OSGI

Roberto Montero Miguel

INDICE

Modularidad en Java

Actualmente, la tendencia del desarrollo de aplicaciones camina hacia la construcción de aplicaciones más complejas, pero con arquitecturas modulares que nos permitan gestionar sus dependencias, así como ampliar la aplicación mediante la construcción de nuevos módulos. Un claro ejemplo de éxito de este tipo de aplicaciones, es el proyecto eclipse, que proporciona mecanismos para ampliar y personalizar fácilmente la plataforma mediante el desarrollo de plugins. Otro caso de éxito, esta vez no basado en java, es la gestión de dependencias que realizan algunos sistemas operativos como UBUNTU. Con Ubuntu y el gestor de paquetes Synaptic, podremos instalar fácilmente nuevos programas, delegando en el Synaptic la gestión de dependencias. Synaptic descargará de los repositorios los paquetes necesarios para la ejecución del nuevo programa que queremos instalar.

Desgraciadamente la plataforma Java 6, en si misma tiene escasos mecanismos de modularización, limitada básicamente a la gestión de paquetes, clases y métodos. Es cierto, que existen algunos herramientas como Spring o algunos patrones de diseño, que nos pueden dar la sensación de que estamos construyendo una aplicación modular, ya que por ejemplo Spring realiza inyección dependencias que permite un acoplamiento débil de las clases mediante el uso de interfaces. Digo que "Spring nos dada la sensación de estar construyendo aplicaciones modulares", por que por ejemplo, en el caso de las aplicaciones WEB, el uso de Spring nos permite dividir nuestro código en diferentes componentes, pero finalmente los desplegamos en el servidor bajo un único modulo monolítico (WAR).

En teoría, una aplicación totalmente modular, nos debería permitir sustituir un modulo de dicha aplicación por otro sin afectar al resto de los módulos. Por ejemplo, que yo sustituya un módulo de la aplicación, no me debería de obligar a recompilar el resto de módulos. Otro ejemplo, en una aplicación Web, que yo sustituya un módulo, no me debería obligar a parar la aplicación por completo, ni siquiera deberían de dejar de dar servicio los módulos que no dependen del módulo sustituido.

Podemos asegurar que una aplicación modular nos proporcionaría

- **Facilidad al cambio**: Si cada módulo de una aplicación sólo se conoce a través de su interfaz (y no por su aplicación interna), entonces es fácil de cambiar una módulo con otro.
- **Facilidad de Comprensión**: Si una aplicación esta compuesta por módulos perfectamente definidos y delimitados, resulta mas sencillo la comprensión y el estudio de cada módulo de forma individual, y por lo tanto mas fácil la compresión de la totalidad de la aplicación.

- **Desarrollo en paralelo**: Los módulos pueden ser desarrollados casi independientemente unos de otros, haciendo posible que los equipos de desarrollo de dividir las tareas.
- **Aplicaciones fáciles de testear**: Podremos probar cada módulo de forma independiente.
- **Reutilización**: Al tratarse de módulos independientes, podremos usar fácilmente las funcionalidades de un módulo de una aplicación en otro aplicativo.

Para escribir estas primeras líneas del tutorial, me he apoyado en el libro "Modular Java: Creating Flexible Applications with OSGI and Spring", libro totalmente recomendado para aquel que quiera indagar un poco mas en sistemas modulares basados en java como OSGI (Claro está, lo recomiendo, pero siempre después de haberse leído mi tutorial ;-D).

1 Introducción a OSGI

OSGI (**O**pen **S**ervices **G**ateway **I**nitiative), podríamos definirlo como un sistema (o framework) modular para Java que establece las formas de crear módulos y la manera en que estos interactuaran entre sí en tiempo de ejecución. OSGI intenta solventar los problemas del tradicional "classloader" de la máquina virtual y de los servidores de aplicaciones Java. En OSGI, cada módulo tiene su propio classpath separado del resto de classpath de los demás módulos.

Este framework proporciona a los desarrolladores un entorno orientado a servicios y basado en componentes, ofreciendo estándares para manejar los ciclos de vida del software.

La arquitectura OSGI se divide en capas, tal y como se representan en la figura 1.1, las cuales se estudiarán detalladamente a lo largo de este documento.

