
Bundle-SymbolicName: org.javahispano.sensorreader
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.service.component
Export-Package: org.javahispano.sensorreader
```

Listado 2.4.1-4 - Manifest.mf etiqueta Service-Component

Por lo tanto la viéndolo desde el package explorer de Eclipse, nuestro proyecto tendría el siguiente aspecto:

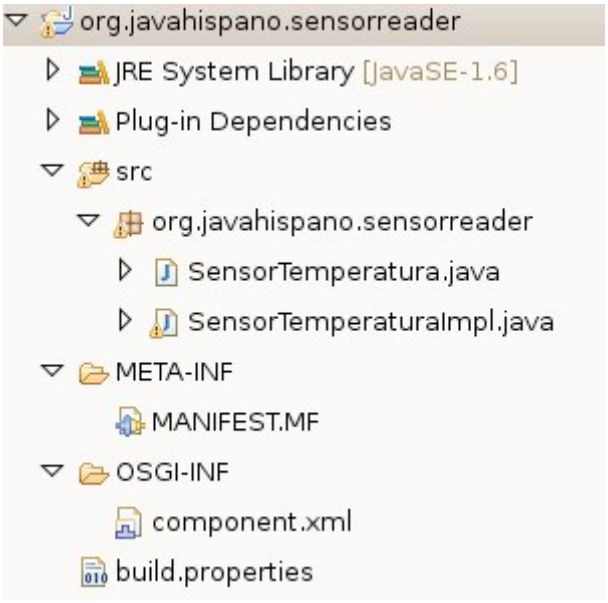


Figura 2.1 - Vista del proyecto desde eclipse

Ya tendríamos nuestro servicio SensorTemperatura, dispuesto para ser desplegado en la plataforma. El servicio será registrado en la plataforma automáticamente cuando sea necesario.
Nótese en el bundle construido, la ausencia de una clase activadora. Aunque no tengamos clase Activadora, podremos controlar el ciclo de vida de nuestro servicio declarando dos métodos en la clase que implementa la interfaz:

- ***protected void activate(ComponentContext context)***

  Este método será invocado por el framework cuando se active el servicio.

- ***protected void deactivate(ComponentContext context)***

  Este método será invocado por el framework cuando se active el servicio.

Así pues si queremos realizar alguna acción específica cuando nuestro servicio sea activado o desactivado, solo tendremos que modificar la clase que implementa la interfaz, quedando algo así:

```
package org.javahispano.sensorreader;

import org.osgi.service.component.ComponentContext;

public class SensorTemperaturaImpl implements SensorTemperatura{

   protected void activate(ComponentContext context ){
         System.out.println("Activated component:" + getClass().getName());
    }
    protected void deactivate(ComponentContext context) {
         System.out.println("Deactivated component:" + getClass().getName());
        }

    public double getTemp(){
         return  ((double) Math.random() * 100.5);
    }
}
```

Listado 2.4.1-5 - SensorTemperaturaImpl – Métodos activate y desactivate

2.4.2 Consumiendo un servicio de forma declarativa

Ahora es ocasión de consumir el servicio anterior, también de forma declarativa.
Al igual que en el caso anterior necesitamos un fichero XML descriptor del componente.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="sensortemperatura.consumer.component"
   immediate="true"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.osgi.org/xmlns/scr/v1.0.0
http://www.osgi.org/xmlns/scr/v1.0.0/scr.xsd ">
   <implementation class="org.javahispano.sensortest.impl.ConsumerImpl"/>

   <service>
         <provide interface="org.javahispano.sensortest.service.Consumer" />
   </service>

         <reference name="SensorTemp"
         interface="org.javahispano.sensorreader.SensorTemperatura"
         cardinality="1..1"
         policy="static"/>

</scr:component>
```

Listado 2.4.2-1 - XML Componente consumidor

Para evitarnos escribir una clase Activator, hemos creado un componente Consumer que hará uso del servicio SensorTemperatura construido en el apartado anterior. Para declarar el componente Consumer hemos utilizado:

```
<implementation class="org.javahispano.sensortest.impl.ConsumerImpl"/>
<service> <provide interface="org.javahispano.sensortest.service.Consumer" /> </service>
```

Para que nuestro componente Consumer pueda capturar el servicio SensorTemperatura, hemos declarado la referencia a dicho servicio a través de:

```
<reference name="SensorTemp"
        interface="org.javahispano.sensorreader.SensorTemperatura"
        cardinality="1..1"
        policy="static"/>
```

Dentro de la etiqueta reference encontramos los siguientes atributos:

- **name:** Nombre que se le quiera dar a la referencia del servicio que se desea capturar.
- **interface**: Nombre de la interface sobre la que esta registrado el servicio a capturar.
- **cardinality:** Este atributo controla cuando la dependencia es opcional y obligatoria o cuando es simple o múltiple. Las opciones para este atributo son:
    - 0..1: Opcional y singular, "cero o uno"
    - 1..1: Obligatorio y singular, "exactamente a uno"
    - 0..n: Opcional y múltiple, "de cero a muchos"
    - 1..n: Obligatorio y múltiple, "de uno a muchos" ó "al menos uno"
- **policy:** El valor para este atributo puede ser "static" o "dynamic". Fijará la política para manejar las referencias a servicios. Para servicios que pueden estar continuamente siendo registrados o eliminados o actualizados conviene utilizar políticas dinámicas. Por defecto este atributo se fija a estático. Sobre esto veremos un ejemplillo, pero donde mejor podéis consultarlo es la documentación de la especificación Declarative Services (112.3.3 Reference Policy).

Después de tener claro el XML descriptor del componente, crearemos la interfaz Consumer y la implementación de dicha interfaz ConsumerImpl.

```
package org.javahispano.sensortest.service;

public interface Consumer {

}
```

Listado 2.4.2-2 - Interfaz Consumer

Para la implementación de la interfaz usaremos la clase TestSensorThread de los ejemplos anteriores (Listado 2.1.2-2), para realizar una lectura periódica del valor del sensor:

```
package org.javahispano.sensortest.impl;


import org.javahispano.sensorreader.SensorTemperatura;

import org.javahispano.sensortest.service.Consumer;
import org.osgi.service.component.ComponentContext;


public class ConsumerImpl implements Consumer{

   private SensorTemperatura      sensorTemperatura    = null;
   private Thread testThread = null;

  protected void activate(ComponentContext context ){
    System.out.println("Activated component:" + getClass().getName());
    sensorTemperatura = (SensorTemperatura) context.locateService("SensorTemp");
    testThread = new Thread( new TestSensorThread(sensorTemperatura));
    testThread.start();
   }

   protected void deactivate(ComponentContext context) {
         System.out.println("Desactivated component:" + getClass().getName());
         testThread.interrupt();
  }
}
```

Listado 2.4.2-3 - Implementación Consumidor

En el código anterior cuando se activa el servicio, se captura la referencia al servicio del sensor de temperatura:

sensorTemperatura = (SensorTemperatura) context.locateService("SensorTemp");

Como parámetro al método *context.locateService* le pasaremos el nombre que le hemos dado en el XML a la referencia del servicio que queremos capturar.
Si queremos evitar construir los métodos activate y desactivate, podremos utilizar los métodos bind y unbind que serán invocados cada vez que se conecten o desconecten los servicios. Para ello modificaremos el XML:

```
<reference name="SensorTemp"
         interface="org.javahispano.sensorreader.SensorTemperatura"
         cardinality="1..1"
         policy="dynamic"
         bind="setService"
        unbind="unsetService"/>
```

En la implementación de la interfaz realizaremos la misma acción que realizábamos con los métodos activate y desactivate, pero en esta ocasión sobre los métodos setService y unsetService:

```
package org.javahispano.sensortest.impl;

import org.javahispano.sensorreader.SensorTemperatura;
import org.javahispano.sensortest.service.Consumer;

public class ConsumerImpl implements Consumer{

   private SensorTemperatura      sensorTemperatura    = null;
   private Thread testThread = null;

   public void setService(SensorTemperatura sensortemp){
         System.out.println("Activated component:" + getClass().getName());
         if(testThread == null){
                  sensorTemperatura = sensortemp;
                  testThread = new Thread( new
TestSensorThread(sensorTemperatura));
                  testThread.start();
         }
   }

   public void unsetService(SensorTemperatura sensortemp){
         testThread.interrupt();
         testThread = null;
   }
}
```

Listado 2.4.2-4 - Implementación Consumidor métodos bind y unbind

2.4.3 Ejecutando el ejemplo

Para poder ejecutar el ejemplo de los apartados 2.4.1 y 2.4.2, tendremos que desplegar en la plataforma los bundles:

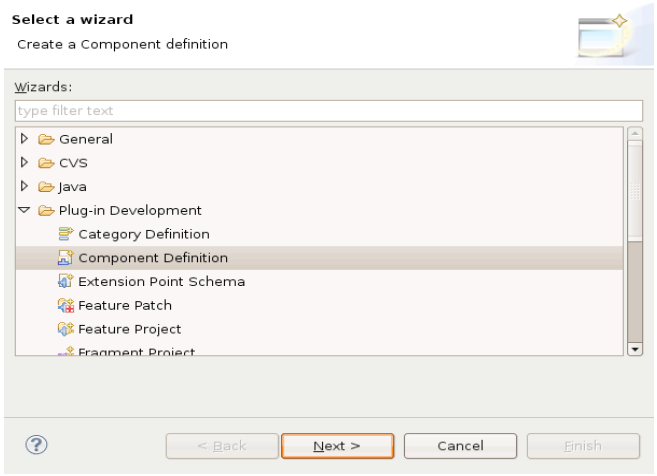
- org.eclipse.osgi.services
- org.eclipse.equinox.ds
- org.eclipse.equinox.util

En vez de usar el bundle org.eclipse.equinox.ds, podríamos utilizar también la implementación de los servicios declarativos de Apache Felix SCR.

Una vez desplegados correctamente los bundles necesarios, la salida en tiempo de ejecución de nuestros bundles, debería ser similar a la de los apartados anteriores al punto 2.1, 2.2 y 2.3.

2.4.4 Acelerando el desarrollo de componentes con Eclipse

Como ya sabemos eclipse incorpora una serie de utilidades para agilizar el desarrollo de bundles , tanto para OSGI "estándar" como para el propio Eclipse .
Una de la utilidades que nos presenta eclipse para desarrollar servicios declarativos es el Wizard de Creación de Componentes y el editor "Component Definition Editor".
Para crear un nuevo componente con Eclipse, pincharemos con el botón derecho sobre el Proyecto y seleccionaremos New --> Other. Del árbol de posibilidades desplegamos Plugin Development y seleccionamos "Component Definition":



Listado 2.4.4-1 - Wizard Eclipse Componentes

Pinchando sobre "Next" apareceremos en la siguiente pantalla:

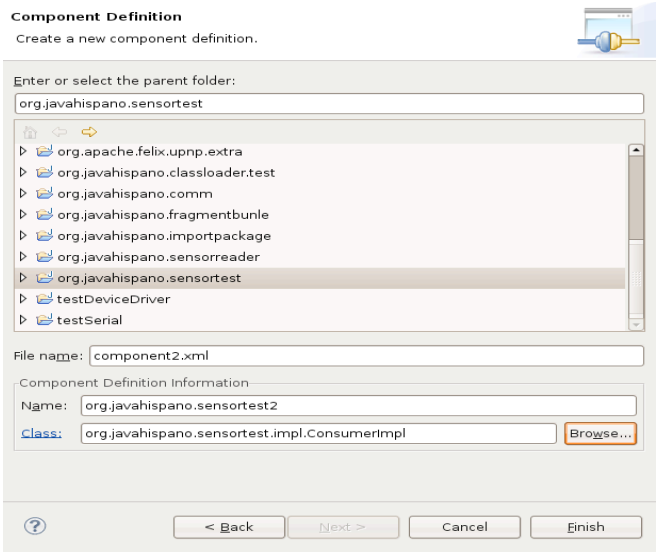


Figura 2.4.4-2 - Wizard Eclipse Componentes II

En la pantalla anterior daremos un nombre al fichero XML. también daremos un nombre al componente y le indicaremos la clase que implementará al componente, en el caso del sensortest será la clase ConsumerImpl.
Si pinchamos en Finish, eclipse creará el fichero XML y modificará el manifest añadir a este nuevo componente:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="org.javahispano.sensortest2">
   <implementation
class="org.javahispano.sensortest.impl.ConsumerImpl"/>
</scr:component>
```

Listado 2.4.4-1 - XML Componente – Wizard Eclipse

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: sensortest Plug-in
Bundle-SymbolicName: org.javahispano.sensortest
Service-Component: OSGI-INF/component2.xml
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.javahispano.sensorreader,
 org.osgi.service.component
```

Listado 2.4.4-2 - manifest – Wizard Eclipse

Si hacemos doble click sobre el fichero component2.xml por defecto se abrirá con el editor de componentes:

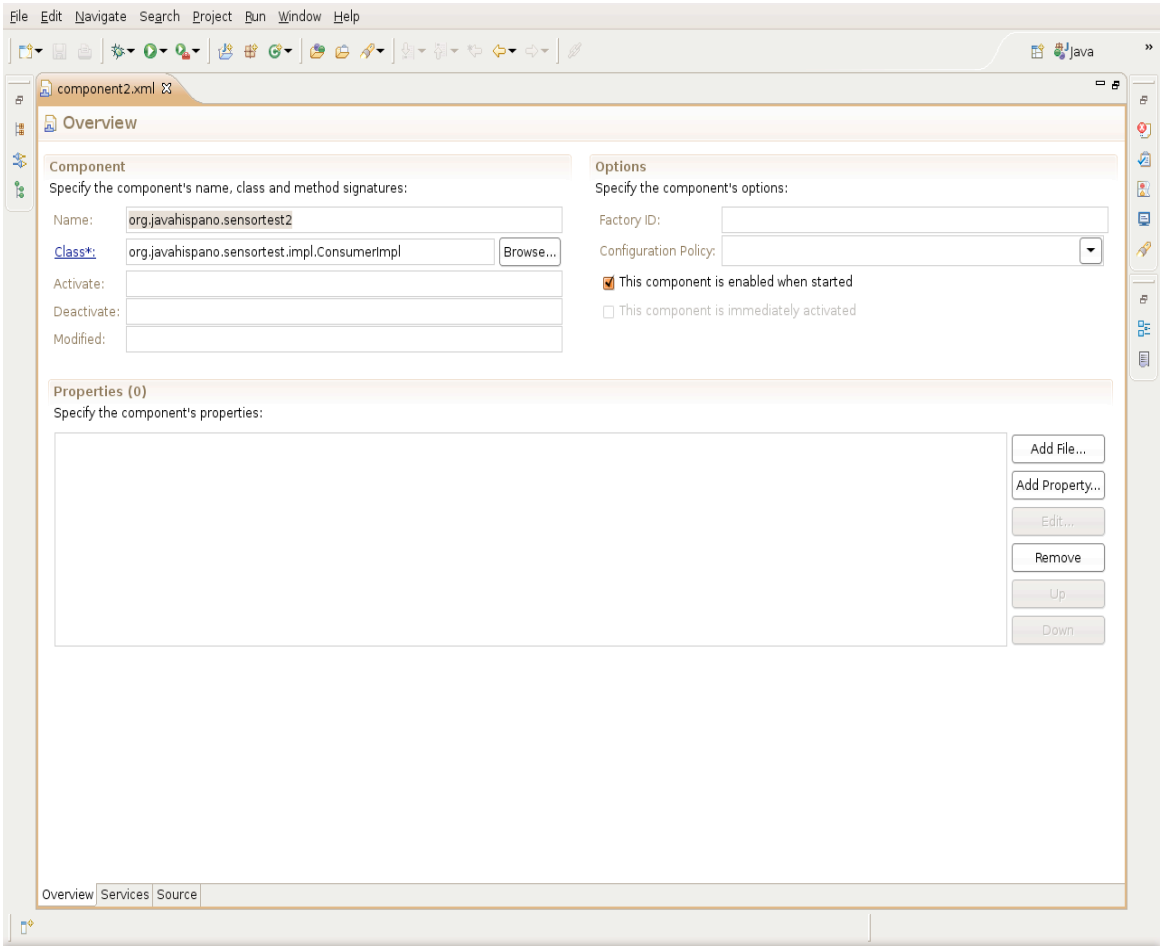


Figura 2.4.4-3 - Editor de XML Componentes Eclipse

Desde el editor de componentes podremos declarar nuestros servicios o referencias a ellos, sin tener que escribir ningún tipo de código XML.

2.5 Extensión de la línea de comandos de la consola

Equinox nos permite extender la consola de comandos para crear comandos personalizados. Extender la consola de comandos de Equinox es muy sencillo, nosotros la vamos a utilizar en este ejemplo para crear dos comandos:

- **activaservicio:** Con este comando se activará el servicio del sensor de temperatura.
- **temperatura**: Con este comando, si esta activo el servicio, obtendremos el valor de la temperatura.