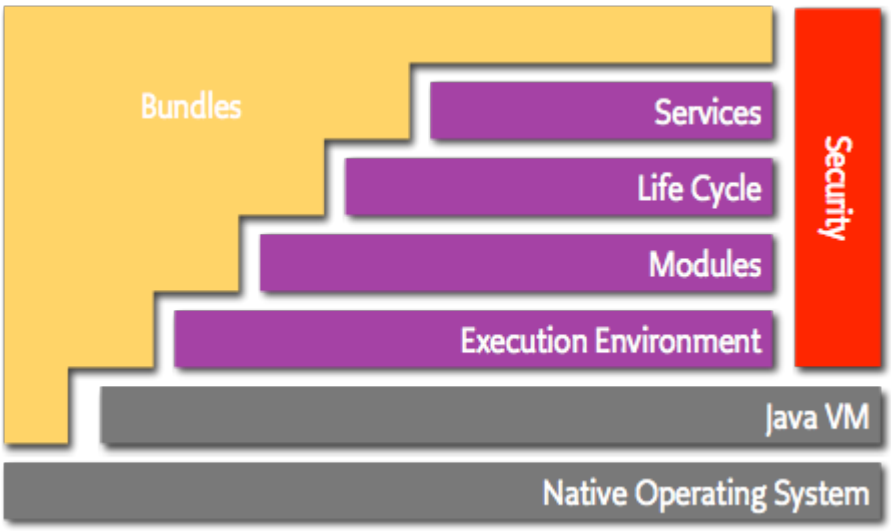

Figura 1.1- Arquitectura OSGI

OSGI fue creado en Marzo de 1999, su objetivo es definir las especificaciones abiertas de software que permitan diseñar plataformas compatibles que puedan proporcionar múltiples servicios. Fue pensado principalmente para su aplicación en redes domésticas, con clara orientación a ser introducido en pequeños y grandes dispositivos como por ejemplo, set-top boxes, cable módems, electrodomésticos, PCs, coches, teléfonos móviles ..

La OSGi Alliance es un consorcio de empresas tecnológicas a nivel mundial que trata de asegurar la interoperabilidad de las aplicaciones y servicios basados en esta plataforma Entre las empresas que componen este consorcio, podemos encontrar compañías de diversa índole: automoción, aeronáutica, fabricantes de electrodomésticos, telecomunicaciones, fabricantes de teléfonos... Algunos ejemplos de miembros: Motorola, Nokia, Mitsubishi Electric Corporation, Vodafone Group Services, LinkedIn, LG Electronics...

La alianza proporciona las especificaciones, las implementaciones de referencia, las suites de prueba y la certificación.

La adopción de la plataforma reduce el tiempo de comercialización y los costes de desarrollo ya que permite la integración con módulos pre-construidos. Reduce los costes de mantenimiento y proporciona capacidades de actualización de los diferentes módulos y servicios.

El consorcio está dividido en diferentes grupos de trabajo, con distintos objetivos dentro de la especificación:

- **Expert Groups:**
  - **Core Platform (CPEG)** : Define las especificaciones básicas de la plataforma. El CPEG se centra en los componentes que forman el entorno de ejecución de OSGi y los servicios fundamentales para todos los entornos de estas características.
  - **Mobile (MEG) :** Define los requisitos y especificaciones para adaptar y ampliar la plataforma de servicios OSGi para dispositivos móviles (aquellos dispositivos capaces de conectarse a redes inalámbricas).
  - **Vehicle (VEG)** : Se centra en la adaptación y ampliación de la plataforma a los entornos empotrados en los vehículos.
  - **Enterprise (EEG)** : Es un nuevo grupo de expertos que definirá los requisitos y especificaciones técnicas para adaptar y ampliar la plataforma de servicios OSGi al ámbito del software empresarial.
  - **Residential (REG)** :Define los requisitos y especificaciones para adaptar y ampliar la Plataforma de Servicios OSGi, al ámbito de las plataformas residenciales.
- **Marketing Working Groups:** Fue creada para ayudar a promover la Alianza OSGi.
- **Market Requirements Working Groups:** Crean documentos de requisitos que describen las principales exigencias del mercado.

1.1 Introducción al documento

A lo largo de este documento recorreremos los diferentes ámbitos de ejecución de la especificación OSGI, veremos algunas de las distintas plataformas existentes en el mercado que cumplen con esta especificación y aprenderemos varias formas de desarrollar sobre estas. Posteriormente veremos algunos frameworks que extienden OSGI aumentando aún más las capacidades de esta plataforma. Analizaremos como crear diferentes aplicaciones, pasando desde una sencilla aplicación con escasas líneas de código, hasta grandes aplicaciones empresariales, incluyendo entornos web, explotando Bases de Datos y creando mecanismos de comunicación entre aplicaciones (SOA, JMS, JMX....). Así pues, veremos de un vistazo rápido algunos proyectos de software libre elaborados sobre osgi, quizás con menos transcendencia, pero con mucho interés tecnológico.

A pesar de que este articulo, no pretende ser un tutorial sobre el API de OSGI, nos vemos obligados a realizar una introducción sobre los principios del API, para poder llegar a entender posteriormente el resto de implementaciones sobre OSGI.

Por lo tanto este tutorial lo dividiremos en cuatro secciones:

- Las dos secciones iniciales formarán parte de esta primera entrega, y principalmente consistirá en los principios básicos de OSGI, sus principales implementaciones y las herramientas básicas para el desarrollo de código para esta plataforma. Así pues las dos primeras secciones constarán de:

    - **Especificación OSGI**: Documentación entresacada de las especificaciones oficiales de OSGI. Se trata de una parte teórica, donde se manejan muchos conceptos, sugerimos el apoyo en las especificaciones oficiales de OSGI, ya que hay algunos conceptos "menos importantes" que hemos podido ignorar. No os preocupéis si no lo entendéis todo en esta primera parte, ya que se trata de definiciones breves de los conceptos, que podemos entender mejor cuando avancemos hasta la parte práctica de este tutorial.

    - **Implementaciones OSGI y entornos de desarrollo**: Veremos las distintas implementaciones OSGI existentes en el mercado. También veremos las diferentes opciones disponibles para crearnos nuestro entorno de desarrollo.

Las últimas dos secciones se presentarán en dos entregas diferentes y consistirán en:

- **Especificación OSGI, explicada mediante ejemplos**: Toda la teoría explicada en el primer apartado, la veremos en este a través de ejemplos prácticos.

- **Aplicaciones OSGI**: Veremos cómo crearnos aplicaciones web basadas en OSGI, integraremos nuestras aplicaciones OSGI con otros frameworks como hibernate, Spring.. Utilizaremos gestiones remotas del entorno, usaremos tecnologías como SOAP. Analizaremos servidores de aplicaciones basados en OSGI y estudiaremos la extensión de Spring para OSGI.

Estas dos últimas entregas, intentaremos orientarlas en la simulación de un entorno real, con el fin de que los dos últimos artículos sean más amenos para el lector.

El entorno real presentado consistirá en controlar remotamente la calefacción de una habitación. Por ejemplo, podremos controlar la temperatura de nuestra casa desde cualquier punto con conexión a Internet. La siguiente figura intenta plasmar una foto de nuestro sistema:

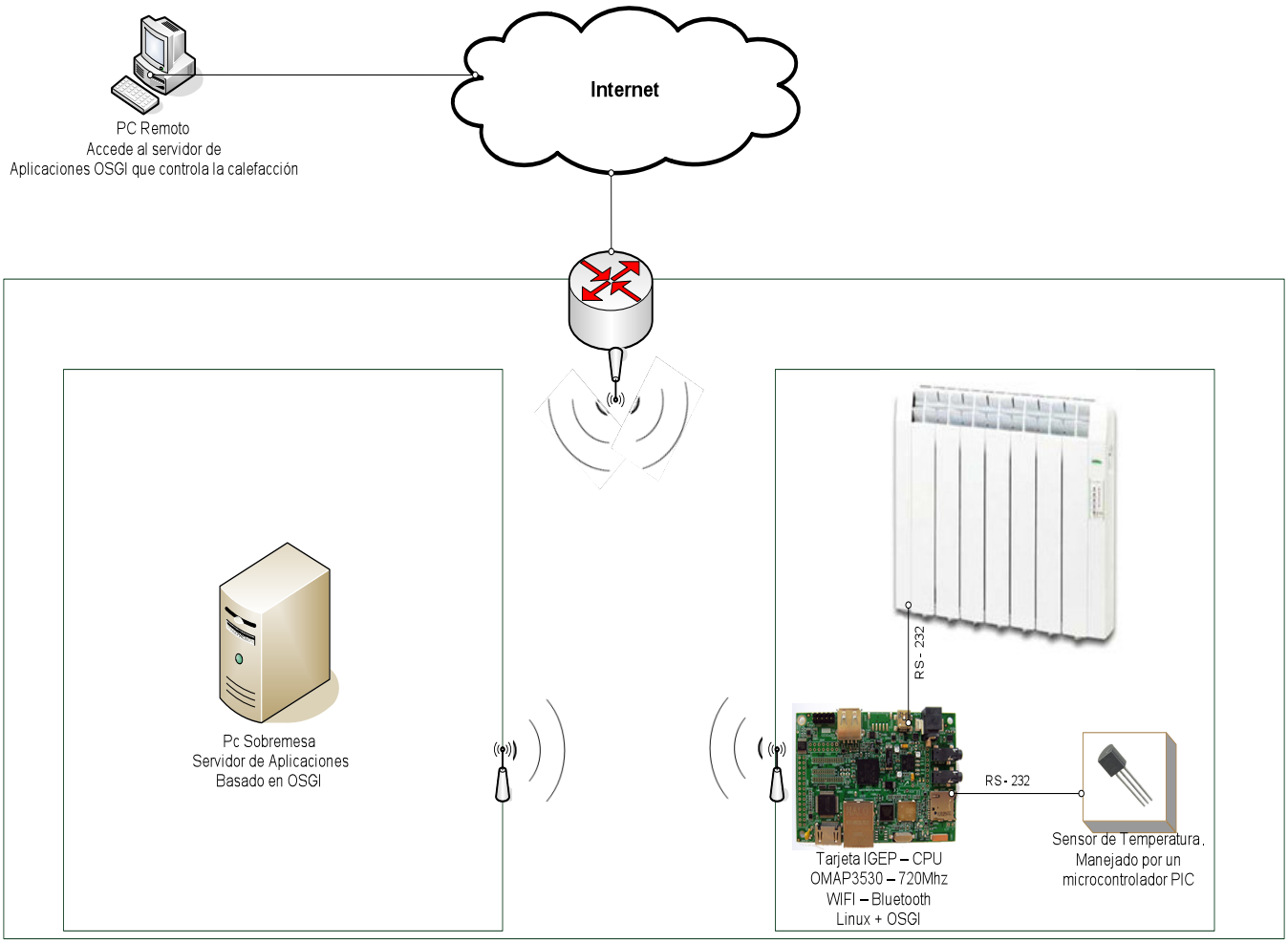

Figura 1.2- Sistema Controlador de Calefacción

En la figura anterior encontramos dos sistemas:

- **Controlador de la calefacción:**

Estará compuesto por una tarjeta IGEP con un sistema operativo Linux. Esta tarjeta es un mini-pc del tamaño de una tarjeta de crédito con un microprocesador OMAP3530 a 720Mhz y 512MB de RAM. A este dispositivo irá conectado a través del puerto RS232 un sensor de temperatura. Este sensor estará gestionado por un sencillo microntrolador PIC que realizará lecturas continuas de temperatura y las enviará a la tarjeta IGEP a través de su conexión RS232.