Ampliar la consola de comandos de Equinox es tan fácil como implementar la interfaz: org.eclipse.osgi.framework.console.CommandProvider y registrarla como un servicio OSGI.
Implementar la interfaz CommandProvider, solo nos obliga a implementar el método *public String getHelp()*. Este método retornará un String con el texto referente a la ayuda para los

comandos que se van a crear. Para definir un comando escribiremos un método que corresponda con el siguiente patrón:

public void _nombredelcomando (CommandInterpreter ci) throws Exception

El nombre del comando irá precedido de un "_" aunque en la consola escribiremos el comando sin guión.
Para construir nuestro ejemplo hemos utilizado una única clase que implementará la interfaz Activator y CommandProvider.

```
package org.javahispano.sensorreader;

import java.util.Hashtable;

import org.eclipse.osgi.framework.console.CommandInterpreter;
import org.eclipse.osgi.framework.console.CommandProvider;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class Activator implements BundleActivator,CommandProvider {
   private ServiceRegistration registrationTemp;
   private ServiceRegistration registrationCommand;
   private  BundleContext context;
   private SensorTemperatura sensorTemperatura = null;

   public void start(BundleContext context) throws Exception {
         this.context = context;
         sensorTemperatura = new SensorTemperaturaImpl();
         Hashtable properties = new Hashtable();
         registrationCommand =
context.registerService(CommandProvider.class.getName(), this, properties);
   }
   public void stop(BundleContext context) throws Exception {
         registrationCommand.unregister();
         if(registrationTemp != null){
                  registrationTemp.unregister();
         }
   }
   public String getHelp() {
         StringBuffer buffer = new StringBuffer();
         buffer.append("\t ROBERTO - returns framework information\n");
         return buffer.toString();
   }
```

```
    public void _activaservicio(CommandInterpreter ci) throws Exception {
        if(registrationTemp == null){
            registrationTemp =
                context.registerService(SensorTemperatura.class.getName(),
                                        sensorTemperatura, null);
            System.out.println("Servicio Temperatura Activado");
        }else{
            System.out.println("El Servicio Temperatura ya esta activo");
        }
    }
    public void _temperatura(CommandInterpreter ci) throws Exception {
        System.out.println(sensorTemperatura.getTemp());
    }
}
```

Listado 2.5-1 - CommandProvider

En el método start de la clase anterior creamos una instancia del SensorTemperatura, pero no lo registraremos hasta que no se ejecute el comando activaservicio. En este método start registraremos la propia clase Activator como un servicio que implementa la interfaz CommandProvider.
Este ejemplo lo desplegaremos junto con el consumidor del servicio construido de forma declarativa en el apartado anterior. Al arrancar la plataforma veremos como están los dos bundles activos (org.javahispano.sensorreader y org.javahispano.sensortest) pero no ocurre nada:

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.knopflerfish.bundle.rxtxcomm-API_2.1.7
            Fragments=2
2       RESOLVED    org.knopflerfish.bundle.rxtxcomm-linux-pc-LIB_2.1.7
            Master=1
3       RESOLVED    org.eclipse.osgi.services_3.1.200.v20071203
4       ACTIVE      org.eclipse.equinox.ds_1.0.0.v20080427-0830
5       RESOLVED    org.eclipse.equinox.util_1.0.0.v20080414
6       ACTIVE      org.javahispano.sensorreader_1.0.0
7       ACTIVE      org.javahispano.sensortest_1.0.0

osgi>
```

A continuación si escribimos el comando “activaservicio”, el servicio SensorTemperatura será registrado y el consumidor org.javahispano.sensortest, comenzará a usarlo:

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.knopflerfish.bundle.rxtxcomm-API_2.1.7
                Fragments=2
2       RESOLVED  org.knopflerfish.bundle.rxtxcomm-linux-pc-LIB_2.1.7
                Master=1
3       RESOLVED  org.eclipse.osgi.services_3.1.200.v20071203
4       ACTIVE      org.eclipse.equinox.ds_1.0.0.v20080427-0830
5       RESOLVED  org.eclipse.equinox.util_1.0.0.v20080414
6       ACTIVE      org.javahispano.sensorreader_1.0.0
7       ACTIVE      org.javahispano.sensortest_1.0.0

osgi> activaservicio
SETSERVICE
Activated component:org.javahispano.sensortest.impl.ConsumerImpl
Servicio Temperatura Activado

osgi> Temp: 76.03212667628212
Temp: 65.11295029390902
```

También podremos ejecutar el comando “temperatura”:

```
osgi> temperatura
13.757790893866133
```

3 Caso Práctico: Control Remoto de una Calefacción

Ahora que tenemos claro los principios básicos de OSGI y hemos aprendido a manejar sus servicios, es hora de enganchar la plataforma al sensor de temperatura.
Este sensor de temperatura, estará controlado por un microcontrolador PIC que enviará los datos referentes a la temperatura a través de un puerto serie RS232.
No entraremos en detalle acerca del código ensamblador que lleva programado el PIC, pero dicho código es adjuntado a este artículo, así como la imagen del circuito que lo rodea.

3.1 Accediendo al puerto serie a través de librerías nativas

Para controlar el puerto serie de nuestro PC o plataforma, necesitamos acceder a una serie de librerías nativas. El acceso a este tipo de librerías se realiza a través JNI. Por suerte no tendremos que construir nosotros mismos estos accesos, sino que existen diferentes APIS java para esta labor. Las más conocidas son:

- javax.comm de SUN
- rxtxserial de rxtx.org

No entraremos en detalles acerca de JNI, ni tan siquiera como manejar el puerto serie, en este tutorial usaremos los métodos más básicos. Si deseáis obtener más información acerca de JNI o de las APIs de control del puerto serie:

- http://rxtx.org/
- http://java.sun.com/products/javacomm/index.jsp
- http://es.wikipedia.org/wiki/Java_Native_Interface
- http://java.sun.com/docs/books/jni/

La mejor idea para trabajar con librerías nativas dependientes del entorno, es usar "fragment bundles". Como ya mencionamos en la anterior entrega, un Fragment Bundle es un trozo de bundle que es anexado a otro. Esto resulta interesante cuando trabajamos con este tipo de librerías, ya que podemos agrupar las librerías dependientes del entorno en diferentes fragmentos de bundle. En nuestro caso construiremos un bundle principal que contendrá el API de rxtxserial y dos fragment bundles con las librerías dependientes del entorno, uno para una arquitectura linux arm y otro para una arquitectura linux x86.

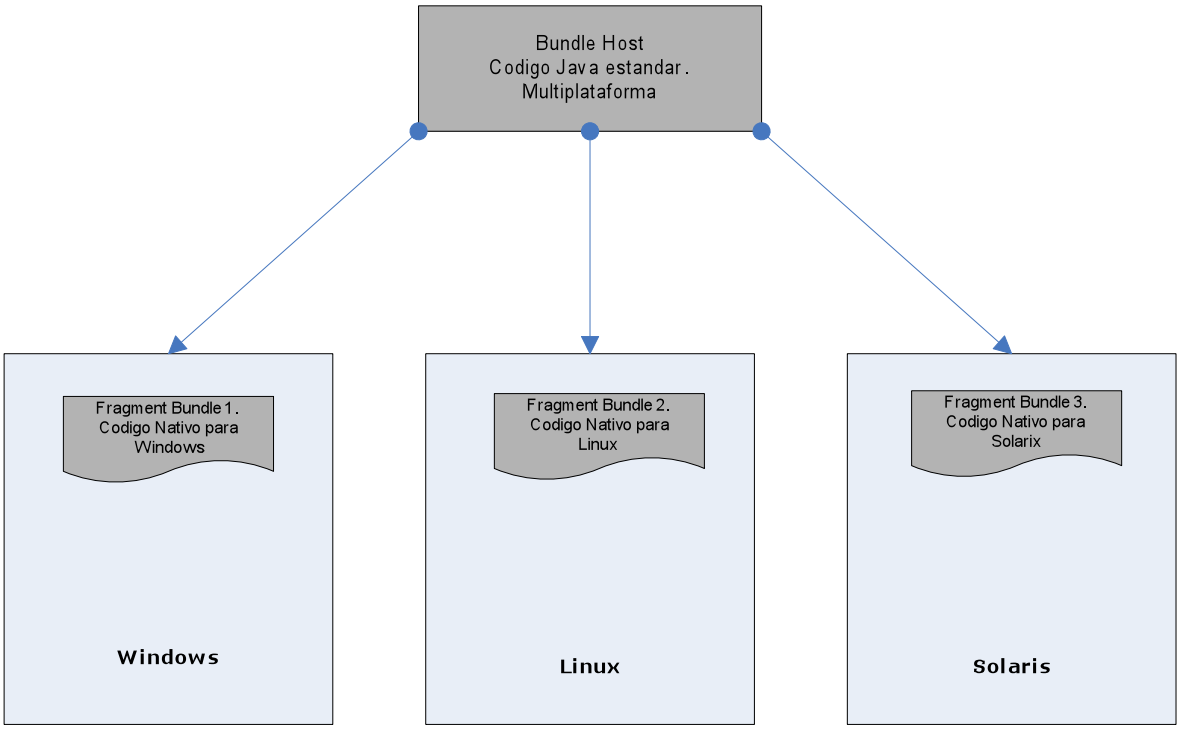


Figura 3.1-1 - FragmentBundles

Como hoy en día esta ya esta todo inventado, nos serviremos de los bundles que ya tienen preparados en el repositorio knopflerfish:

http://www.knopflerfish.org/releases/current/docs/jars/index.html

Concretamente usaremos dos bundles:

- rxtxcomm-linux-arm-2.1.7: Fragment bundle que contiene la librerías nativas para un sistema linux 2.6 corriendo sobre un sistema arm.
- rxtxcomm_api-2.1.7: Librería java RXTX, contiene la parte independiente del sistema operativo y del tipo de procesador. Necesita del fragment bundle con las librerías nativas.

Desde la página de knopflerfish nos bajaremos estos dos bundles, el API RXTX y las librerías nativas para procesadores ARM. Supongo que no todo el mundo tiene un sistema ARM a mano, así que vamos a construirnos un nuevo fragment bundle para la arquitectura de nuestro PC, y ya de paso aprovechamos y "destripamos" los dos bundles que nos hemos descargado.

Lo primero que haremos es descargarnos la librería libSerial.so (para linux), podemos buscar en Google algún proyecto que ya la tenga y descárgala o bajarnos los fuentes y compilarlos.

El siguiente paso será descomprimir el bundle rxtxcomm-linux-arm-2.1.7.jar, para duplicarlo con otro nombre. Sustituiremos la librería libSerial.so por la correspondiente a la

arquitectura de nuestro PC para seguidamente editar el MANIFEST.MF, utilizando lo explicado en la anterior entrega acerca de la etiqueta Bundle-NativeCode.

Manifest de rxtxcomm-linux-arm-2.1.7
Manifest-Version: 1.0 **Ant-Version:** Apache Ant 1.7.1 **Created-By:** 16.3-b01-279 (Apple Inc.) **Bundle-ManifestVersion:** 2 **Bundle-Name:** RXTXcomm-linux-arm-LIB **Bundle-SymbolicName:** org.knopflerfish.bundle.rxtxcomm-linux-arm-LIB **Bundle-Version:** 2.1.7 **Bundle-Category:** service **Bundle-Description:** RXTX comm native driver for Linux/armle (LIB) **Bundle-Vendor:** Piayda/RXTX **Bundle-DocURL:** https://www.knopflerfish.org/svn/knopflerfish.org/trunk /osgi/bundles_opt/serial/readme.txt **Bundle-ContactAddress:** http://www.knopflerfish.org ***Bundle-NativeCode:*** *librxtxI2C.so; librxtxParallel.so; librxtxRS485.so; librxtxRaw.so; librxtxSerial.so ;osname = Linux ;processor = armle; p rocessor = armv4tl* **Fragment-Host:** org.knopflerfish.bundle.rxtxcomm-API **Knopflerfish-Version:** 3.0.0 **Bundle-Classpath:** . **Bundle-SubversionURL:** https://www.knopflerfish.org/svn/ **Bundle-UUID:** org.knopflerfish:rxtxcomm-linux-arm:2.1.7:lib **Build-Date:** Mon June 21 2010, 18:00:32 **Built-From:** /Users/jan/work/3.0.0/osgi/bundles_opt/serial/rxtxcomm-lin ux-arm

Listado 3.1-1 - Manifest procesador ARM

Nuevo Manifest
Manifest-Version: 1.0 **Ant-Version:** Apache Ant 1.7.1 **Created-By:** 16.3-b01-279 (Apple Inc.) **Bundle-ManifestVersion:** 2 **Bundle-Name:** RXTXcomm-linux-PC-LIB **Bundle-SymbolicName:** org.knopflerfish.bundle.rxtxcomm-linux-pc-LIB **Bundle-Version:** 2.1.7 **Bundle-Category:** service **Bundle-Description:** RXTX comm native driver for Linux/x86 (LIB) **Bundle-Vendor:** Piayda/RXTX **Bundle-DocURL:** https://www.knopflerfish.org/svn/knopflerfish.org/trunk

```
 /osgi/bundles_opt/serial/readme.txt
Bundle-ContactAddress: http://www.knopflerfish.org
Bundle-NativeCode:  librxtxSerial.so ;osname = Linux ;processor = x86
Fragment-Host: org.knopflerfish.bundle.rxtxcomm-API
Knopflerfish-Version: 3.0.0
Bundle-Classpath: .
Bundle-SubversionURL: https://www.knopflerfish.org/svn/
Bundle-UUID: org.knopflerfish:rxtxcomm-linux-pc:2.1.7:lib
Build-Date: Mon June 21 2010, 18:00:32
Built-From: /Users/jan/work/3.0.0/osgi/bundles_opt/serial/rxtxcomm-lin
 ux-arm
```

Listado 3.1-2 - Manifest procesador X86

Una vez realizados los cambios, volvemos a comprimir el directorio como un .jar, pero con otro nombre, por ejemplo rxtxcomm-linux-pc-2.1.7.jar.

En este momento tenemos tres bundles, dos fragment bundles y uno tipo "Host" con el API java RXTX. Si echamos un vistazo al manifest del "Host Bundle", veremos que hace un export del paquete gnu.io, que será importado por el bundle que vaya a usar este API.

Ahora que tenemos correctamente desplegados nuestros bundles con el API RXTX, para acceder al puerto serie nos podemos plantear, gracias a la flexibilidad de OSGI, distintas formas de trabajar, tal y como veremos en los próximos apartados.

3.1.1 Método 1: Usando Servicios

Utilizaremos los ejemplos del apartado 2.2 como base para comenzar a escribir los siguientes ejemplos.

En el apartado 2.2 construimos un servicio que retornaba el valor del sensor de temperatura (bundle org.javahispano.sensorreader), este valor era un valor aleatorio, que ahora lo capturaremos del puerto serie.

La arquitectura de nuestro ejemplo se puede ver reflejada en la siguiente figura:

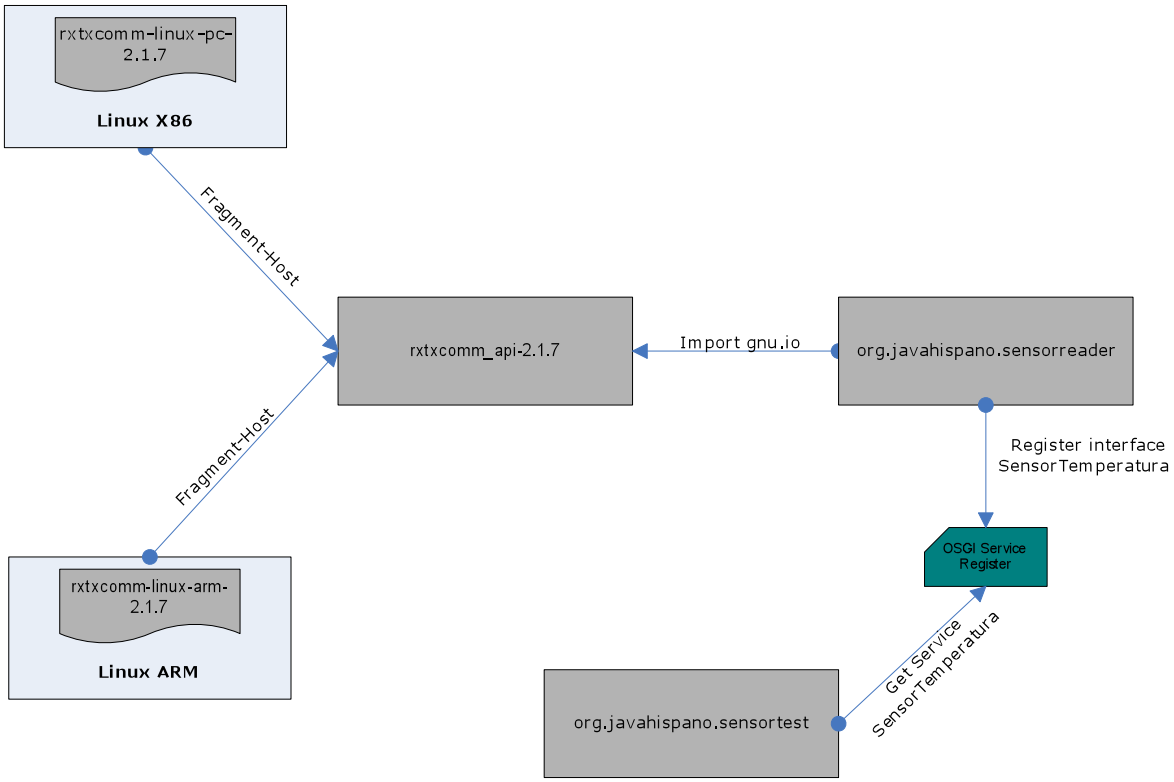


Figura 3.1.1 - 1 - Arquitectura ejemplo 3.1.1

Para capturar los datos del puerto serie modificaremos la implementación del servicio:

```
package org.javahispano.sensorreader;

import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;
import gnu.io.SerialPortEvent;
import gnu.io.SerialPortEventListener;

import java.io.IOException;
import java.io.InputStream;

import java.util.Enumeration;

public class SensorTemperaturaImpl implements SensorTemperatura,
        SerialPortEventListener {
    String defaultPort = "/dev/ttyS0";
    InputStream inputStream;

    String temperatura = "";

    public SensorTemperaturaImpl() {
```

```
        try {
                String defaultPort = "/dev/ttyS0";
                        CommPortIdentifier portId;

                        SerialPort serialPort;
                portId =    CommPortIdentifier.getPortIdentifier("/dev/ttyS0");
                        serialPort = (SerialPort) portId.open("SimpleReadApp", 2000);
                        serialPort.setSerialPortParams(9600,
SerialPort.DATABITS_8,
                                        SerialPort.STOPBITS_1,
SerialPort.PARITY_NONE);
                        inputStream = serialPort.getInputStream();
                        serialPort.addEventListener(this);
                        serialPort.notifyOnDataAvailable(true);
                        serialPort.notifyOnCarrierDetect(true);
                } catch (Exception e) {
                        e.printStackTrace();
                }
    }

    public double getTemp() {
        if (temperatura != null) {
                try{
                        return new Double(temperatura).doubleValue();
                }catch(Exception e){
                        System.out.println("Error transformando temp");
                }
        }
        return 0.0;
    }

    public void serialEvent(SerialPortEvent event) {
        switch (event.getEventType()) {
        case SerialPortEvent.DATA_AVAILABLE:
                byte[] readBuffer = new byte[20];
                try {
                        while (inputStream.available() > 0) {
                                inputStream.read(readBuffer);
                        }
                        temperatura = new String(readBuffer) + "";
                        System.out.println("Data available" + temperatura);
                } catch (IOException e) {
                        e.printStackTrace();
                }
                break;
        }
```

```
    }
}
```

Listado 3.1.1 - 1 - SensorTemperaturaImpl tomando datos del puerto COM

En código anterior vemos como en el constructor de la clase se obtiene la referencia al puerto serie (gnu.io.SerialPort) y se registra un listenner (Nuestra clase implementa también a la clase gnu.io.SerialPortEventListener). El método serialEvent será invocado cada vez lleguen datos del sensor a través del puerto serie y se encargará de actualizar la variable global de temperatura con el nuevo dato. Cuando se llame al método getTemp del servicio, este retornará el último valor guardado en la variable global. Según tenemos programada la clase SensorTemperaturaImpl, en esta ocasión no nos interesa crear un ServiceFactory (Ver apartado 2.3), ya que solo queremos que haya una única instancia del servicio. Para compilar esta clase no debemos olvidar importar el paquete gnu.io en el manifest del bundle. (Este paquete es exportado por el bundle rxtxcomm_api-2.1.7).