En la tarjeta IGEP tendremos instalada una plataforma OSGI que recibirá los valores de la temperatura y los enviará a la red a través de la conexión WIFI que posee esta tarjeta.

En la segunda entrega de este tutorial, nos centraremos en este sistema controlador de la calefacción para llevar a la práctica los principios teóricos que estudiaremos en la primera parte.

- **Servidor de aplicaciones web.**

Este sistema estará compuesto por un PC de sobremesa, con un servidor de aplicaciones embebido en la plataforma OSGI (tomcat o jetty). Este PC recibirá a través de la intranet los datos de la temperatura de la calefacción y los presentará en una interfaz Web para que puedan ser accedidos desde cualquier punto de Internet. Así pues, esta interfaz Web dispondrá de un cuadro de mandos para el control de la calefacción (encendido y apagado, programar la temperatura deseada…).

Será en este sistema en el que nos basaremos para la última entrega de esta serie de artículos.

SECCIÓN I

ESPECIFICACIÓN OSGI

2 Especificación OSGI

OSGI proporciona un marco de trabajo java de uso general, seguro y administrado que soporta el despliegue dinámico de aplicaciones conocidas como "Bundles" o módulos.

Algunas de las características que componen este marco de trabajo:

- Es un sistema de módulos para la plataforma java.
- Incluye reglas de visibilidad, gestión de dependencias y versionado de los bundles.
- Es dinámico.
- La instalación, arranque, parada, actualización y desinstalación de bundles se realiza dinámicamente en tiempo de ejecución sin tener que detener por completo la plataforma.
- Se trata de una arquitectura orientada a servicios.
- Los servicios pueden ser registrados y consumidos dentro de la VM.

Podríamos destacar algunos de los principales beneficios que proporciona esta tecnología:

- Reutilización del código “out of the box”.
- Simplifica los proyectos en los que participan muchos desarrolladores en diferentes equipos.
- Se puede utilizar en sistemas más pequeños.
- Gestiona los despliegues locales o remotos.
- Se trata de una herramienta fácilmente ampliable.
- No es una tecnología cerrada.
- Gran aceptación. Es una tecnología usada por muchas empresas fabricantes.

La especificación OSGI, se define en los documentes publicados por el consorcio:

Figura 2.1 – Especificaciones OSGI

2.1 OSGI Layers

La funcionalidad del framework se divide en las siguientes capas:

- **Security Layer:** Capa de seguridad.
- **Module Layer:** Define el modelo de modularización, Este módulo define la reglas para el intercambio de paquetes java entre los Bundles.
- **Life Cycle Layer:** Gestiona el ciclo de vida de un bundle dentro del framework, sin tener que detener la VM.
- **Service Layer:**La capa de servicios proporciona un modelo programación dinámico para los desarrolladores de bundles, simplificando el desarrollo y despliegue de módulos a través del desacople de la especificación del servicio (java interface), de su implementación.
- **Actual Services, Execution Environment:** OSGi Minimum Execution Environment.

Podemos ver las diferentes capas de un sistema OSGI en la siguiente figura:

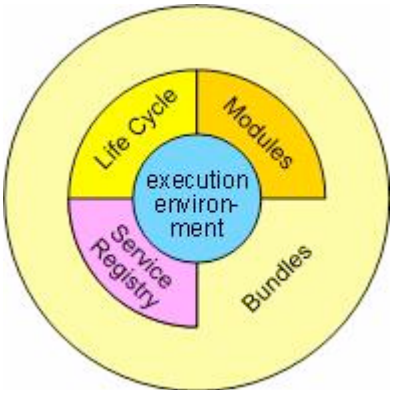

Figura 2.2 – Capas Sistema OSGI

2.1.1 Security Layer

En la última figura del apartado anterior, aunque no aparece dibujada esta capa, podríamos plasmarla envolviendo a todo el dibujo.

La capa de seguridad, es una capa opcional basada en "Java 2 Security", aunque añade nuevas reglas que la especificación "Java 2 Security" deja abiertas.

Esta capa provee una infraestructura para desplegar y manejar aplicaciones que tienen que ejecutarse en un entorno seguro y controlado.

Al tratarse de una capa opcional que no usaremos a lo largo de este tutorial, no entraremos en detalles, pero si comentar que OSGI proporciona un modelo de seguridad capaz de autenticar el código a través de:

- Ubicación del bundle y de sus paquetes.
- Firma del bundle y de sus paquetes.

La finalidad de esta capa, la podríamos resumir en dos objetivos:

- Integridad
- Autentificación

A más alto nivel, existen servicios capaces de controlar la gestión de permisos:

- **Permission Admin service** – Controla los permisos basados en "strings" de ubicaciones del código.
- **Conditional Permission Admin service** – Gestión los permisos basados en ubicaciones o firmantes.

Aunque con la breve descripción que hemos proporcionado en este apartado, no podemos afirmar que conocemos a fondo esta capa de OSGI, os propongo algunos sitios donde se puede obtener más información:

- http://sfelix.gforge.inria.fr/osgi-security/index.html
- http://www.ifi.uzh.ch/pax/uploads/pdf/publication/1099/Bachelorarbeit.pdf
- http://felix.apache.org/site/presentations.data/Building%20Secure%20OSGi%20Applications.pdf

2.1.2 Module Layer

La plataforma java en sí misma, proporciona una capacidad limitada para crear aplicaciones modulares.

El framework OSGI proporciona una solución genérica y estandarizada para la modularizacion de aplicaciones java.

"Module Layer" define la anatomía de los módulos (bundles) , las reglas de interoperación entre los mismos y la arquitectura de carga de clases.

En la siguiente figura se representa la arquitectura de un classloader tradicional en un servidor de aplicaciones.Podemos observar que se produce una carga vertical de clases, sin visibilidad horizontal entre módulos.

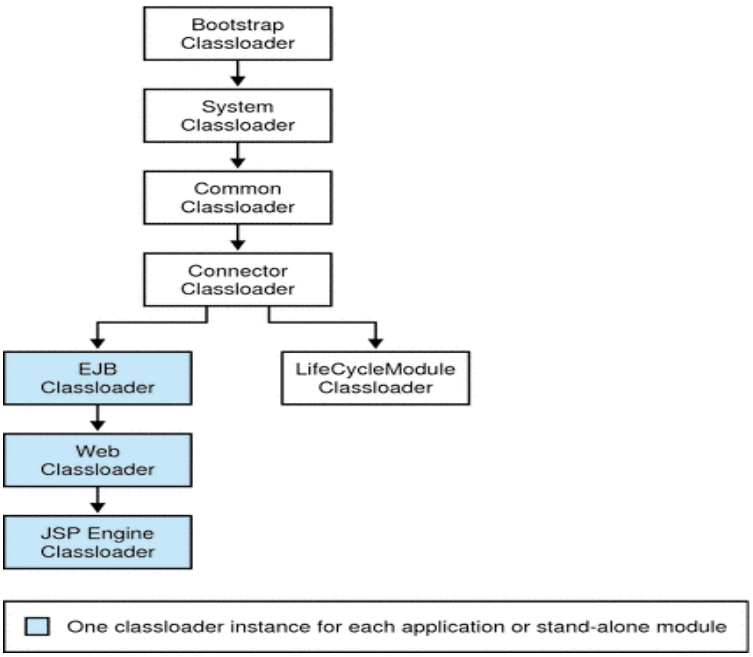

Figura 2.3 - Carga de clases tradicional en aplicaciones J2EE

La siguiente figura representa la interrelación entre los diferentes módulos de la plataforma. En el apartado 2.1.2.1.2 se detallará el orden de carga de clases en OSGI. Un bundle podrá importar paquetes de clases de otros bundles, así como exportar los suyos propios para ser usadas por otros módulos:

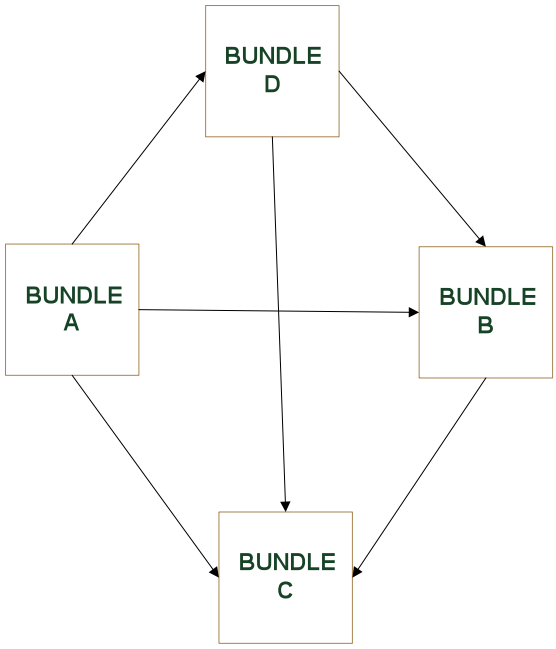

Figura 2.4 - OSGI ClassLoader

Algunas de las principales características que identifican esta capa de OSGI:

- La unidad de despliegue de aplicación es el Bundle, normalmente empaquetado en un jar.
- ClassLoaders independientes para cada bundle.
- Soporte para distintas versiones de un mismo bundle. (Se pueden importar versiones de paquetes).
- Código declarativo especifico para gestionar las dependencias (Importación y Exportación de paquetes).
- Filtrado de paquetes. Los bundles que exportan paquetes pueden definir las clases que serán visibles dentro de ese paquete.
- Soporte para "Bundle Fragments" (Son fragmentos de bundle que se adjuntan a otro bundle).