Ya tenemos construido debidamente el bundle con el servicio (org.javahispano.sensorreader), ahora solo nos falta probarlo con un bundle que lo consuma, para ello usaremos el bundle org.javahispano.sensortest_1.0.0.jar del apartado 2.2.

La salida en tiempo de ejecución debería ser similar a:

```
osgi>REGISTRADO SERVICIO
Añadiendo servicio SensorTemperatura

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.knopflerfish.bundle.rxtxcomm-API_2.1.7
                    Fragments=2
2       RESOLVED    org.knopflerfish.bundle.rxtxcomm-linux-pc-LIB_2.1.7
                    Master=1
3       ACTIVE      org.javahispano.sensorreader_1.0.0
4       ACTIVE      org.javahispano.sensortest_1.0.0

osgi> Temp: 35.0
Temp: 34.5
```

3.1.2 Método 2: Usando Eventos

En el apartado anterior usamos Servicios para que cualquier bundle de la plataforma pueda acceder a los datos del sensor, en esta ocasión usaremos eventos internos de la plataforma en los cuales enviaremos los valores del sensor. Cualquier otro bundle instalado podría subscribirse a estos eventos/mensajes y recibir también los valores del sensor.

Para utilizar eventos de OSGI tendremos que usar la especificación EvenAdmin (*OSGi Compendium Services*). Esta especificación nos provee de un mecanismo estándar para trabajar con eventos, un estilo similar al modelo de subscripción de en eventos JMS.

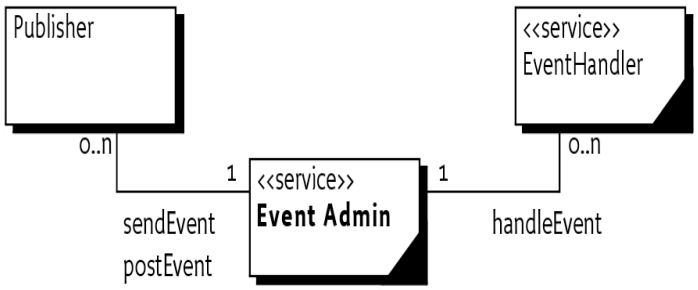


Figura 3.1.2 - 1 - Modelo de Eventos OSGI

3.1.2.1 Publicar un evento

Para publicar un evento en OSGI tendremos que:

- Adquirir una referencia al servicio EventAdmin OSGI.
- Elegir un nombre para el evento. Según la especificación, todo nombre "topic" debe seguir un estándar. (apartado 113.3.1 Topics de la especificación OSGI).
- Anexar los parámetros a publicar en el evento dentro de una clase java.util.Dictionary.
- Elegir el tipo de envío: síncrono o asíncrono. Invocaremos al método *postEvent* para los eventos sincronos y el método sendEvent para los asíncronos.

Para adquirir la referencia al servicio hemos utilizado la especificación Service Tracker que vimos en el apartado 2.2, pero esta vez implementaremos también la interfaz eventAdmin:

```
package org.javahispano.sensorreader;

import org.osgi.framework.BundleContext;
import org.osgi.service.event.Event;
import org.osgi.service.event.EventAdmin;
import org.osgi.util.tracker.ServiceTracker;

public class EventAdminTracker extends ServiceTracker implements EventAdmin{

	public EventAdminTracker(BundleContext context){
		super(context,EventAdmin.class.getName(),null);
	}
	public void postEvent(Event event){
		EventAdmin evtAdmin=(EventAdmin)getService();
		evtAdmin.postEvent(event);
	}
	public void sendEvent(Event event){
		EventAdmin evtAdmin=(EventAdmin)getService();
		evtAdmin.sendEvent(event);
	}
}
```

Listado 3.1.2.1 - 1 - EventAdminTracker

Una vez que tenemos el Tracker, lo utilizaríamos desde la clase que queramos publicar eventos:

```
EventAdminTracker evtAdmTracker=new EventAdminTracker(context);
evtAdmTracker.open();
```

El siguiente paso es crear el evento con los datos que queremos enviar:

```
Map data  = new Hashtable();
data.put("temp","10"); //Enviamos la temperatura
Event ev = new Event("org/javahispano/sensorreader/Temp",(Dictionary) data);
```

Y el último paso, enviar:

```
evtAdmTracker.sendEvent(ev);
```

ó

```
evtAdmTracker.postEvent(ev);
```

Por motivos de espacio y comodidad para el lector se ha omitido parte del código relacionado con el sensor de temperatura, pero el principio es el mismo. Si revisáis el servicio del ejemplo del apartado 3.1.1, veréis que implementábamos la interfaz SerialPortEventListener para escuchar los mensajes COM que envía el sensor de temperatura. Utilizaremos este listenner para publicar eventos OSGI con los datos de la

temperatura, para que el bundle que quiera consumirlos no tenga por que saber como se realiza la lectura de la temperatura en el sensor.

La clase SensorTemperaturaImpl ahora solo implementará a SerialPortEventListener, no tendrá método getTemp y en el método serialEvent escribiremos:

```
public void serialEvent(SerialPortEvent event) {
                switch (event.getEventType()) {
                    case SerialPortEvent.DATA_AVAILABLE:
          ....
   Map data  = new Hashtable();
   //Leemos el Buffer del puerto serie y escribimos su valor en el event
   data.put("temp",new String(readSerialBuffer));
Event ev = new Event("org/javahispano/sensorreader/Temp",(Dictionary) data);
    evtAdmTracker.sendEvent(ev);
             .....
```

Listado 3.1.2.1 - 2 - Enviar eventos

3.1.2.2 Escuchar un evento

Para escuchar eventos usando el servicio Event Admin, es necesario registrar un servicio listenner que implemente la interfaz EventHandler.

A la hora de registrar un EventHandler, hay que tener en cuenta que es necesario filtrar los eventos que va a manejar.

Para manejar eventos es necesario:

- Crear una clase que implemente la interfaz EventHandler y acceda a la propiedades o valores recibidos en el evento.
- Registrar el servicio (la clase que implemente el EventHandler) con el filtro de los "topics" que queremos recibir.

Esto se verá mejor con nuestro ejemplo del sensor de temperatura. Esta vez tomaremos de partida el bundle org.javahispano.sensortest, que es el bundle que consumirá los eventos generados por el bundle org.javahispano.sensorreader construido en el apartado anterior.

Como hemos mencionado antes es necesario registrar un objeto que implemente la interfaz EventHandler, que será el listenner de los eventos:

```
package org.javahispano.sensortest;

import org.osgi.service.event.Event;
import org.osgi.service.event.EventHandler;

public class TempEventListenner implements EventHandler{
	@Override
	public void handleEvent(Event event) {
		String strTemp = (String)event.getProperty("temp");
		System.out.println("Evento Recibido---> ");
		System.out.println(strTemp);
	}
}
```

Listado 3.1.2.2 - 1 - TempEventListenner

Implementar la interfaz la EventHandler, implica al menos sobrescribir el método handleEvent(Event event). Este método será invocado cada vez que un evento sea publicado.
Con la implementación del EventHandler construida, solo nos queda registrarla en la plataforma y aplicarle el filtro para solo recibir los eventos que nos interesan, para ello utilizaremos el Activator del bundle:

```
package org.javahispano.sensortest;

import org.osgi.framework.ServiceRegistration;
import org.osgi.service.event.EventConstants;
import org.osgi.service.event.EventHandler;

public class Activator implements BundleActivator{
private ServiceRegistration sr;

	public void start(BundleContext context) throws Exception {
		EventHandler eh = new TempEventListenner();
		String[] topics = new String[]{"org/javahispano/sensorreader/Temp"};
		Hashtable ht = new Hashtable();
		ht.put(EventConstants.EVENT_TOPIC, topics);
		sr = context.registerService(EventHandler.class.getName(),eh, ht);
	}

	public void stop(BundleContext context) throws Exception {
		sr.unregister();
	}
}
```

Listado 3.1.2.2 - 2 - Activator

3.1.2.3 Entorno de Ejecución

Antes de ejecutar nuestro ejemplo tenemos que añadir a la plataforma nuestros bundles construidos, mas la especificación OSGI de EventAdmin y la implementación correspondiente a la plataforma que estemos utilizando.

En el caso de Equinox, el config.ini debería ser similar a:

```
osgi.clean=true
eclipse.ignoreApp=true

org.osgi.framework.bootdelegation=javax.jms
osgi.bundles=../serial/plugins/rxtxcomm_api-2.1.7.jar@start, \
../serial/plugins/rxtxcomm-linux-pc-2.1.7.jar, \
org.eclipse.osgi.services_3.1.200.v20071203.jar@start,\   <----------
---------- Especificación OSGI Services
org.eclipse.equinox.event_1.1.0.v20080225.jar@start,\     <----------
---------- Implementación Equinox del EventAdmin
../serial_eventos/plugins/org.javahispano.sensorreader_1.0.0.jar@start, \
../serial_eventos/plugins/org.javahispano.sensortest_1.0.0.jar@start, \
```

Listado 3.1.2.3 - 1 - Config.ini

Cuando ejecutemos nuestro entorno la salida debería ser similar a la siguiente:

```
osgi> ss
Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      org.knopflerfish.bundle.rxtxcomm-API_2.1.7
                    Fragments=2
2       RESOLVED    org.knopflerfish.bundle.rxtxcomm-linux-pc-LIB_2.1.7
                    Master=1
3       ACTIVE      org.eclipse.osgi.services_3.1.200.v20071203
4       ACTIVE      org.eclipse.equinox.event_1.1.0.v20080225
5       ACTIVE      org.javahispano.sensorreader_1.0.0
6       ACTIVE      org.javahispano.sensortest_1.0.0

osgi> Generando  evento...
Evento Recibido--->
0034.50
Generando  evento...
Evento Recibido--->
0035.00
```

3.1.3 Método 3: Device Access

Otra forma de implementar el acceso a nuestro sensor de temperatura seria usando los servicios de la especificación Device Access.

La especificación Device Access facilita la coordinación para la detección automática de dispositivos conectados al entorno. Podemos decir que facilita la creación de los escenarios Plug and Play.

La primera vez que leí la especificación de este servicio, me resultó muy difícil de entender, ya que a mi parecer es bastante abstracta. En este apartado intentaremos mostrar los principios básicos de esta especificación, desde el punto de vista de un desarrollador de aplicaciones para OSGI, posiblemente obviemos algunos conceptos de la especificación, pero si alguien quiere adentrarse mas en ella, le invito a que recurra a los documentos oficiales.

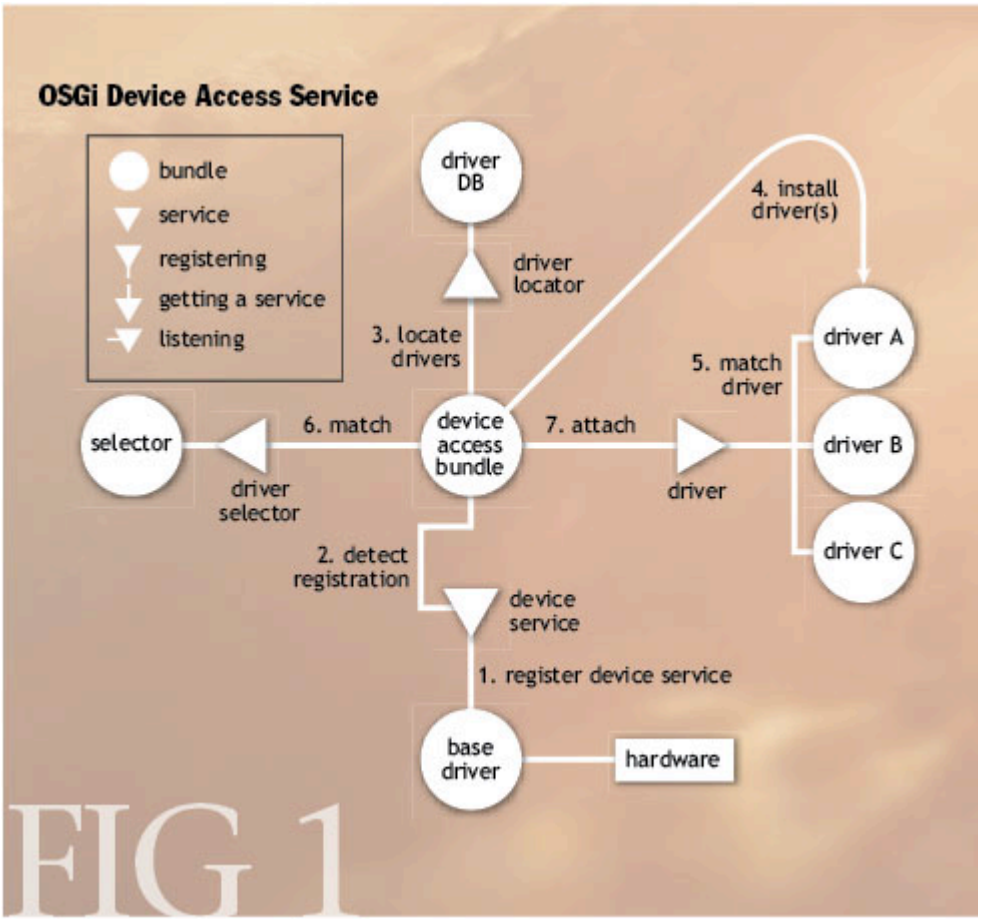


Figura 3.1.3 - 1 - Osgi Device Access Service

En la figura anterior podemos ver la foto de los componentes que forman parte de esta especificación y el orden en el que interactúan. Algunos de los componentes que aparecen en la imagen, son componentes internos de la propia especificación, aunque nosotros nos centraremos en los componentes que son necesarios de implementar por el desarrollador (Base Driver, Device Service y Driver), vamos a intentar explicar brevemente la secuencia de actuación para localizar el driver mas apropiado a un dispositivo conectado:

1. La primera acción será detectar un dispositivo conectado. Este dispositivo puede haber sido conectado físicamente, inalambricamente o virtualmente. La detección automática del dispositivo debe ser programada por el desarrollador.
2. Una vez detectado el dispositivo, el desarrollador deberá registrar un servicio OSGI bajo la interfaz org.osgi.service.device.Device. Para diferenciar los diferentes servicios registrados bajo la interfaz Device, el desarrollador deberá añadir a dicho servicio propiedades especificas del dispositivo, como por ejemplo, Categoría, Modelo o Numero de Serie.
3. Cada vez se registra un dispositivo bajo la interfaz Device, OSGI internamente buscará todos Driver registrados en la plataforma para encontrar el más apropiado.
4. En OSGI un Driver es un servicio programado por el desarrollador registrado bajo la interfaz org.osgi.service.device.Driver.
5. Un servicio Driver debe implementar dos métodos: match y attach. Internamente OSGI invocará a los métodos match de todos los Driver instalados en la plataforma. Este método programado por el desarrollador, comparará las propiedades del servicio Device (Categoría, modelo, numero de serie etc..), para tratar de averiguar si dicho driver es apropiado para el dispositivo conectado.
6. Los métodos match de los driver retornan un entero, este numero será mayor o menor dependiendo del grado de similitud del Driver con las propiedades del dispositivo (Device). Internamente OSGI seleccionará como Driver mas apropiado para el dispositivo conectado, aquel que retorne un mayor numero en su método match.
7. OSGI invocará al método attach del Driver seleccionado. Este método attach, será programado por el desarrollador y proporcionará los mecanismos para manejar el dispositivo.

Imagínense que nuestro sensor basado en un microcontrolador PIC, es capaz de identificarse cuando se conecta fisicamente al PC (no es el caso, nuestro sensor es mas sencillo). Cuando lo conectemos, lo primero que podría hacer es enviar los bytes para su identificación por ejemplo:

- Categoría: RS232
- Modelo: Temp232
- Serial Number:X-0000001-A

Imagínense ahora que en nuestra plataforma OSGI, tenemos listenners escuchando en todos los puertos serie que tenga nuestro equipo, por ejemplo tenemos dos puertos serie y dos

sensores; uno de temperatura y otro de humedad. En teoría debería darnos igual en que puerto serie enchufemos cualquiera de los dos sensores, ya que gracias a esta especificación, el framework podrá buscar el driver mas apropiado para cada uno de los sensores, ayudado por la información proporcionada por el sensor.

Podemos decir, que es un caso similar a cuando conectamos a nuestro Pc un hardware nuevo, el sistema automáticamente intenta buscar el driver mas apropiado al hardware conectado. El driver será el encargado de manejar el dispositivo. Si nosotros intentamos desarrollar una aplicación que trabaje contra dicho dispositivo, utilizaremos las librerías proporcionadas por el driver. En OSGI este concepto llega un poco mas lejos, ya que esto es aplicable tanto a dispositivos conectados fisicamente, como a los conectados de manera inalámbrica o conectados virtualmente.

Para entender esta especificación, lo mejor será verlo de una manera práctica con nuestro sensor de temperatura.

En este ejemplo desarrollaremos tres bundles:

- **org.javahispano.rs232listenner**: Se encargará de detectar cuando se conecta un dispositivo serie y buscará el driver mas apropiado. En nuestro caso supondremos que solo se conectará a este puerto el sensor de temperatura. (Nuestro sensor no implementa ninguna lógica de identificación al ser conectado).
- **org.javahispano.sensortempdriver**: Driver controlador del sensor.
- **org.javahispano.sensortest**: Hará uso del Driver para trabajar con el sensor.