En siguiente figura vemos como se ordenan los módulos dentro de la plataforma OSGI:

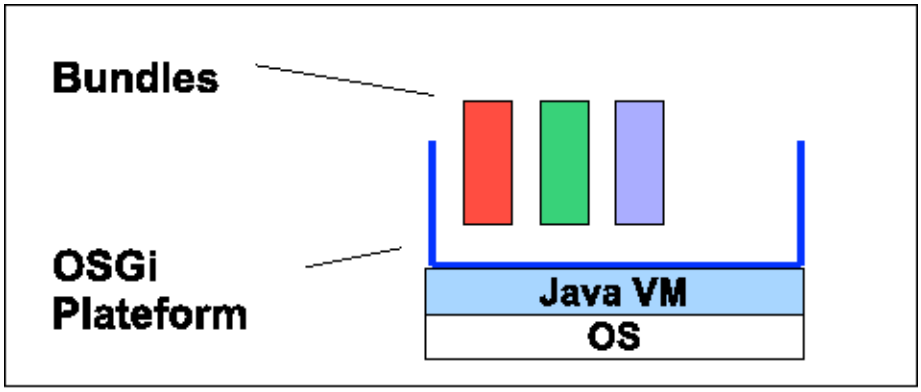

Figura 2.5 – Bundles

Y en la siguiente imagen podremos ver como la capa "Module Layer", permite la relación entre bundles:

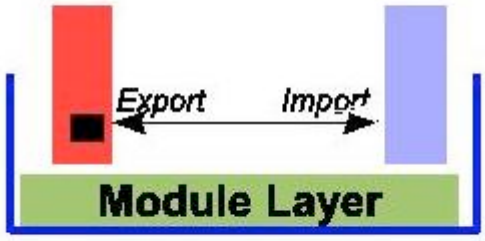

Figura 2.6 – Module Layer

2.1.2.1 **Bundle**

Como hemos mencionado anteriormente, un bundle es una aplicación empaquetada en un fichero jar, que se despliega en una plataforma OSGI.

Cada bundle contiene un fichero de metadatos organizado por pares de nombre clave: valor donde se describen las relaciones del bundle con el mundo exterior a él. (Por ejemplo que paquetes importa o que paquetes exporta....). Este fichero viene dado con el nombre MANIFEST.MF y se encuentra ubicado en el directorio META-INF.

Un ejemplo de un fichero MANIFEST.MF:

```
Manifest-Version: 1.0
```

```
Bundle-ManifestVersion: 2

Bundle-Name: HelloService Plug-in

Bundle-SymbolicName: com.javaworld.sample.HelloService

Bundle-Version: 1.0.0

Bundle-Vendor: JAVAWORLD

Bundle-Localization: plugin

Export-Package: com.javaworld.sample.service

Import-Package: org.osgi.framework;version="1.3.0"
```

Listado 2.1 – Manifest.mf

En el fichero de metadatos anterior, podemos diferenciar claves bastante intuitivas, donde se describen características del bundle, como por ejemplo, el nombre del bundle, la versión, paquetes que importa o exporta, etc.

En la siguiente imagen podemos ver la estructura de un Bundle:

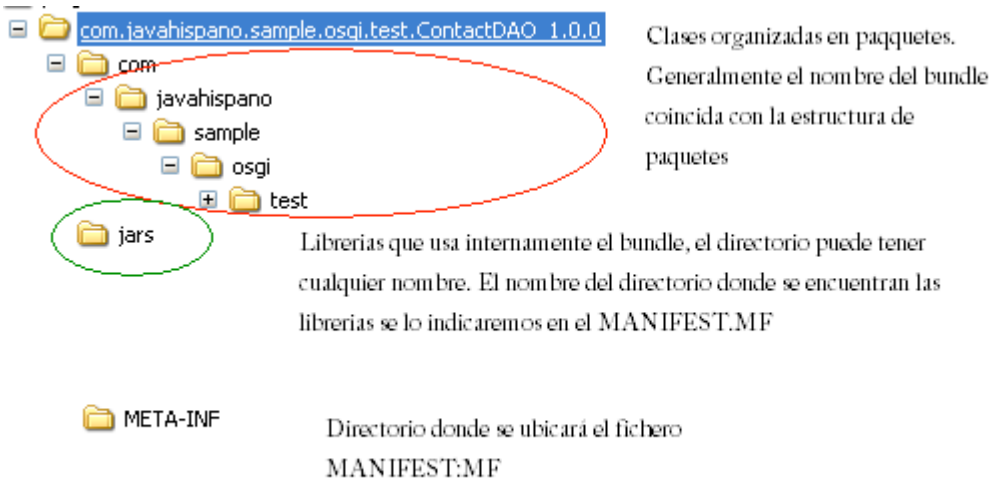

Figura 2.7 – Estructura de un Bundle

Como veremos más adelante, cuando desplegamos un bundle dentro de una plataforma OSGI comienza su ciclo de vida controlado por el propio framework. El desarrollador podrá controlar el ciclo de vida de su bundle, implementado una serie de interfaces de

OSGI. La principal interfaz a implementar es **org.osgi.framework.BundleActivator**, que nos obliga a declarar al menos dos métodos:

- **Start:** Método que se invoca al arrancar el bundle.
- **Stop:** Método que se invoca al parar el bundle.