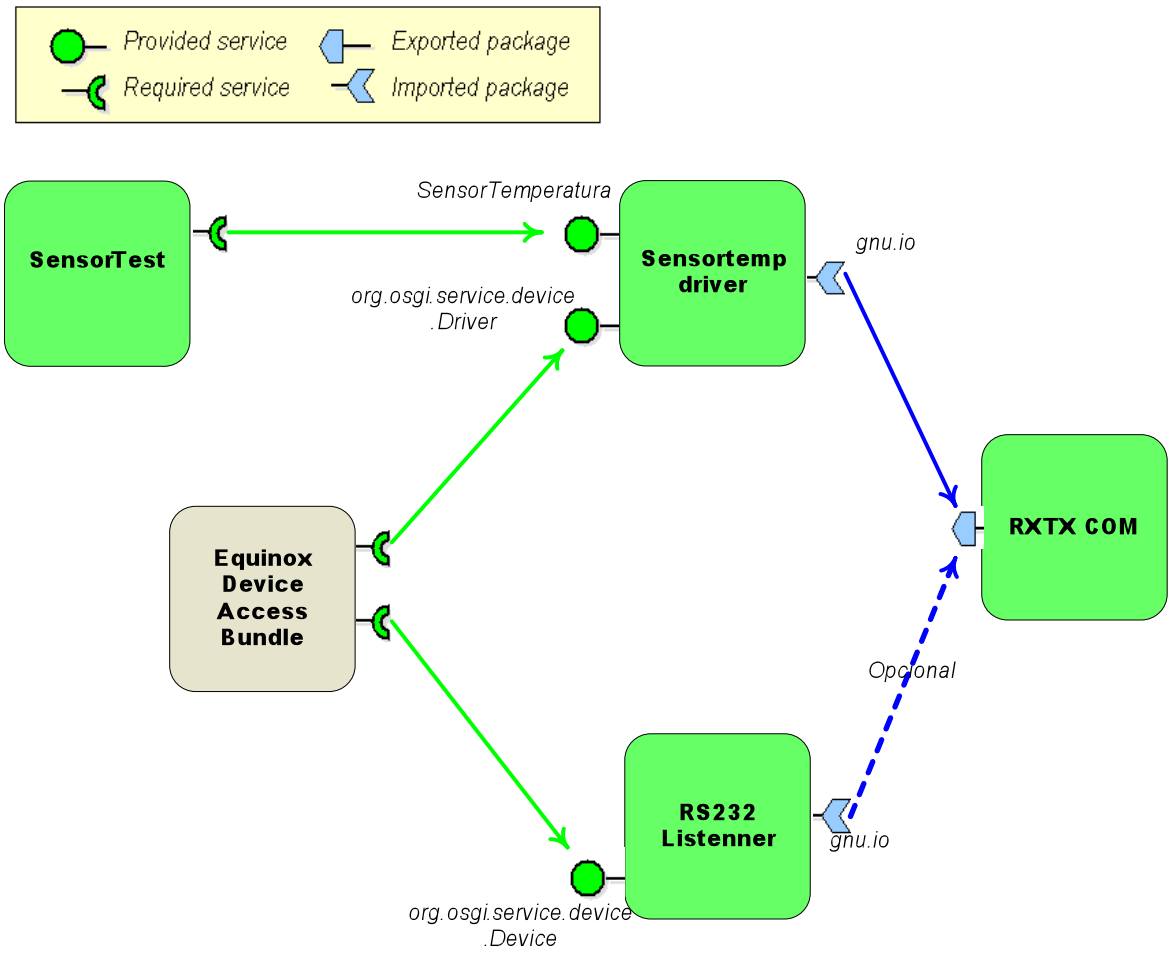


Figura 3.1.3 - 2 – Arquitectura Ejemplo Device Access

3.1.3.1 Detectando dispositivos

En teoría el bundle **org.javahispano.rs232listenner** debería ser capaz de detectar e identificar cuando se ha conectado un nuevo dispositivo a los puertos COM de nuestro sistema. Realizar esto, se convierte en un tema un poco mas tedioso y directamente relacionado con el API Serial RXTX, como nuestro objetivo con este tutorial es ver de la manera mas sencilla posible los entresijos de OSGI, el método start del Activator de nuestro bundle será equivalente al supuesto método que llamaría la plataforma al detectar e identificar el sensor de temperatura. Por lo tanto la acción de arrancar este bundle será equivalente a la supuesta detección automática del dispositivo.
Nuestro Activator invocará a la clase SerialDevice. Como nuestro sensor no se identifica, para el ejemplo hemos puesto los datos a mano (Categoría, Descripción del dispositivo y número de serie):

```
public class Activator implements BundleActivator {

	public void start(BundleContext bc) throws Exception {
		new SerialDevice(bc,"Serial", "Serial Temp Sensor", "XXXXX");
	}

	public void stop(BundleContext context) throws Exception {
	}
}
```

Listado 3.1.3.1 - 1 – Activator

La clase SerialDevice implementará la interfaz OSGI: org.osgi.service.device.Device, que nos obligará a declarar el método noDriverFound(), que será invocado cuando no se encuentre un driver apropiado para el dispositivo conectado. Lo que haremos al invocar al constructor de esta clase, será registrar un servicio OSGI bajo la interfaz org.osgi.service.device.Device, con las propiedades pasadas como parámetros (categoría, descripción y modelo):

```
package org.javahispano.rs232listenner;

import java.util.Dictionary;
import java.util.Hashtable;
import org.osgi.framework.BundleContext;
import org.osgi.service.device.Device;
import org.osgi.service.device.Constants;

public class SerialDevice implements Device{
	 private static final String[] clazzes=new String[] { Device.class.getName()  };
public SerialDevice(BundleContext bc,String category, String descripcion, String
model){
		Dictionary props=new Hashtable();
		props.put(Constants.DEVICE_DESCRIPTION,descripcion);
		props.put(Constants.DEVICE_CATEGORY,category);
		props.put(Constants.DEVICE_SERIAL,model);
		bc.registerService(clazzes,this,props);
	}
	public void noDriverFound() {
		System.out.println("NO SE HA ENCONTRADO NINGUN DRIVER
PARA EL DISPOSITVO CONECTADO");
	}
}
```

Listado 3.1.3.1 - 2 – SerialDevice

Cuando arranquemos el bundle se registrará este servicio, automáticamente OSGI buscará el driver mas apropiado que será aquel que se aproxime mejor a todas las características anexadas: DEVICE_DESCRIPTION, DEVICE CATEGORY y DEVICE_SERIAL.

3.1.3.2 Construyendo un Driver

Como hemos mencionado antes, nuestro driver será el bundle **org.javahispano.sensortempdriver**.
Para que nuestro bundle actúe como un posible driver, tendremos que registrar un servicio bajo la interfaz org.osgi.service.device.Driver. Implementar dicha interfaz, implica sobrescribir estos dos métodos:

- public int match(ServiceReference reference) throws Exception
- public String attach(ServiceReference reference) throws Exception

El método match será invocado cada vez que se registre un servicio bajo la interfaz Device. A través del ServiceReference que le llega como parámetro, podrá acceder a las propiedades del servicio y comprobar si se trata del dispositivo para el que el driver esta preparado. El mecanismo por el que OSGI decide cual de todos los drivers registrados en la plataforma es el mas apropiado para controlar el dispositivo, es a través del "int" retornado por el método match. El Driver que retorne mayor numero gana y la plataforma invocará el método attach de dicho Driver.

En nuestro caso el método attach registrará un nuevo servicio, idéntico al del ejemplo del apartado 3.1.1, que permitirá a otros bundles manejar el sensor.

```
package org.javahispano.sensortempdriver;

import org.osgi.framework.ServiceReference;
import org.osgi.service.device.Constants;
import org.osgi.service.device.Driver;

public class SerialDriver implements Driver{

	@Override
	public String attach(ServiceReference reference) throws Exception {
		System.out.println("Atacando Driver Temperatura");
//Registramos el Servicio que controlara el dispositivo
	reference.getBundle().getBundleContext().registerService(SensorTemperatura
.class.getName(),  new SensorTemperaturaImpl(), null);
		return null;
	}

	@Override
	public int match(ServiceReference reference) throws Exception {
		int match = -1;
		System.out.println("TEMP DRIVER MACH CHECK");
		if (reference != null) {
			String deviceCategory =
(String)reference.getProperty(Constants.DEVICE_CATEGORY);
			String deviceDescription =
(String)reference.getProperty(Constants.DEVICE_DESCRIPTION);
			String deviceSerialNumber =
(String)reference.getProperty(Constants.DEVICE_SERIAL);
			if (deviceCategory.equals("Serial")) {
				match = match + 3;
			}
			if (deviceDescription.equals("Serial Temp Sensor")) {
				match = match + 3;
			}
			if (deviceSerialNumber.equals("XXXXX")) {
				match = match + 4;
			}
		}
		return match;
	}
}
```

Listado 3.1.3.2 - 1 – SerialDriver

El método match de la clase anterior, comprueba las propiedades del servicio Device que se ha conectado y suma al entero de retorno, valores dependiendo de la cantidad de propiedades con las que concuerde.

Si este Driver resulta "ganador" entre todos los driver que estén registrados en la plataforma, inmediatamente será invocado el método attach, que registrará un servicio que los demás bundles podrán utilizar para interactuar con el sensor. Este servicio registrado será igual al del ejemplo del apartado 3.1.1.

Este servicio Driver lo registraremos al arrancar el bundle, a través de su Activator:

```
public class Activator implements BundleActivator {
        public void start(BundleContext context) throws Exception {
                context.registerService(Driver.class.getName(),
                                        new SerialDriver(), null);
        }

        public void stop(BundleContext context) throws Exception {
        }
}
```

Listado 3.1.3.2 - 2 – Driver Activator

Para consumir el servicio registrado por el driver, utilizaremos el bundle de test del apartado 2.1.2, que consume el servicio "SensorTemperatura" registrado por el Driver.

3.1.3.3 Ejecutando el ejemplo

Ahora que ya tenemos construidos los tres bundles, solo nos queda ejecutarlos en la plataforma. Para ello necesitaremos, al igual que en el caso del EventAdmin, el API de la especificación Device Access y la implementación correspondiente a la plataforma donde este corriendo, en nuestro caso Equinox. Por lo tanto, a parte de nuestros tres bundles hemos añadido a la plataforma estos dos nuevos módulos:

- **org.eclipse.osgi.services XXXX.jar**: Especificación OSGI de los servicios.
- **org.eclipse.equinox.device_XXX.jar**: Implementación Equinox del servicio Device Access.

Así pues, el config.ini de nuestro entorno debería quedar algo así:

```
osgi.clean=true
eclipse.ignoreApp=true

org.osgi.framework.bootdelegation=javax.jms
osgi.bundles=../serial/plugins/rxtxcomm_api-2.1.7.jar@start, \
../serial/plugins/rxtxcomm-linux-pc-2.1.7.jar, \
org.eclipse.osgi.services_3.1.200.v20071203.jar@start,\
org.eclipse.equinox.device_1.0.1.v20080303.jar@start,\
../driver/plugins/org.javahispano.sensortest_1.0.0.jar@start, \
../driver/plugins/org.javahispano.sensortempdriver_1.0.0.jar@start, \
../driver/plugins/org.javahispano.rs232listenner_1.0.0.jar, \
```

Listado 3.1.3.3 - 1 – config.ini

Cuando arranquemos la plataforma, lo primero que ocurrirá es que se registrará nuestro Driver, pero este no comenzará a funcionar, hasta que no se relacione con dispositivo (con un servicio org.osgi.service.device.Device).

Si ahora arrancamos el bundle **org.javahispano.rs232listenner**, se registrará el Device y la plataforma buscará el driver mas apropiado, que en nuestro caso será el bundle org.javahispano.sensortempdriver, este driver registrará un servicio para que los demás bundles puedan consumir los datos del sensor.

3.1.4 Método 4: Wire Admin Service

Wire Admin Service es una especificación que define como se interconectan entre ellos un servicio productor de datos y un consumidor de los mismos. Esta especificación trata de minimizar el acoplamiento entre los bundles conectados. El servicio Wire Admin será el encargado de gestionar las conexiones entre bundles, determinando que consumidor será conectado a que "publicador".

En la especificación Wire Admin, los Bundles participan en el proceso de conexión registrando servicios que consumen o producen datos. Este servicio de Wire Admin se ocupará de conectar unos con otros.

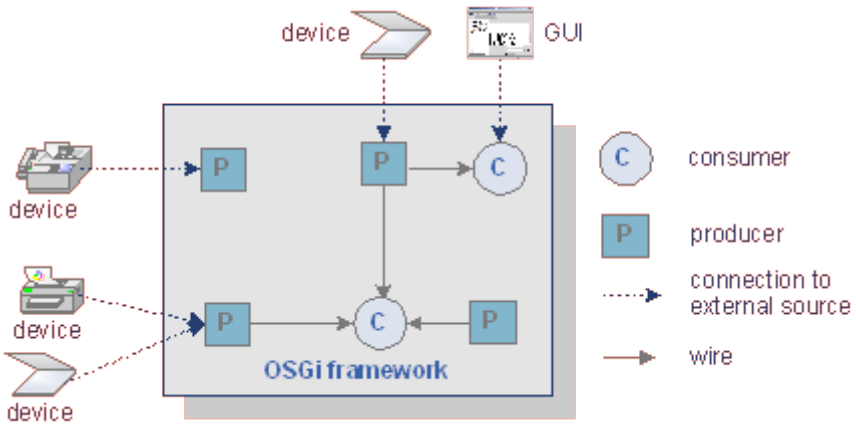


Figura 3.1.4 - 1 – Sistema Interconexión Wire Admin Service

Un productor será aquel servicio que produzca una salida de datos. Un productor se debe registrar bajo la interfaz org.osgi.service.wireadmin.Producer.
Un consumidor será un servicio que consume los datos de un productor. Un consumidor se debe registrar bajo la interfaz org.osgi.service.wireadmin.Consumer.

A continuación veremos este servicio aplicado a nuestro ejemplo del sensor de temperatura, para ello necesitaremos tres bundles:

- **org.javahispano.sensorreader:** Será el productor, leerá del puerto serie y producirá una salida de datos.
- **org.javahispano.sensortest:** Será el consumidor, consumirá los datos del "sensorreader".
- **org.javahispano.sensorwiredconector:** Será el encargado de realizar la conexión de los dos bundles anteriores.

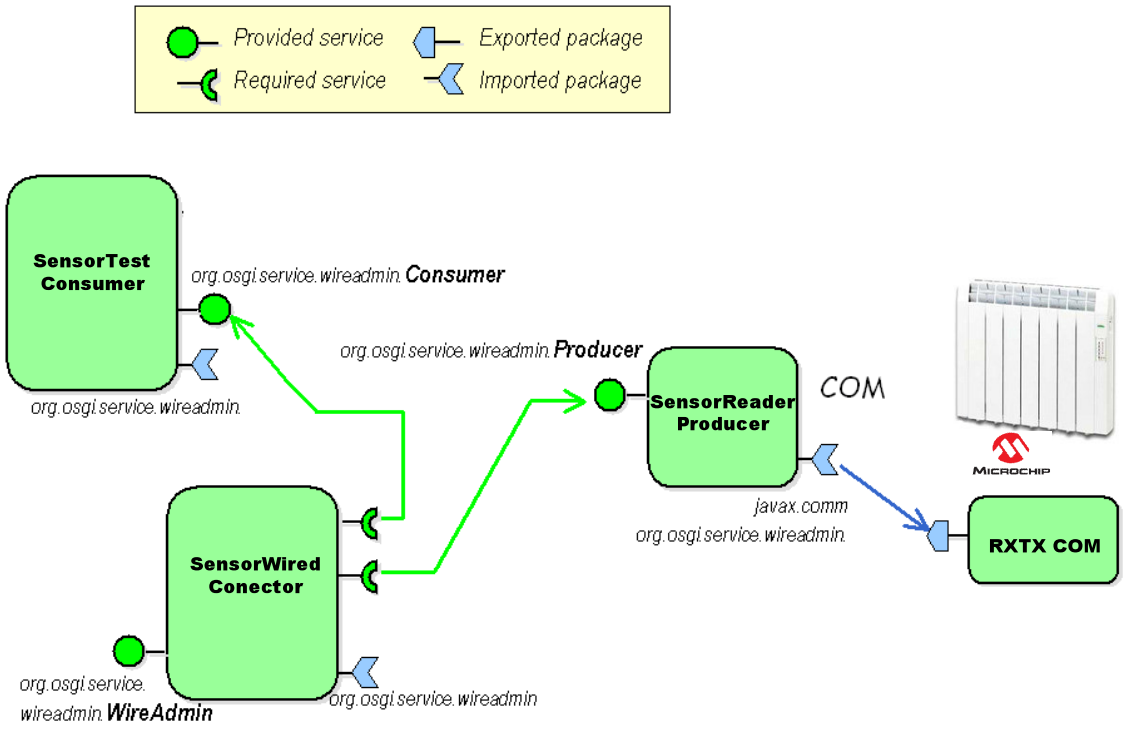


Figura 3.1.4 - 2 – Conexión de Bundles que participan en el caso práctico.

3.1.4.1 Wire Admin Producer

Para crear un "Productor" debemos registrar un servicio bajo la interfaz org.osgi.service.wireadmin.Producer asignándole un PID.

Implementar la interfaz "Producer", nos obliga a sobrescribir dos métodos:

- *public void consumersConnected(Wire[] wires);*

Método que será invocado cuando algún consumidor se conecta al bundle Productor. Con el vector de objetos pasado como parámetro, podremos obtener la referencia a los objetos consumidores. En nuestro ejemplo, en este método, cada vez que se conecte un consumidor, actualizaremos los datos de todos los "Wires" conectados.

- public Object polled(Wire wire);

Método que será invocado para publicar datos.

Para intentar simplificar la cantidad de código, esta vez, lo he programado todo en una misma clase, implementando a la vez a tres interfaces:

- **org.osgi.service.wireadmin.Producer**: Como hemos mencionado antes será la encargada de publicar los datos.
- **org.osgi.framework.BundleActivator:** Encargada de controlar el ciclo de vida de nuestro bundle.
- **gnu.io.SerialPortEventListener:** Listenner del puerto serie. Cada vez que nos llegue un dato del puerto serie, lo publicaremos invocando al método polled(Wire wire).

Nuestro bundle comenzará a funcionar cuando arranque e invoque a su método Start:

```
 ...
public class ProducerTest implements Producer,SerialPortEventListener,
BundleActivator {
 ...
Wire[] wiresconectados;
public void start (BundleContext bc) throws BundleException {
        configureSerialListenner();
   Hashtable props = new Hashtable();
   Class[] flavors = new Class[] {Double.class};
   props.put(WireConstants.WIREADMIN_PRODUCER_FLAVORS, flavors);
   props.put("service.pid", "producer.all");
   reg = bc.registerService(Producer.class.getName(), this, props);
 }
```

Listado 3.1.4.1 - 1 – Producer

Al comenzar el método Start, se invoca a la función configureSerialListenner para configurar el puerto serie y registrar el listenner.

Al igual que en otros ejemplos que hemos visto anteriormente, la jugada consiste en registrar un servicio bajo una interfaz concreta (Producer) y con una serie de propiedades. En este caso le hemos añadido dos propiedades:

- **WireConstants.WIREADMIN_PRODUCER_FLAVORS**: Tipo de dato que será publicado, en nuestro ejemplo le hemos indicado que es de tipo Double, ya que se trata de una temperatura.
- **service.pid**: Identificador del servicio. Le podremos dar el valor que queramos, en nuestro ejemplo, al proceso le hemos llamado "producer.all".

El siguiente método que veremos de nuestro productor será, la configuración del puerto serie:

```
private void configureSerialListenner(){
          try {
                    String defaultPort = "/dev/ttyS0";
                    CommPortIdentifier portId;
                     SerialPort serialPort;
                    portId =       CommPortIdentifier.getPortIdentifier("/dev/ttyS0");
                              serialPort = (SerialPort) portId.open("SimpleReadApp", 2000);
                              serialPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
                                        SerialPort.STOPBITS_1,
SerialPort.PARITY_NONE);
                              inputStream = serialPort.getInputStream();
                              serialPort.addEventListener(this);
                              serialPort.notifyOnDataAvailable(true);
                              serialPort.notifyOnCarrierDetect(true);
                    } catch (Exception e) {
                              e.printStackTrace();
                    }
  }
public void serialEvent(SerialPortEvent event) {

                    switch (event.getEventType()) {
                    case SerialPortEvent.DATA_AVAILABLE:
                              byte[] readBuffer = new byte[20];
                              try {
                                        while (inputStream.available() > 0) {
                                                  inputStream.read(readBuffer);
                                        }
                                        String strTemperatura = new String(readBuffer) + "";
                                        temperatura = new Double(strTemperatura);
                                        consumersConnected(wiresConectados);
                              } catch (IOException e) {
                                        e.printStackTrace();
                              }
                              break;
                    }
          }
```

Listado 3.1.4.1 - 2 – Producer II

El código anterior, a estas alturas del tutorial, no implica nada que no hayamos visto, registramos el listenner del puerto serie (configureSerialListenner()) e implementamos el método serialEvent. Cada vez que detectemos un nuevo dato en el puerto serie lo leeremos,

lo guardaremos en una variable global e informaremos a todos los consumidores conectados (consumersConnected(wiresConectados));

Por ultimo veremos los métodos consumersConnected y polled derivados de la interfaz Producer:

```
public void consumersConnected(Wire[] wires) {
  if (wires != null) {
                wiresConectados = wires;
    for (int i = 0; i < wires.length; i++) {
      wires[i].update(polled(wires[i]));
    }
  }
}

/** This method is responsible for creating the output */
public Object polled(Wire wire) {
        System.out.println("POLL");
  return temperatura;
}
```

Listado 3.1.4.1 - 3 – Producer III

Como hemos mencionado antes, el método consumersConnected será invocado por la plataforma automáticamente cada vez que se conecte un consumidor aunque también lo podremos invocar manualmente (en este caso desde el método serialEvent).

3.1.4.2 Wire Admin Consumer

Para crear un "Consumer" debemos registrar un servicio bajo la interfaz org.osgi.service.wireadmin.Consumer asignándole un PID. Implementar la interfaz "Consumer", nos obliga a sobrescribir dos métodos:

- *public void producersConnected(Wire[] wires)*

Este método será invocado cada vez que el Consumidor se conecte al Productor. También será invocado siempre y cuando el productor al que esta conectado sufra algún cambio: se conecte, se desconecte o se actualice. Como parámetro a esta función recibiremos el array de objetos a los que esta conectado el consumidor.

- *public void updated(Wire wire, Object value)*

Este método será invocado cada vez que se actualice el valor retornado por el productor.

También esta a ocasión con el objetivo de simplificar el código, hemos creado el bundle consumidor con una única clase que implementa a dos interfaces:

- **org.osgi.service.wireadmin.Consumer:** Como hemos mencionado antes será la encargada de consumir los datos.
- **org.osgi.framework.BundleActivator:** Encargada de controlar el ciclo de vida de nuestro bundle.

La única clase del bundle consumidor sería:

```
import org.osgi.framework.BundleContext;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.ServiceRegistration;
import org.osgi.framework.BundleException;
import org.osgi.service.wireadmin.*;

public class ConsumerImpl implements Consumer, BundleActivator {

  private ServiceRegistration reg;

  public void start(BundleContext bc) throws BundleException {
    Hashtable prop = new Hashtable();
    //Propiedad para filtrar los tipos de datos aceptados por el cosumidor
    prop.put(WireConstants.WIREADMIN_PRODUCER_FLAVORS,
                  new Class[] {Double.class});
   prop.put("service.pid", "consumer.all");//PID del consumidor
    //Registrar servicio Consumidor
    reg = bc.registerService(Consumer.class.getName(), this, prop);
  }
  public void stop(BundleContext bc) throws BundleException {
    reg.unregister();
  }

  public void producersConnected(Wire[] wires) {  }

  public void updated(Wire wire, Object value) {
    System.out.println("Consumidor: Nuevo valor recibido " + value);
  }
}
```

Listado 3.1.4.2 - 1 – ConsumerImpl

En el código anterior, al arrancar el bundle registramos el servicio consumidor. Para registrar este servicio hemos fijado dos propiedades:

- WireConstants.WIREADMIN_PRODUCER_FLAVORS: Tipos de datos aceptados por el consumidor.
- service.pid: PID del servicio consumidor

No hemos programado nada para el método producersConnected, en esta ocasión no nos interesa realizar ninguna acción ante cualquier cambio en el estado del productor. El método updated, será invocado cada vez que el productor publique sus datos, nosotros simplemente mostramos una traza por consola con el valor recibido.

3.1.4.3 Conectando los bundles

Una vez que tengamos el productor y el consumidor construidos, solo nos falta conectarlos entre si, para ello utilizaremos el servicio WireAdmin.

La interfaz WireAdmin tiene un método createWire(PID Producer, PID Consumer, Propiedades o filtro de la conexión) que nos permitirá conectar nuestros dos bundles. Para realizar esta conexión hemos utilizado un nuevo bundle (sensorwiredconector) que captura el servicio WireAdmin e invoca al método createWire comentado anteriormente.

```
public void start(BundleContext bc) throws BundleException {
        // getting a Wire Admin service reference
        ServiceReference waRef =
        bc.getServiceReference(WireAdmin.class.getName());

    WireAdmin wa = (WireAdmin) bc.getService(waRef);
   Wire wire = wa.createWire("producer.all", "consumer.all", null);
...
```

Listado 3.1.4.3 - 1 – CreateWire

Con el objeto Wire podremos acceder a las propiedades de la conexión o incluso invocar al envío de datos a través del método wire.poll().
En el listado anterior todavía no le hemos aplicado ningún filtro para la conexión, pero debemos saber que existen una serie de propiedades básicas de la conexión:

- **WireConstants.WIREVALUE_ELAPSED:** Filtra el tiempo transcurrido desde que se recibió el ultimo dato. Con esta propiedad podremos ignorar los valores producidos con menor periodicidad a la indicada en este filtro.
- **WireConstants.WIREVALUE_CURRENT:** Filtra que el valor recibido cumpla la condición aplicada en este filtro.

- **WireConstants.WIREVALUE_DELTA_ABSOLUTE:** Filtra que el valor actual recibido menos el valor recibido previamente cumpla la condición aplicada en este filtro.

En el javadoc de la clase WireConstants, podréis ver algunas propiedades mas de este tipo de filtros: http://www.osgi.org/javadoc/r3/org/osgi/service/wireadmin/WireConstants.html
Por lo tanto a nuestro código anterior le aplicaremos un filtro, para solo usar los valores enviados cada segundo:

```
 private ServiceReference waRef;
    private WireAdmin wa;
    private Wire wire;

    public void start(BundleContext bc) throws BundleException {
        // getting a Wire Admin service reference
        waRef = bc.getServiceReference(WireAdmin.class.getName());

        if (waRef == null) {
        throw new BundleException("Unable to get Wire Admin service
reference!");
        }

        wa = (WireAdmin) bc.getService(waRef);
        if (wa == null) {
        throw new BundleException("Wire Admin service has not been
registered!");
        }

        Hashtable wireProps = new Hashtable();
        String wireFilter = "(" +  WireConstants.WIREVALUE_ELAPSED +
">=1000)";
        //+ "(" + WireConstants.WIREVALUE_CURRENT + ">=0)"
        //+ "(" + WireConstants.WIREVALUE_CURRENT + "<=100)"
        //+ "(" + WireConstants.WIREVALUE_DELTA_ABSOLUTE + "<=20)" +
                //    + ")";
        wireProps.put(WireConstants.WIREADMIN_FILTER, wireFilter);
        wire = wa.createWire("producer.all", "consumer.all", wireProps);
    }

    public void stop(BundleContext bc) {
        wa.deleteWire(wire);
        bc.ungetService(waRef);
    }
```

Listado 3.1.4.3 - 2 – CreateWire Filter

3.2 Conectando nuestra plataforma con el mundo exterior

Hasta ahora hemos visto como trabajar dentro de un mismo Host con los datos de un sensor conectado a través del puerto serie. En esta sección la idea es poder propagar estos datos del sensor a través de un red WIFI, haciendo llegar estos valores a un servidor central conectado a Internet y que permitirá controlar desde cualquier lugar del mundo nuestra calefacción a través de un entorno Web. La parte de construir un servidor Web será objeto de la última entrega de esta serie de tutoriales, pero en esta entrega nos centraremos en propagar los datos del sensor a través de la red WIFI. Como en el caso anterior, dada la flexibilidad de OSGI, podremos realizar esto, mediante diferentes mecanismos, algunos de ellos los comentaremos a continuación.

3.2.1 Método 1: Servicios Web

Como venimos repitiendo a lo largo de estos tutoriales, OSGI es un framework modular, que otorga una gran flexibilidad al entorno, permitiéndonos empotrar en él, casi cualquier aplicación basada en Java. En esta ocasión añadiremos a nuestro entorno un servidor basado en AXIS2.
Axis2 es el proyecto de Apache que nos facilita el trabajo con servicios Web. Los servicios Web son sistemas software diseñados para soportar interoperabilidad entre dos máquinas sobre una red. Para conseguir esta interoperabilidad (independencia de tecnología y sistema operativo), los servicios Web están basados en tecnologías estándares: XML, WSDL (Web Service Description Language), SOAP (Simple Object Access Protocol), UDDI (Universal Description, Discovery and Integration).
No entraremos en detalle acerca de como se trabaja con servicios Web o con AXIS2, por ello recomiendo tener algunos conocimientos básicos sobre Servicios Web antes de leer esta sección.
Montar un entorno Apache Axis2 en OSGI, resulta bastante engorroso, ya que Axis arrastra multitud de dependencias, llegando a desesperar a cualquier desarrollador. El lado bueno, es que una vez montado el entorno, construiremos servicios Web de forma igual de sencilla a la que contruiamos servicios OSGI.
Para agilizar el aprendizaje, adjunto con este tutorial y los fuentes, el entorno Axis2-Equinox, listo para usar. Este entorno fue montado siguiendo el manual del siguiente link: http://svn.apache.org/repos/asf/axis/axis2/java/core/scratch/java/saminda/osgi_test/axis2_osgi_integration.pdf
Actualmente si rebuscáis en Google, seguramente encontrareis otros entornos Axis2-Osgi. Yo solo he probado el entorno que aquí os presento, pero estoy convencido que según va pasando el tiempo, integrar OSGI y Axis2 se va simplificando.
Dentro del archivo webservices.zip adjunto a este tutorial encontraremos dos carpetas:

- **Equinox-axis2:** Contiene el entorno de ejecución Equinox, con las dependencias de AXIS debidamente configuradas.
- **org.apache.axis2.osgi.bundle:** Proyecto eclipse con las librerías de AXIS. Este proyecto lo importaremos en el Workspace, para que en tiempo de desarrollo nuestro bundle sea capaz de acceder a las dependencias.

Una vez que tenemos preparado el entorno, comenzaremos con el desarrollo. La siguiente figura muestra la arquitectura de nuestro nuevo entorno:

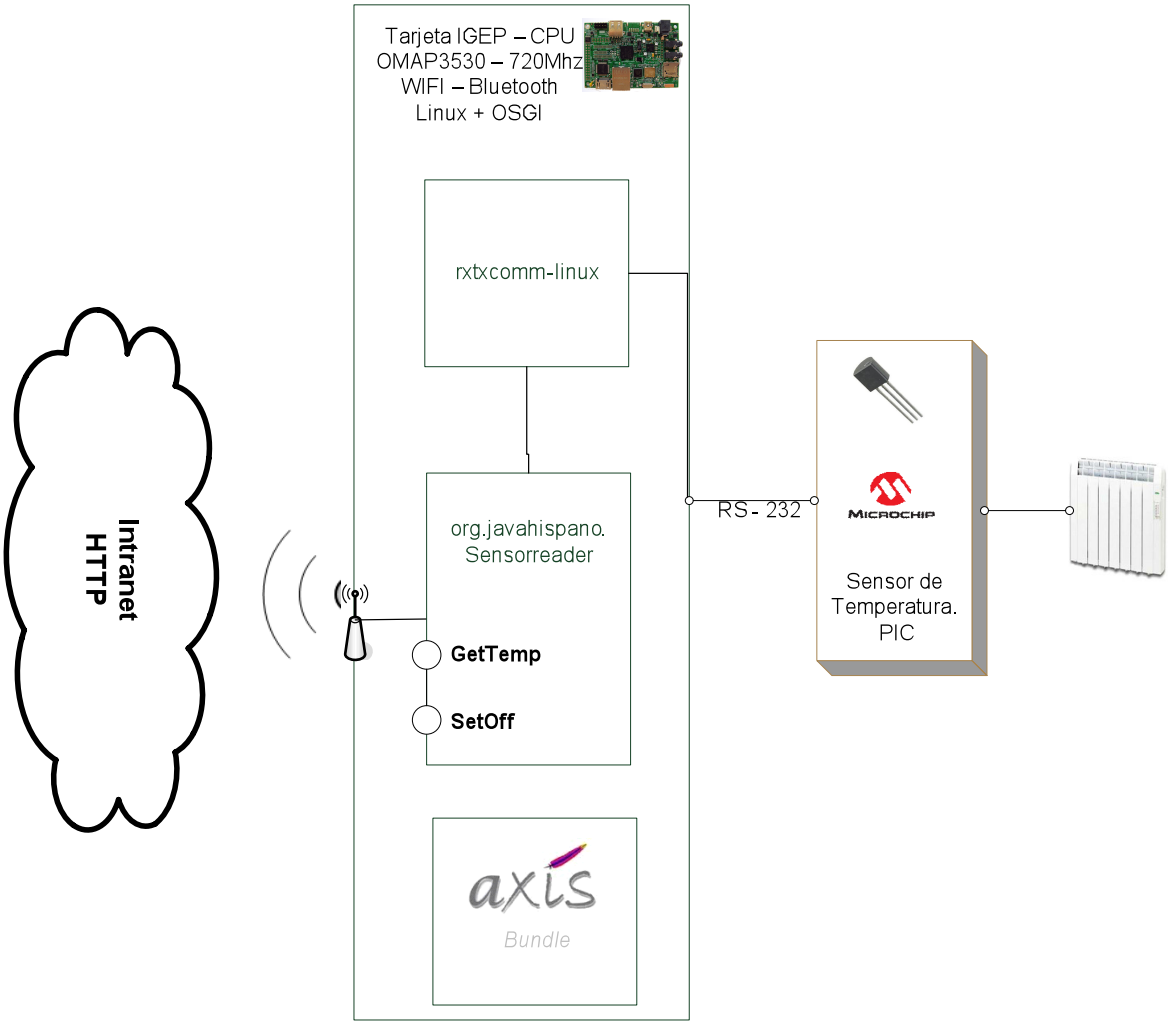


Figura 3.2.1 - 1 – Arquitectura ejemplo servicios Web

Como he mencionado antes, crear un servicio Web con Axis2, será exactamente igual que crear un servicio con OSGI, por ello partiremos del ejemplo mas sencillo del apartado 3.1.1. En este apartado registrábamos el servicio del *org.javahispano.sersorreader* de la siguiente manera:

```
public void start(BundleContext context) throws Exception {
        registration =
context.registerService(SensorTemperatura.class.getName(),
          new SensorTemperaturaImpl(), null);
   }
   public void stop(BundleContext context) throws Exception {
        registration.unregister();
   }
```

Listado 3.2.1 - 1 – Activator

Para registrar este servicio como un servicio Web de AXIS2, tan solo deberemos hacer lo siguiente:

```
import org.apache.axis2.osgi.deployment.tracker.WSTracker;

public class Activator implements BundleActivator {
   private ServiceRegistration registration;

   public void start(BundleContext context) throws Exception {
        Dictionary prop = new Properties();
        prop.put(WSTracker.AXIS2_WS, "SensorService");
        registration =
context.registerService(SensorTemperatura.class.getName(),
          new SensorTemperaturaImpl(), prop);
   }
   public void stop(BundleContext context) throws Exception {
        registration.unregister();
   }
}
```

Listado 3.2.1 - 2 – Activator Axis2

Tan solo hemos importado el paquete *org.apache.axis2.osgi.deployment.tracker.WSTracker* (no deberemos olvidar registrar este paquete en el manifest.mf) y le hemos añadido una propiedad al servicio publicado:

prop.put(WSTracker.AXIS2_WS, "SensorService");

Donde *SensorService* es el nombre que se le asignará al servicio Web.
Si a continuación arrancamos la plataforma con este bundle instalado, ya tendremos desplegado nuestro primer servicio Web. Para acceder a él, desde el navegador pondremos la siguiente URL:

http://localhost:8080/services/SensorService?wsdl

La configuración del puerto HTTP se realizará en el script de arranque de la plataforma. Si habéis utilizado el entorno adjunto a este tutorial, podréis editar el fichero equinox.sh y modificar el puerto de escucha con la propiedad:

-Dorg.osgi.service.http.port=8080

En esta sección no hemos entrado en detalle acerca de la configuración del servidor de aplicaciones (*jetty*) sobre el que se monta *AXIS2*, ya que profundizaremos en ello en la última entrega de esta serie de tutoriales.