2.1.2.1.1 Orden de búsqueda de clases

En los puntos anteriores hemos visto que un bundle puede importar paquetes de otro bundle o importar paquetes de las librerías que se encuentran "dentro" de él. Además de esto, un bundle puede importar paquetes del "System ClassLoder" (paquetes java.* o javax.*) o usar los paquetes de los "Fragment Bundles" anexados a el mismo. Por tanto para cada bundle debe de existir un orden de búsqueda o carga de clases. Este orden de búsqueda lo podemos ver reflejado en la siguiente ilustración:

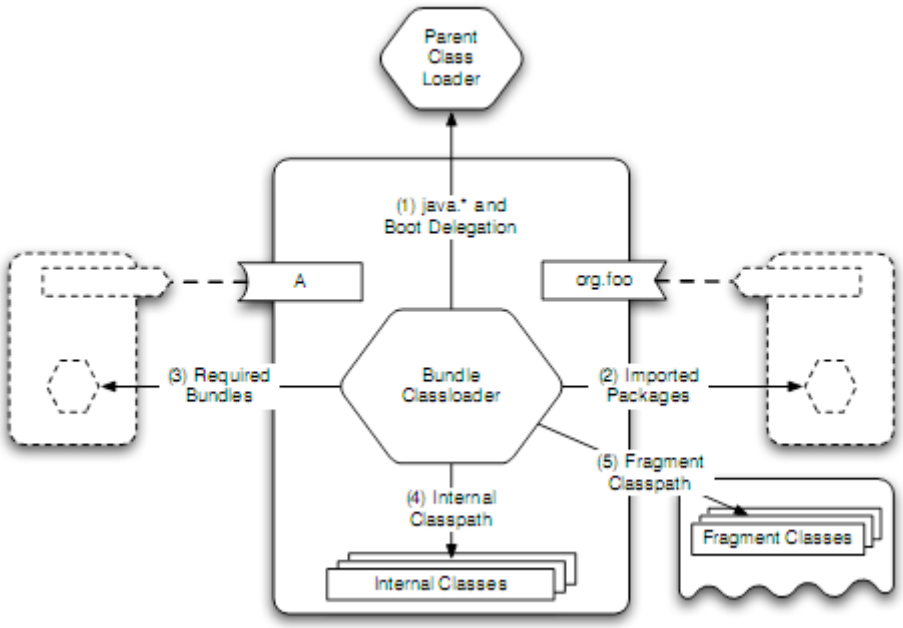

Figura 2.8 – Orden de búsqueda de clases

No os preocupéis si no entendéis todos los conceptos que aparecen en la figura anterior, iremos viéndolos a los largo de este tutorial.

Sin conocer todos estos conceptos, vamos intentar explicar el orden de carga de las clases:

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. (Veremos la clausula import-package mas adelante).
3. Comprobar si la clase a buscar se encuentra en el paquete importado a través de la clausula Required-Bundle del archivo Manifiest. (Veremos la clausula Required-Bundle mas adelante).
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". Veremos más adelante la definición de los fragment bundles, pero en una definición rápida podríamos decir que se tratan de "trozos" de bundle anexados a un bundle principal.

2.1.2.1.2 **MANIFEST.MF**

En el apartado 2.1.2.1 se presentó un ejemplo del un fichero "manifest.mf", que está compuesto pares de clave: valor, a las cuales en este tutorial llamaremos directivas.

A continuación se explican algunas de las directivas más interesantes del fichero MANIFEST.MF, que condiciona en gran medida el comportamiento del Bundle.

- **Bundle-ActivationPolicy: *lazy***

Políticas de activación del bundle. Especifica cómo se cargará el bundle una vez iniciado el framework.

El único valor posible para esta directiva es "lazy", que le indica al framework que el bundle se debe activar en modo "perezoso", es decir el bundle se cargara únicamente cuando sea requerido por otro bundle.

Si no se especifica esta directiva , por defecto el bundle se activará en modo "eager".

Si elegimos la política de activación en modo lazy, el framework seguirá los siguientes pasos en su arranque:

1. Se crea el "Bundle Context" para el bundle en cuestión. (Mas adelante veremos que es el bundle context. De momento podéis pensar que se trata de un concepto similar al HttpSession en una aplicación WEB. Esto no es del todo cierto, pero de momento nos puede valer para hacernos una idea...)
2. El estado del bundle se fija en "STARTING".

3. Se genera el evento "LAZY_ACTIVATION".
4. El framework se queda a la espera que alguna clase del bundle sea requerida para su carga.
5. Una vez que sea requerido, se generará el evento "STARTING".
6. El bundle es activado.
7. El nuevo estado del bundle será ACTIVE.
8. Se genera el evento STARTED.

Más adelante intentaremos crear un caso práctico sobre esta directiva, para comprobar que estos pasos de activación indicados por la especificación, son respetados. Pero de momento, toca seguir con la teoría....

- **Bundle-Activator: com.acme.fw.Activator**

Define el nombre de la clase que será invocada en el momento de activación del bundle. Esta directiva y la clase asociada a ella, es uno de los mecanismos que proporciona OSGI para controlar el ciclo de vida del software.

La clase especificada en la directiva bundle-activator, debe de existir e implementar a la interfaz BundleActivator, al implementar esta interfaz, nos veremos obligados a definir al menos los siguientes métodos:

- **start(BundleContext)** : Método que será instanciado al activar el bundle.
- **stop(BundleContext)** : Método invocado al parar el bundle.

En los próximos apartados, cuando comencemos a realizar prácticas, esta directiva será la primera que probaremos, para realizar nuestro primer "hola mundo". Paciencia ...

- **Bundle-Category: osgi, test, nursery**

A través de esta etiqueta podemos indicar la categoría de un bundle. Es una manera de poder agrupar bundle. Los valores de esta directiva son libres y pueden ser varios valores separados por comas. A pesar de tratarse de una directiva que admite cualquier valor, desde la especificación OSGI se aconsejan una serie de valores que podéis consultar en la siguiente URL: http://www.osgi.org/Specifications/Reference#categories

- **Bundle-Classpath: /jar/http.jar,.**

Esta cabecera define una lista de jars, separados por comas que especifican el classpath interno del bundle. Estas librerías se encontrarán comprimidas dentro del bundle.