Si revisáis el código adjunto del bundle *org.javahispano.sersorreader*, podréis comprobar, que a la interface además del método *getTemp*, le hemos añadido un nuevo método al servicio: *setOff(boolean apagar),* que nos servirá para apagar o encender la calefacción:

```
package org.javahispano.sensorreader;

public interface SensorTemperatura {
    public double getTemp();
    public void setOff(boolean apagar);
}
```

Listado 3.2.1 - 3 – Nueva Interface SensorTemperatura

La construcción del cliente para consumir el Servicio Web la dejaremos para la ultima entrega de este tutorial, de momento podréis pobrar el funcionamiento de nuestro servicio Web usando alguna herramienta tipo SOAPUI (soapui.org).

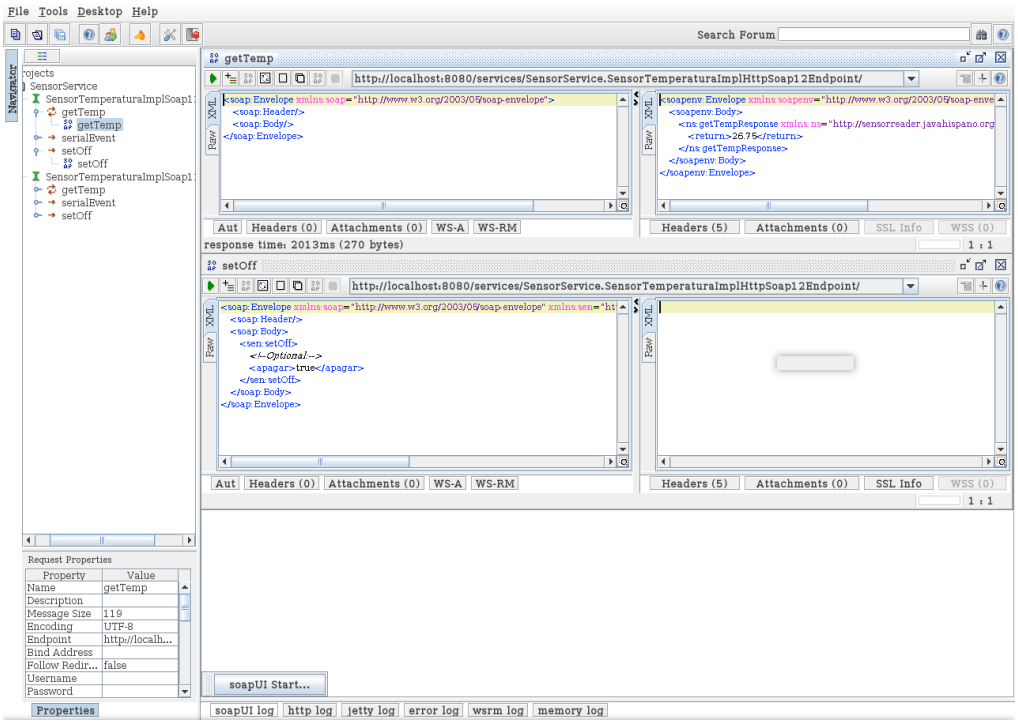


Figura 3.2.1 - 2 – GUI SOAPUI

3.2.2 Método 2: Especificación UpnP

Para escribir esta sección me he servido de la siguiente documentación, la cual aconsejo leer para extender y afinar lo expuesto en este apartado:

- La documentación sobre UpnP de Apache Felix: http://felix.apache.org/site/apache-felix-upnp.html
- Documentación de un proyecto de fin de carrera de la Universidad Carlos III de Madrid: http://e-archivo.uc3m.es/bitstream/10016/7353/1/UPnPMediaRenderer-PFC.pdf

UPnP, de las siglas en inglés Universal Plug and Play, es un protocolo de transmisión de datos punto a punto para la comunicación entre aplicaciones. Define una arquitectura software, abierta y distribuida, que proporciona una forma de conectividad entre distintos equipos pertenecientes a una red, permitiendo el intercambio de información y datos entre los mismos.

Usando el modelo UPnP, un dispositivo puede agregarse dinámicamente a una red, obtener una dirección IP, anunciar sus servicios, y saber de la existencia de otros dispositivos, todo esto de forma automática y transparente para el usuario final.

La arquitectura UPnP define la interacción entre un Punto de Control UPnP y un dispositivo UPnP. Un UPnP Control Point maneja y coordina la comunicación entre un dispositivo servidor de contenido y un dispositivo consumidor de contenido.

La especificación OSGI UPnP define una serie de interfaces las cuales pueden ser usadas por los desarrolladores para construir dispositivos UPnP y puntos de control UPnP en la plataforma de servicios OSGI. Desde el punto de vista de OSGI, los dispositivos UPnP son servicios registrados en la plataforma, por lo tanto las distintas fases del protocolo UPnP, tal como se define en la especificación UPnP™ Device Architecture (UDA 1.0), se han localizado en los mecanismos de descubrimiento y de notificación que ofrece OSGi. El framework recubrirá los aspectos del protocolo UPnP, de manera que el desarrollador no tiene por que tener conocimientos avanzados de dicho protocolo.

Para realizar nuestros ejemplos hemos usado la implementación de la especificación UPnP de Apache Felix y la hemos incrustado en nuestro Equinox.
Para ello nos hemos descargado de la página de Apache Felix (http://felix.apache.org/site/downloads.cgi) los siguientes Bundles:

- **UPnP Base Driver:** Implementación de Apache Felix de la especificación UPnP. Este bundle actuará como puente entre la red UPnP y OSGI.
- **UPnP Extra:** Utilidades extras para trabajar con la red UPnP.
- **UPnP Tester:** Bundle con interfaz gráfica para probar nuestros servicios UPnP.

También podremos descargarnos del repositorio de código fuente (http://felix.apache.org/site/building-felix.html)los ejemplos de UPnP.
Para este tutorial me he servido de dos ejemplos (podéis encontrar más ejemplos en la página de Felix):

- **Clock:** Reloj que emite la hora a través de la Red UPnP. Tiene métodos o acciones UPnP para retornar la hora actual o ponerlo en hora.
- **TV:** Dispositivo que simula la pantalla de una televisión. Recibirá los mensajes enviados por el reloj, imprimiéndolos en la pantalla.

Basándonos en estos dos ejemplos, crearemos el dispositivo UPnP sensor de temperatura.

3.2.2.1 Construyendo un dispositivo UPnP

Para construir un dispositivo UPnP, tendremos que implementar una serie de interfaces, tal y como representa la siguiente figura:

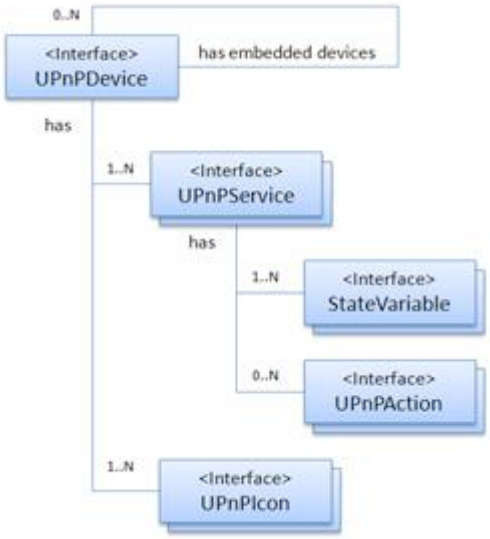


Figura 3.2.2.1 -1 Interfaces UPnP Device

- **UPnPDevice:** Representa un dispositivo UPnP. Un dispositivo UPnP puede contener otros dispositivos UPnP y servicios (UPnP Services).
- **UPnP Service:** Un dispositivo UPnP consiste en un número de servicios. Un servicio UPnP tiene cierto número de variables de estado UPnP (UPnP State Variable), que pueden ser consultadas o modificadas a través de acciones UPnP (UPnPAction).
- **UPnPAction:** Las acciones UPnP forman parte de los servicios UPnP y pueden trabajar sobre las StateVariables de un servicio. Estas acciones podrán ser invocadas por otros dispositivos conectados a la red.
- **StateVariable:** Variable asociada a un servicio UPnP.

- **LocalStateVariable:** Interface que extiende de UPnPStateVariable, cuando dicha variable es implementada localmente.

- **UPnPIcon:** Clase que representa el icono asociado a un dispositivo UPnP.

Nuestro bundle org.javahispano.sensorreader implementará las interfaces comentadas anteriormente:

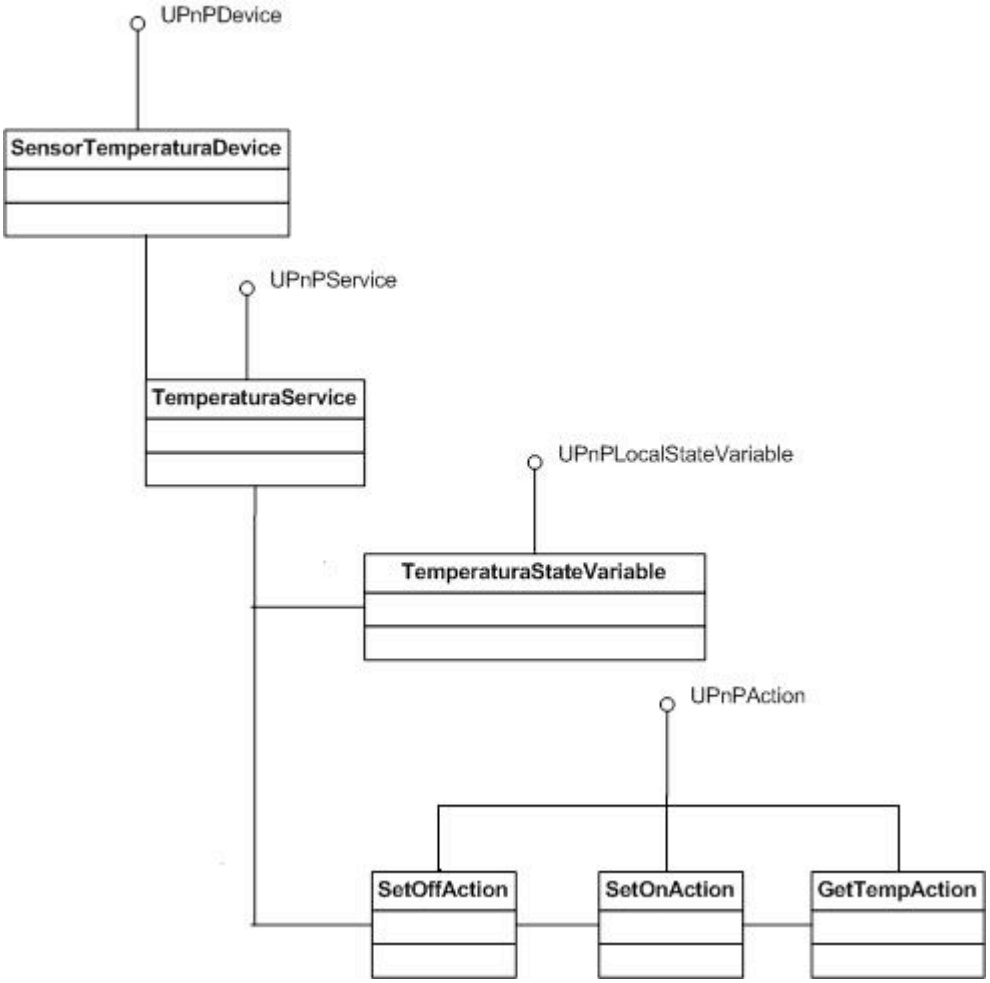


Figura 3.2.2.1 -2 Estructura UPnP Device Ejemplo

Una vez implementadas las interfaces reflejadas anteriormente, registraremos el dispositivo UPnP como un servicio OSGI. Esta acción en nuestro caso la realizaremos en el Activator:

```
package org.javahispano.sensorreader;

import java.util.Dictionary;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.upnp.UPnPDevice;


public class Activator implements BundleActivator {
   private ServiceRegistration registration;
  SensorTemperaturaDevice sensorTemperaturaDevice;

   public void start(BundleContext context) throws Exception {
         sensorTemperaturaDevice = new SensorTemperaturaDevice(context);
         Dictionary dict = sensorTemperaturaDevice.getDescriptions(null);

COMSensor.getInstance().doConfigureUPnPDevice(sensorTemperaturaDevice);
         registration = context.registerService(
                        UPnPDevice.class.getName(),
                        sensorTemperaturaDevice,
                        dict
                  );
   }
   public void stop(BundleContext context) throws Exception {
         registration.unregister();
   }
}
```

Listado 3.2.2.1 -1 Activator

En el listado anterior registramos el dispositivo UPnP como un servicio OSGI. También aprovecharemos para configurar el "listenner" del puerto COM. (COMSensor.getInstance().doConfigureUPnPDevice(...)).

La clase SensorTemperaturaDevice registrada como servicio en el listado anterior, implementará la interfaz UPnPDevice.

Implementar a la interfaz UPnPDevice nos obliga a escribir al menos los siguientes métodos:

- *java.util.Dictionary* **getDescriptions**(java.lang.String locale)

  Conjunto de propiedades UPnP. Pueden ser internacionalizadas a través del uso de cadenas ISO de localización.

- *UPnPIcon[]* **getIcons**(java.lang.String locale)

Lista de todos los iconos asociados a un dispositivo.

- *UPnPService* **getService**(java.lang.String serviceId)

  Localiza un servicio específico a través de su servicieId.

- *UPnPService[]* **getServices**()

  Lista de todos los servicios de un dispositivo.

Implementando los métodos citados anteriormente, nuestra clase SensorTemperaturaDevice quedaría:

```
import org.osgi.framework.BundleContext;
import org.osgi.service.upnp.UPnPDevice;
import org.osgi.service.upnp.UPnPIcon;
import org.osgi.service.upnp.UPnPService;
import org.osgi.service.upnp.UPnPStateVariable;



public class SensorTemperaturaDevice implements UPnPDevice    {
   final private String DEVICE_ID =
"uuid:temperaturesensor+"+Integer.toHexString(new
Random(System.currentTimeMillis()).nextInt());
   private Dictionary dic;
   private UPnPService[] services;
   private TemperaturaService temperaturaService;
   public static UPnPEventNotifier notifier = null;
   private BundleContext context;

   public SensorTemperaturaDevice(BundleContext context){
         this.context = context;
         temperaturaService =  new TemperaturaService();
         services = new UPnPService[]{temperaturaService};
         setupDeviceProperties();
   }

   private void setupDeviceProperties(){
         dic = new Properties();
         dic.put(UPnPDevice.UPNP_EXPORT,"");
         dic.put(org.osgi.service.device.Constants.DEVICE_CATEGORY,
                  new String[]{UPnPDevice.DEVICE_CATEGORY}
         );

         dic.put(UPnPDevice.FRIENDLY_NAME,"Javahispano Temperature
```

```
Sensor");
        dic.put(UPnPDevice.MANUFACTURER,"Javahispano - Roberto
Montero");
        dic.put(UPnPDevice.MANUFACTURER_URL,"http://javahispano.org");
        dic.put(UPnPDevice.MODEL_DESCRIPTION,"UPnP Specification
Tests");
        dic.put(UPnPDevice.MODEL_NAME,"Test Model");
        dic.put(UPnPDevice.MODEL_NUMBER,"1.0");
        dic.put(UPnPDevice.MODEL_URL,"http://javahispano.org");
        dic.put(UPnPDevice.SERIAL_NUMBER,"1234567893");
        dic.put(UPnPDevice.TYPE,"urn:schemas-upnp-
org:javahispano:sensorreader:1");
        dic.put(UPnPDevice.UDN,DEVICE_ID);
        dic.put(UPnPDevice.UPC,"12134567893");

        HashSet types = new HashSet(services.length+5);
        String[] ids = new String[services.length];
        for (int i = 0; i < services.length; i++) {
                ids[i]=services[i].getId();
                types.add(services[i].getType());
        }

        dic.put(UPnPService.TYPE, types.toArray(new String[]{}));
        dic.put(UPnPService.ID, ids);
    }

    public Dictionary getDescriptions(String arg0) {
        return dic;
    }

    public UPnPIcon[] getIcons(String arg0) {
        return null;
    }

    public UPnPService getService(String serviceId) {
        if  (serviceId.equals(temperaturaService.getId())){
             return temperaturaService;
        }
        return null;
    }

    public UPnPService[] getServices() {
        return services;
    }
}
```

Listado 3.2.2.1 -2 SensorTemperaturaDevice

Al invocar al constructor de la clase del listado anterior, crearemos una instancia de la clase TemperaturaService que guardaremos en el array de servicios UPnP:

```
temperaturaService =  new TemperaturaService();
services = new UPnPService[]{temperaturaService};
```

A continuación configura una serie de propiedades (setupDeviceProperties()) que identificarán al dispositivo en una red UPnP. Los nombres de estas propiedades los podemos encontrar como constantes de la clase UPnPDevice.

```
dic.put(UPnPDevice.UPNP_EXPORT,"");
dic.put(org.osgi.service.device.Constants.DEVICE_CATEGORY,
        new String[]{UPnPDevice.DEVICE_CATEGORY}
);
dic.put(UPnPDevice.FRIENDLY_NAME,"Javahispano Temperature Sensor");
dic.put(UPnPDevice.MANUFACTURER,"Javahispano - Roberto Montero");
dic.put(UPnPDevice.MANUFACTURER_URL,"http://javahispano.org");
dic.put(UPnPDevice.MODEL_DESCRIPTION,"UPnP Specification Tests");
dic.put(UPnPDevice.MODEL_NAME,"Test Model");
dic.put(UPnPDevice.MODEL_NUMBER,"1.0");
dic.put(UPnPDevice.MODEL_URL,"http://javahispano.org");
dic.put(UPnPDevice.SERIAL_NUMBER,"1234567893");
dic.put(UPnPDevice.TYPE,"urn:schemas-upnp-
org:javahispano:sensorreader:1");
dic.put(UPnPDevice.UDN,DEVICE_ID);
dic.put(UPnPDevice.UPC,"12134567893");
```

Antes de empezar a ver la clase TemperaturaService, tendríamos que añadir a la clase anterior mecanismos para generar eventos UPnP en la red. Hasta ahora no hemos hablado de eventos UPnP, solo de acciones (UPnPAction) y variables de estado (UPnPStateVariable). Lo que se pretende conseguir con este tipo de eventos es que cada vez el sensor de temperatura varíe, envie una actualización del valor a través de UPnP o en su defecto que cada X segundos se produzca un evento UPnP recordando el valor del sensor de temperatura.

Para generar un evento UPnP utilizaremos el bundle descargado anteriormente Apache Felix UPnP Extra. Ese bundle entre otras utilidades nos brinda la posibilidad de crear notificadores de eventos UPnP así como listenners de estos eventos.
Así pues, modificaremos el constructor de la clase SensorTemperaturaDevice y añadiremos dos nuevos métodos:

```
import org.apache.felix.upnp.extra.util.UPnPEventNotifier;
...
public static UPnPEventNotifier notifier = null;
   private BundleContext context;

   public SensorTemperaturaDevice(BundleContext context){
         this.context = context;
         temperaturaService =  new TemperaturaService();
         services = new UPnPService[]{temperaturaService};
         setupDeviceProperties();
         buildEventNotifyer();
   }

   private void buildEventNotifyer() {
          notifier = new UPnPEventNotifier(context,this,temperaturaService);
   }
   public void update(Double newTemp) {
   UPnPStateVariable variable =
temperaturaService.getStateVariable("Temperatura");
   notifier.propertyChange(new
PropertyChangeEvent(variable,"Temperatura",null,newTemp));
   }
```

Listado 3.2.2.1 -3 SensorTemperaturaDevice II

En el listado anterior hemos creado una instancia de UPnPEventNotifier para el servicio TemperaturaService que veremos mas adelante. También hemos creado un nuevo método **update** que recibe como parámetro un nuevo valor para la temperatura. Este método será invocado desde el listenner del puerto COM (La clase COMSensor que veremos mas adelante). Este método update actualizará la variable de estado TemperaturaStateVariable, generando un evento en la red UPnP.

Siguiendo el orden de jerarquía de clases, a continuación veremos la clase TemperaturaService que implementará a la interfaz UPnPService. Implementar esta interfaz obliga a escribir los siguientes métodos:

- *UPnPAction* **getAction**(java.lang.String name)

  Localiza una acción específica dado el nombre recibido como parámetro.
  Recordemos que nuestro dispositivo va a tener tres acciones: On, Off y GetTemp.

- *UPnPAction[]* **getActions**()

  Lista todas las acciones asociadas al servicio.

- *java.lang.String* **getId**()

  Retorna el id identificativos del servicio

- *UPnPStateVariable* **getStateVariable**(java.lang.String name)

  Retorna la variable de estado asociada al servicio e identificada con el nombre recibido como parámetro.

- *UPnPStateVariable[]* **getStateVariables**()

  Retorna todas las variables de estado asociadas al servicio. En nuestro caso solo tendremos una única variable de estado TemperaturaStateVariable.

- *java.lang.String* **getType**()

  Retorna el campo serviceType de la descripcion del servicio UPnP.

- *java.lang.String* **getVersion**()

  Retorna la version del servicio.

```
import org.osgi.service.upnp.UPnPAction;
import org.osgi.service.upnp.UPnPService;
import org.osgi.service.upnp.UPnPStateVariable;

public class TemperaturaService implements UPnPService{

   final static private String SERVICE_ID = "urn:schemas-upnp-
org:serviceId:temperatura:1";
   final static private String SERVICE_TYPE = "urn:schemas-upnp-
org:service:temperatura:"
         + TemperaturaService.VERSION;
   final static private String VERSION = "1";

   private UPnPStateVariable temperatura;
   private UPnPStateVariable[] states;
   private HashMap actions = new HashMap();

   public TemperaturaService(){

         temperatura = new TemperaturaStateVariable();
         this.states = new UPnPStateVariable[]{temperatura};

         UPnPAction setOff= new SetOffAction();
         UPnPAction setOn= new SetOnAction();
```

```
        UPnPAction getTemp= new
GetTempAction((TemperaturaStateVariable)temperatura);
        actions.put(setOff.getName(),setOff);
        actions.put(setOn.getName(),setOn);
        actions.put(getTemp.getName(),getTemp);
    }

    public String getId() {
        return SERVICE_ID;
    }

    public String getType() {
        return SERVICE_TYPE;
    }

    public String getVersion() {
        return VERSION;
    }

    public UPnPAction getAction(String name) {
        return (UPnPAction)actions.get(name);
    }

    public UPnPAction[] getActions() {
        return (UPnPAction[])(actions.values()).toArray(new UPnPAction[]{});
    }

    public UPnPStateVariable[] getStateVariables() {
        return states;
    }

    public UPnPStateVariable getStateVariable(String name) {
                return temperatura;
    }

}
```

Listado 3.2.2.1 -4 TemperaturaService

En la clase del listado anterior lo que hacemos es asignarle un ID , un tipo y un numero de version al servicio UPnP. En el constructor de la clase aprovecharemos para definir las variables de estado (TemperaturaStateVariable) y las acciones (SetOffAction, SetOnAction y GetTempAction).

La siguiente clase a revisar es TemperaturaStateVariable que implementará la interface UPnPLocalStateVariable. Implementar dicha interface implica escribir los siguientes métodos:

- *java.lang.String[]* **getAllowedValues**()

  Retorna los valores permitidos si están definidos.

- *java.lang.Object* **getDefaultValue**()

  Retorna el valor por defecto si esta definido

- *java.lang.Class* **getJavaDataType**()

  Retorna la clase java asociada con el tipo de dato UPnP para esa variable de estado.

- *java.lang.Number* **getMaximum**()

  Retorna el valor máximo si esta definido.

- *java.lang.Number* **getMinimum**()

  Retorna el valor mínimo si esta definido

- *java.lang.String* **getName**()

  Retorna el nombre de la variable

- *java.lang.Number* **getStep**()

  Retorna el tamaño del incremento de la variable si esta definido.

- *java.lang.String* **getUPnPDataType**()

  Retorna el tipo de dato UPnP de la variable de estado. Estos tipos de datos están definidos como constantes de la interfaz UPnPStateVariable.

- *boolean* **sendsEvents**()

  Indica si la variable de estado puede ser usada para generar eventos

```
import org.osgi.service.upnp.UPnPLocalStateVariable;

public class TemperaturaStateVariable implements UPnPLocalStateVariable{

    final private String NAME = "Temperatura";
    final private Double DEFAULT_VALUE =0.0;

    public String[] getAllowedValues() {
            return null;
    }
    public Object getDefaultValue() {
            return DEFAULT_VALUE;
    }
    public Class getJavaDataType() {
            return Double.class;
    }
    public Number getMaximum() {
            return null;
    }
    public Number getMinimum() {
            return null;
    }
    public String getName() {
            return NAME;
    }
    public Number getStep() {
            return null;
    }
    public String getUPnPDataType() {
            return this.TYPE_NUMBER;
    }
    public boolean sendsEvents() {
            return true;
    }
    public Double getTemp(){
            return COMSensor.getInstance().getTemp();
    }

    public Object getCurrentValue() {
            return COMSensor.getInstance().getTemp();
    }
}
```

Listado 3.2.2.1 -5 TemperaturaStateVariable

Además de implementar los métodos que obliga la interfaz, hemos implementado el método getTemp que invocará a la clase COMSensor para obtener el último valor proporcionado por el dispositivo sensor.

Finalmente solo nos quedan ver las acciones del servicio UPnP, que como hemos visto en la clase TemperaturaService. En nuestro caso tenemos tres acciones:

- **SetOffAction:** Acción para apagar la calefacción. Invocará a la clase controladora del puerto COM para enviar una señal al microcontrolador del dispositivo sensor y apagar la calefacción.
- **SetOnAction:** Acción para encender la calefacción. El mecanismo para encender la calefacción será igual que en el caso anterior.
- **GetTempAction:** Acción que retorna la temperatura actual. Esta acción se realizará gracias a la ayuda de la clase que controla el puerto COM.

Estas tres clases implementarán todas la misma interfaz UPnPAction. Implementar esta interfaz, implica escribir los siguientes métodos:

- *java.lang.String[]* ***getInputArgumentNames()***

  Lista todos los argumentos de entrada de la acción. En nuestro caso las acciones no recibirán ningún parámetro de entrada.

- *java.lang.String* ***getName()***

  Retorna el nombre de la acción

- *java.lang.String[]* ***getOutputArgumentNames()***

  Lista todos los argumentos de salida de la acción.

- *java.lang.String* ***getReturnArgumentName()***

  Retorna el nombre del parámetro de retorno.

- *UPnPStateVariable* ***getStateVariable****(java.lang.String argumentName)*

  Busca la variable de estado asociada al nombre pasado como parámetro.

- *java.util.Dictionary* ***invoke****(java.util.Dictionary args)*

  Invoca a la accion.

A continuación veremos la clase SetOffAction:

```
package org.javahispano.sensorreader;

import java.util.Dictionary;
import java.util.Hashtable;

import org.osgi.service.upnp.UPnPAction;
import org.osgi.service.upnp.UPnPStateVariable;

public class SetOffAction implements UPnPAction{

   final private String NAME = "Apagar Calefaccion";
   final private String RESULT_STATUS = "Apagado";

   public String[] getInputArgumentNames() {
         return null;
   }
   public String getName() {
         return NAME;
   }
   public String[] getOutputArgumentNames() {
         return null;
   }
   public String getReturnArgumentName() {
         return null;
   }
   public UPnPStateVariable getStateVariable(String arg0) {
         return null;
   }

   public Dictionary invoke(Dictionary arg0) throws Exception {
         COMSensor.getInstance().setOff(true);
         Hashtable result = new Hashtable();
        //Siempre retornamos true
         result.put(RESULT_STATUS,true);
         return result;
   }
}
```

Listado 3.2.2.1 -6 SetOffAction

Ignoraremos la clase SetOnAction ya que será exactamente igual que la anterior, aunque siempre podréis revisarla en los ejemplos adjuntos a este tutorial.

Finalmente veremos la acción GetTempAction:

```
import org.osgi.service.upnp.UPnPAction;
import org.osgi.service.upnp.UPnPStateVariable;

public class GetTempAction implements UPnPAction{

   final private String NAME = "GetTemp";
   final private String RESULT_STATUS = "CurrentTemp";
   final private String[] OUT_ARG_NAMES = new String[]{RESULT_STATUS};
   private TemperaturaStateVariable temp;

   public GetTempAction(TemperaturaStateVariable temp){
         this.temp = temp;
   }
    public String getName() {
         return NAME;
   }
   public String getReturnArgumentName() {
         return null;
   }
   public String[] getInputArgumentNames() {
         return null;
   }
   public String[] getOutputArgumentNames() {
         return OUT_ARG_NAMES;
   }
   public UPnPStateVariable getStateVariable(String argumentName) {
         return temp;
   }
   public Dictionary invoke(Dictionary args) throws Exception {
         Double value = temp.getTemp();
         Hashtable result = new Hashtable();
        //Retornamos en el Hashtable la temperatura actual
         result.put(RESULT_STATUS,value);
         return result;
   }
}
```

Listado 3.2.2.1 -7 GetTempAction

3.2.2.2 Testear un dispositivo UPnP

Para probar el bundle construido en el apartado anterior, he utilizado el módulo Apache Felix UPnP Tester que me he bajado ya compilado de la página de descargas de Apache Felix.

Para este entorno de pruebas, he aprovechado también para desplegar los bundles de ejemplo TV y Clock de Apache Felix. Estos dos bundles están adjuntos a los ejemplos anexos a este tutorial, pero también los podréis descargar del repositorio de Apache Felix.

Para arrancar la plataforma, el config.ini debería quedar similar al siguiente:

```
osgi.clean=true
eclipse.ignoreApp=true

org.osgi.framework.bootdelegation=javax.jms
osgi.bundles=../serial/plugins/rxtxcomm_api-2.1.7.jar@start, \
../serial/plugins/rxtxcomm-linux-pc-2.1.7.jar, \
org.eclipse.osgi.services_3.1.200.v20071203.jar@start,\
org.eclipse.equinox.util_1.0.0.v20080414.jar@start,\
org.eclipse.osgi.util_3.1.300.v20080303.jar@start,\
../upnp/org.apache.felix.upnp.basedriver-0.8.0.jar@start,\
../upnp/plugins/org.apache.felix.upnp.extra_0.4.0.jar@start,\
../upnp/plugins/org.javahispano.sensorreader_1.0.0.jar@start,\
../upnp/plugins/org.apache.felix.upnp.tester-0.4.0.jar@start,\
../upnp/plugins/tv_1.0.0.jar@start,\
../upnp/plugins/clock_1.0.0.jar@start,\
```

Listado 3.2.2.2 -1 config.ini

Al arrancar se abrirán tres ventanas diferentes:

- **Clock:** Simula un reloj digital que enviará notificaciones UPnP a través de la Red.
- **TV:** Simula la pantalla de un televisión. Leerá los eventos del reloj para pintar por pantalla la hora.
- **UPnP Tester:** Desde esta herramienta podremos interactuar con todos los dispositivos UPnP de la Red.

Figura 3.2.2.2 -1 Reloj UPnP

Figura 3.2.2.2 -2 TV que imprime la hora del reloj por pantalla

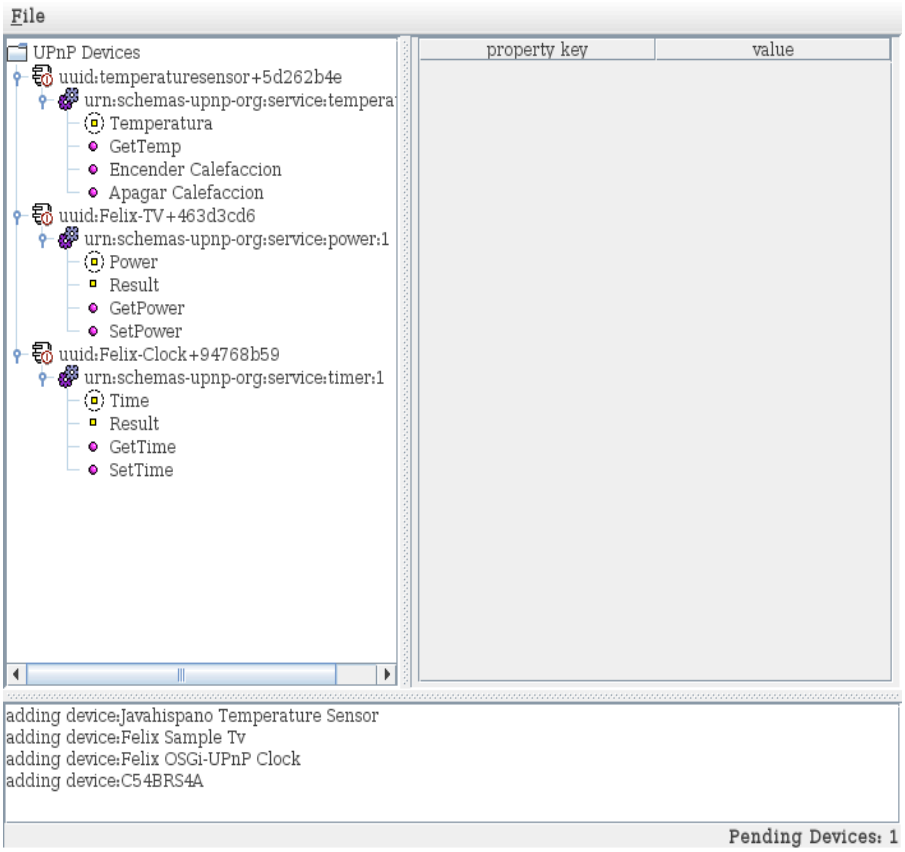


Figura 3.2.2.2 -3 Pantalla del bundle UPnP Tester

En la pantalla del tester, podemos ver que se han reconocido todos los dispositivos UPnP de la red. Este bundle lo podríamos ejecutar en otro equipo de la red y veríamos exactamente lo mismo.

Los dispositivos aparecen en una estructura tipo árbol, identificados por su UID, debajo aparecerán sus servicios y dentro de los servicios las acciones y variables de estado. En la siguiente figura podemos ver la correspondencia con sus iconos:

- Root Device
- Embedded Device
- Service
- Action
- State Variable
- Evented State Variable
- Subscribed State Variable

Desde esta pantalla podremos "jugar" a controlar los dispositivos. Por ejemplo podremos parar la "emisión" del bundle TV, desplegando las acciones del servicio, seleccionando la acción SetPower y pinchando en el botón "do Action".

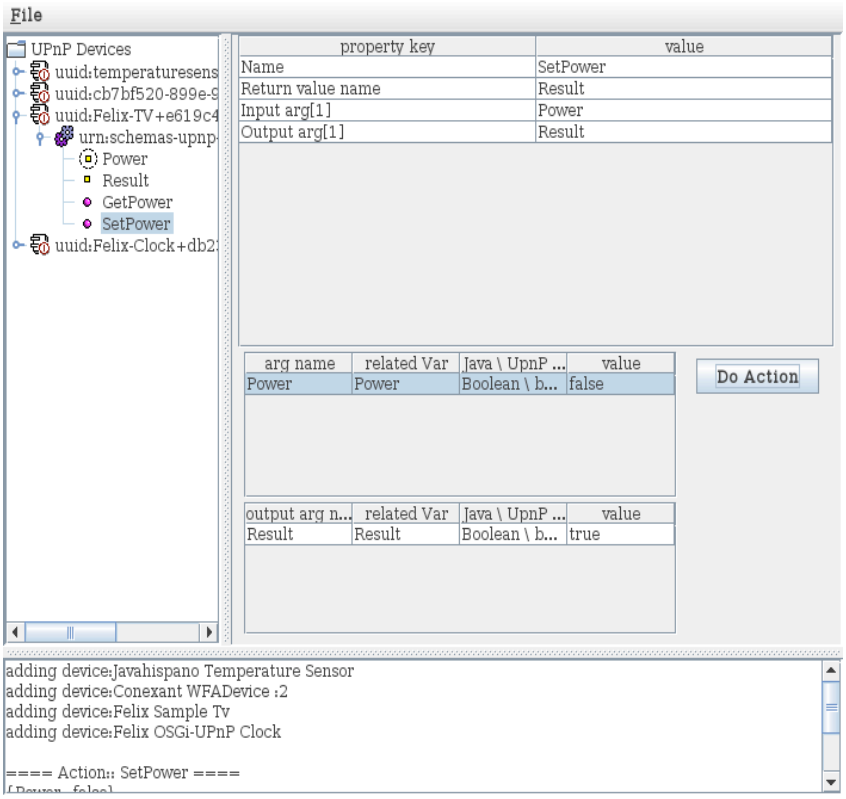


Figura 3.2.2.2 -4 Pantalla del bundle UPnP Tester

Esta acción recibe un parámetro que por defecto es valor a "false". Con este valor a false apagaremos la "emisión" de la TV.

Figura 3.2.2.2 -5 Tv Apagada

Para volver a encender la TV, tendremos que volver a ejecutar la accion SetPower, pero esta vez pasándole como parámetro “true”:

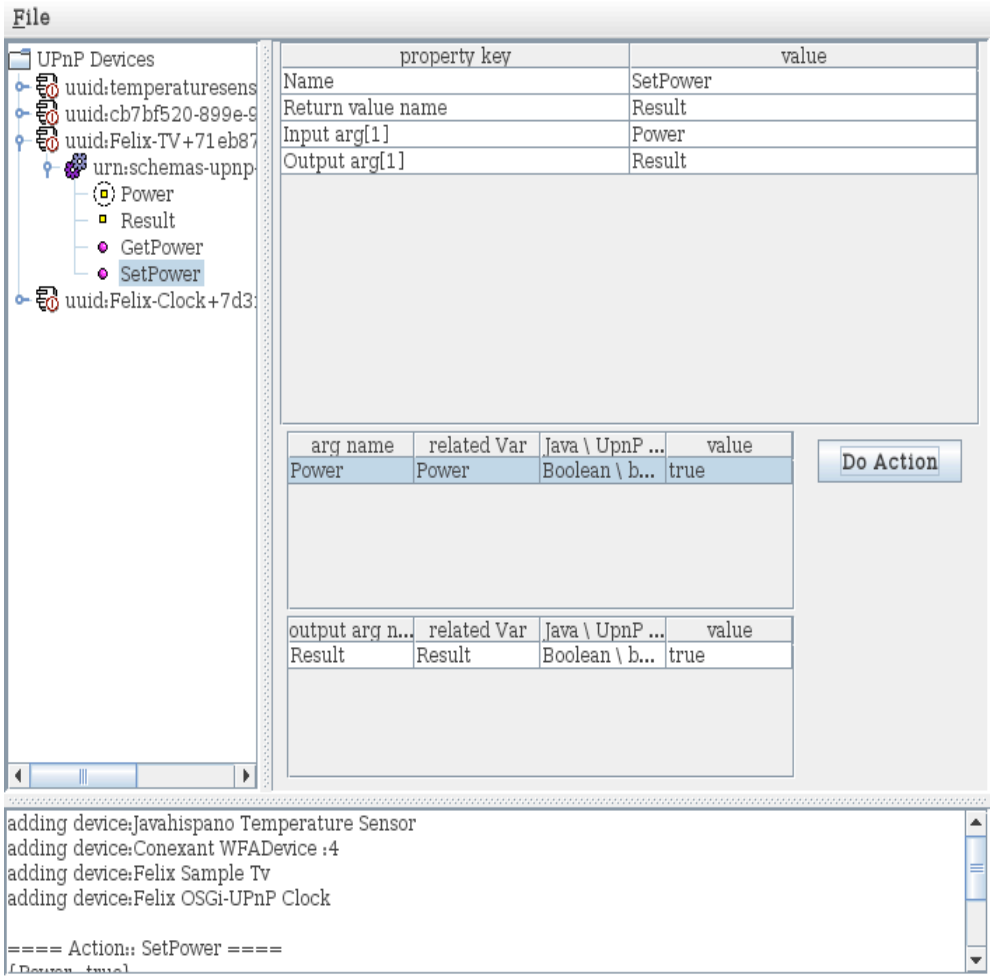


Figura 3.2.2.2 -6 Pantalla del bundle UPnP Tester

En el caso de nuestro sensor de temperatura, tenemos dos acciones “Apagar Calefacción” (SetOffAction) y “Encender Calefacción” (SetOnAction) para apagar o encender la calefacción. Ambas acciones no reciben parámetros. Si lo que queremos es obtener la temperatura actual invocaremos la acción GetTemp (GetTemAction):

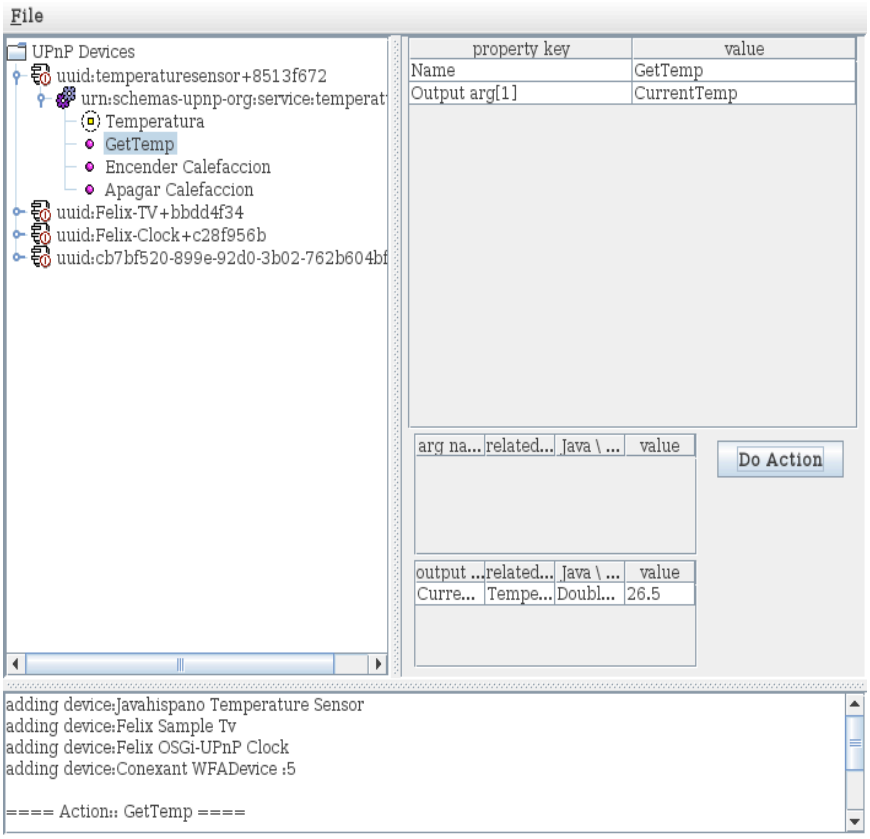


Figura 3.2.2.2 -7 Pantalla del bundle UPnP Tester

3.2.2.3 Escuchando eventos UPnP

Hasta ahora hemos testeado el bundle UPnP org.javahispano.sensorreader, mediante la herramienta UPnP Tester, ahora vamos a escribir el bundle org.javahispano.sensortest para escuchar los eventos UPnP producidos por el sensorreader. Este bundle que vamos a construir podría ir ubicado en cualquier lugar de la red, en nuestro caso, lo ubicaremos en el PC de sobremesa al cual en la última entrega de este tutorial le instalaremos un servidor de aplicaciones J2EE basado en OSGI.

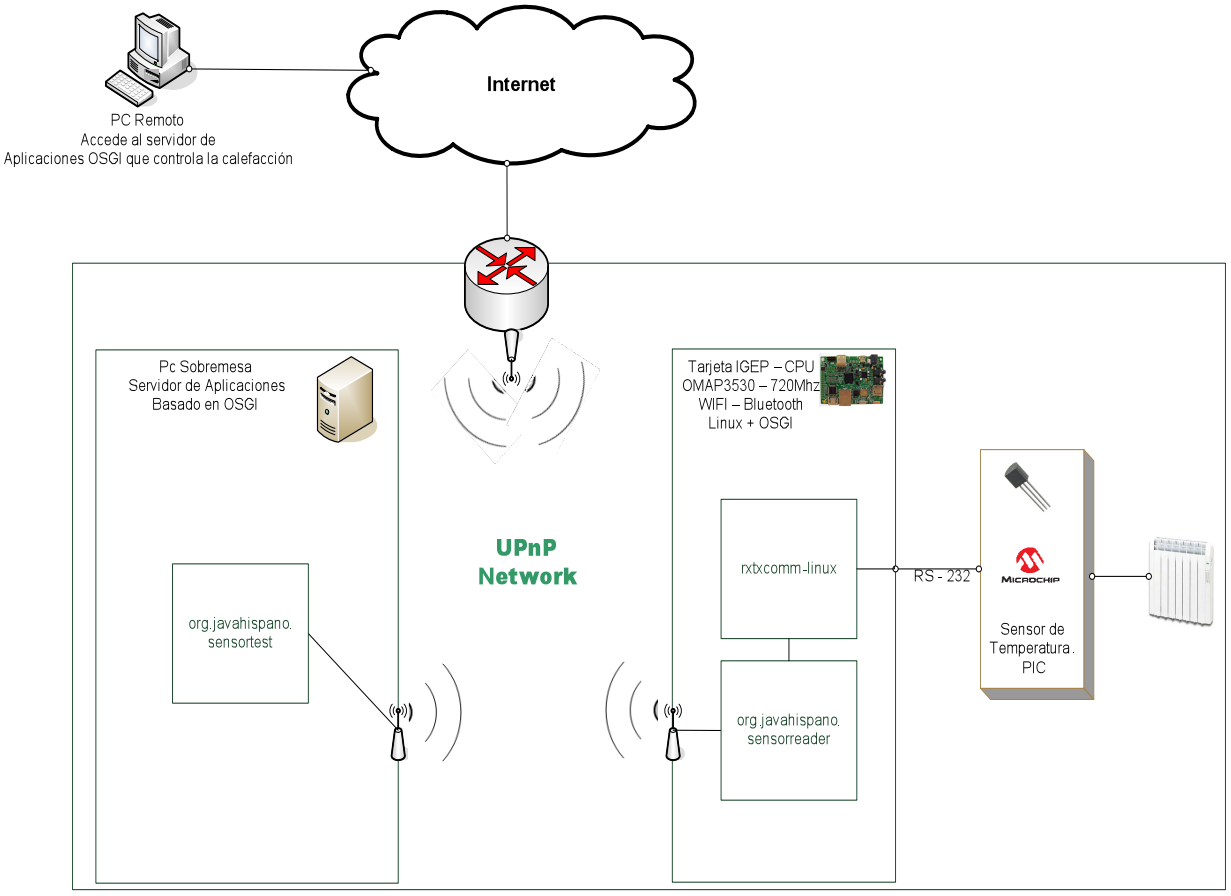


Figura 3.2.2.3 -1 Arquitectura Ejemplo

Para subscribirnos a los eventos utilizaremos la clase UPnPSubscriber del bundle UPnP Extra de Felix.
Nuestro bundle org.javahispano.sensortest solo tendrá una única clase que implementará a las interfaces BundleActivator y UPnPEventListener:

```
import org.apache.felix.upnp.extra.util.UPnPSubscriber;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import org.osgi.service.upnp.UPnPEventListener;

public class Activator implements BundleActivator,UPnPEventListener {

   //Dispositivo UPnP al que queremos subscribirnos
    private final static String SENSOR_DEVICE_TYPE = "urn:schemas-upnp-
org:javahispano:sensorreader:1";
```

```
  //Servicio UPnP al que queremos subscribirnos
   final static private String TEMPERATURE_SERVICE_TYPE = "urn:schemas-
upnp-org:service:temperatura:1";

   private UPnPSubscriber subscriber;
   private BundleContext context;


   public void start(BundleContext context) throws Exception {
         this.context = context;
         doSubscribe();
   }

   public void stop(BundleContext context) throws Exception {
   }

   public void notifyUPnPEvent(String deviceId, String serviceId, Dictionary
events) {
         Double temp = (Double) events.get("Temperatura");
         System.out.println("SensorTest::Notificacion UPnP:temperatura= " +
temp);
   }


   public void doSubscribe()
   {
         subscriber = new UPnPSubscriber(context,this);
         subscriber.subscribeEveryServiceType(SENSOR_DEVICE_TYPE,
TEMPERATURE_SERVICE_TYPE);

   }

   public void undoSubscribe(){
         subscriber.unsubscribeAll();
   }

}
```

Listado 3.2.2.3 -1 Activator

Para subscribirnos utilizamos los UID del dispositivo(SENSOR_DEVICE_TYPE) y del servicio (TEMPERATURE_SERVICE_TYPE).
El implementar la interfaz UPnPEventListener nos obliga a escribir el método notifyUPnPEvent, que será la función que se encargue de recibir y tratar el evento UPnP.
En el método doSubscribe(), registraremos la clase Activator como listenner UPnP:

subscriber = new UPnPSubscriber(context,this);

4. Conclusión

A lo largo de esta segunda entrega hemos repasado la forma en que OSGI carga las clases y hemos profundizado un poco más en la manera de trabajar con servicios y componente OSGI. También hemos visto algunas de las especificaciones más interesantes de OSGI. En los casos prácticos nos hemos centrado principalmente en como se interconectan entre si los bundles bajo una misma plataforma, aunque también hemos visto como conectar estos bundles con el mundo exterior (Servicios Web y protocolo UPnP).

Para quien quiera profundizar mas en las especificaciones OSGI, aconsejo acudir a la documentación oficial, ya que en este tutorial no hemos entrado en los verdaderos entresijos de OSGI.

Todavía nos quedan por ver algunos servicios básicos de la plataforma, como pueden ser la gestión de logs y la gestión de la configuración, que veremos en la última entrega de esta serie de tutoriales. Además en la última entrega intentaremos centrarnos en entornos Web sobre la plataforma OSGI, así como el control remoto de la misma y la integración con OSGI.

ANEXO I: Índice de Figuras

ANEXO II: Indice de Listado Codigo Fuente

ANEXO III: Microcontrolador – Sensor de Temperatura

La mayoría de los ejercicios descritos en este tutorial, se basan en un sensor de temperatura conectado a través de un puerto serie al dispositivo donde se encuentra desplegada la plataforma OSGI.

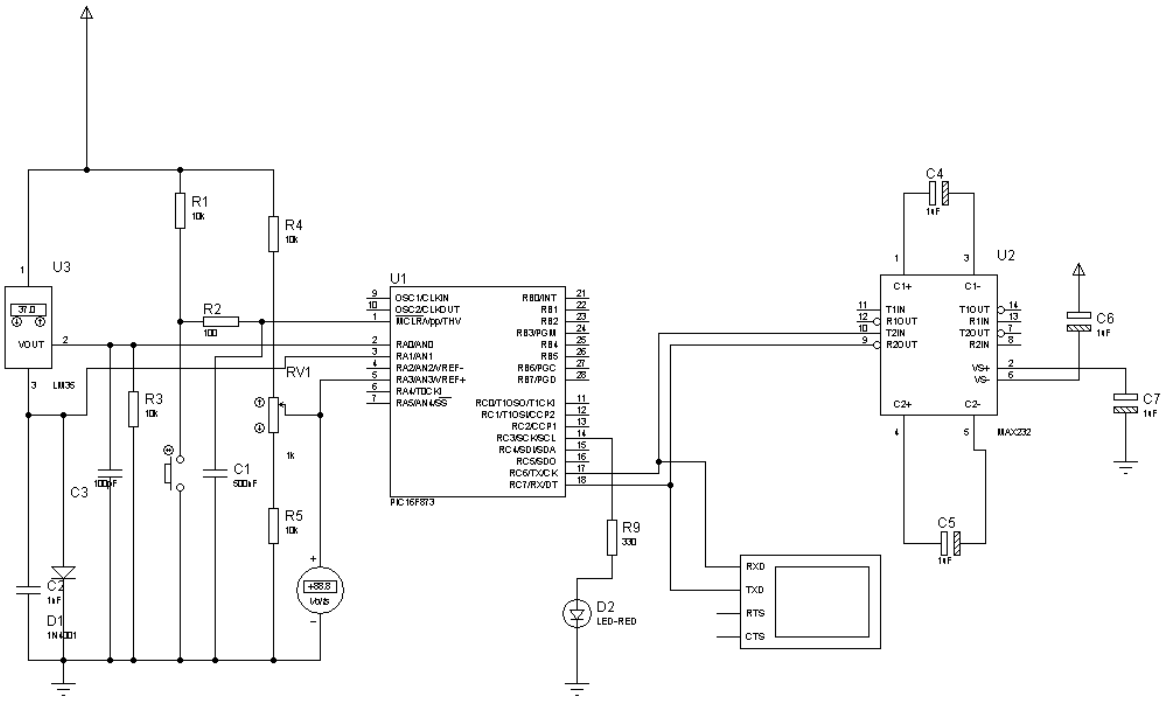
Este sensor de temperatura se basa en un microcontrolador PIC 16F873. Los PIC son una familia de microcontroladores tipo RISC fabricados por Microchip Technology Inc. y derivados del PIC1650, originalmente desarrollado por la división de microelectrónica de General Instrument.

Gracias a mi "aita", que me ha construido este pequeño sensor de temperatura, he podido hacer un poco mas práctico este tutorial, además de aprovechar la ocasión de profundizar en el mundillo de la programación de microcontroladores.

El sensor de temperatura esta basado en la información sacada de las siguientes paginas:

- http://www.seta43.netau.net/electro.html
- http://www.msebilbao.com/

En la siguiente figura podréis ver el esquema del circuito:



Este circuito a sido diseñado usando la herramienta Proteus, en los ejercicios adjuntos a este tutorial, podréis encontrar los ficheros Proteus asociados a este proyecto del sensor de temperatura, así como el código fuente ASM con el que programaremos el microcontrolador.

ANEXO IV: Practicas adjuntas al tutorial

Se adjuntan todos los ejemplos completos explicados a lo largo del manual.
Se trata de proyectos Eclipse, debidamente numerados de acuerdo con el indice de este tutorial.