Por ejemplo, tenemos la siguiente estructura en nuestro bundle "com.javahispano.test":

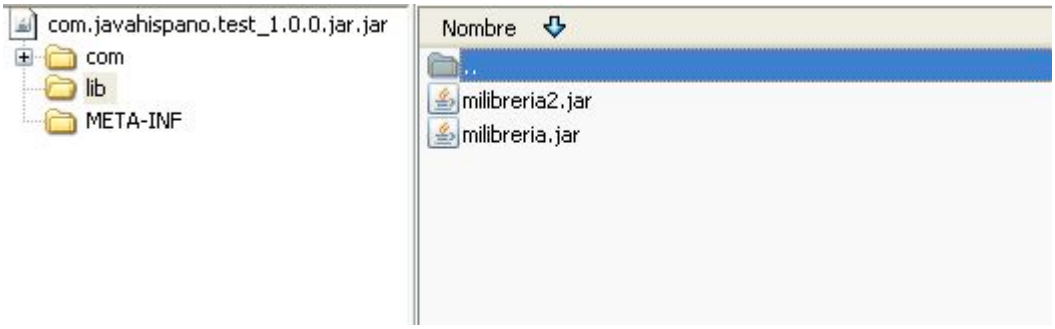

Figura-2.9 Bundle Classpath

Para poder registrar estas librerías como classpath interno del bundle deberemos añadirlas a nuestro manifest.mf:

```
Manifest-Version: 1.0

Ant-Version: Apache Ant 1.7.0

Created-By: 10.0-b22 (Sun Microsystems Inc.)

Bundle-ManifestVersion: 2

Bundle-Name: Datastorage Plug-in

Bundle-SymbolicName: com.javahispano.test

Bundle-Version: 1.0.0

Bundle-Activator: com.javahispano.test.Activator

Bundle-Vendor: Javahispano

Eclipse-LazyStart: true

Import-Package: org.osgi.framework;version="1.3.0"

Bundle-ClassPath: lib/milibreria.jar, lib/milibreria2.jar
```

Listado 2.2 Manifest.mf – Bundle Classpath

- **Bundle-ContactAddress: 2400 Oswego Road, Austin, TX 74563**

Proporciona una dirección de contacto, del proveedor del bundle.

- **Bundle-Copyright: OSGi (c) 2009**

Copyright del bundle.

- **Bundle-Description: Network Firewall**

Breve descripción del bundle.

- **Bundle-DocURL: http:/www.acme.com/Firewall/doc**

URL donde se encuentra ubicada la documentación relacionada con el bundle.

- **Bundle-Localization: OSGI-INF/l10n/bundle**

Ubicación de los ficheros dependientes del idioma.

Esta cabecera también la probaremos a lo largo del tutorial.

- **Bundle-ManifestVersion: 2**

Indica la especificación de osgi sobre la que está basada el Bundle. Por defecto, si no se indica nada, el valor es 1. Este valor indica que el bundle se basa en las reglas de la especificación OSGI 3. Si tiene valor 2, se tratará de la especificación de OSGI 4.

- **Bundle-Name: Firewall**

Nombre del bundle. Nombre corto del bundle (human-readable).

- **Bundle-NativeCode: /lib/http.DLL; osname = QNX; osversion = 3.1**

Librerías nativas del bundle. A la hora de cargar una librería nativa, primeramente el bundle leerá esta directiva para buscar esta librería en el classpath interno, si no la encuentra la buscará en los "Fragment bundles" y si finalmente sigue sin encontrarla delegará al "parent classloader".

Los atributos que acepta esta directiva son:

  - *osname* – Nombre del sistema operativo sobre el que correrán las librerías nativas. En la "Core Specification" podemos encontrar una lista de los nombres para los sistemas operativos.

  - *osversion* –Version del sistema operativo.

  - *processor* – Arquitectura del procesador. En la "Core Specification" podemos encontrar una lista con los nombres de las arquitecturas del procesador.

  - *language* – Código ISO del lenguaje.

- *selection-filter* – Filtro.

Los "Fragment Bundles" los explicaremos más adelante, también realizaremos algunas prácticas de carga de librerías nativas, junto con los "Fragment Bundles".

- **Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0**

Entorno de ejecución sobre el que se puede ejecutar el bundle. Si nuestro módulo puede ejecutarse en varios entornos, podremos indicarlo separando sus nombres por comas.

Algunos de los entornos más comunes donde se puede ejecutar OSGI, son:

Name	Description
CDC-1.0/Foundation-1.0	Equal to J2ME Foundation Profile
OSGi/Minimum-1.1	OSGi EE that is a minimal set that allows the implementation of an OSGi Framework.
JRE-1.1	Java 1.1.x
J2SE-1.2	Java 2 SE 1.2.x
J2SE-1.3	Java 2 SE 1.3.x
J2SE-1.4	Java 2 SE 1.4.x
J2SE-1.5	Java 2 SE 1.5.x
JavaSE-1.6	Java SE 1.6.x
PersonalJava-1.1 Personal	Java 1.1
PersonalJava-1.2 Personal	Java 1.2
CDC-1.0/PersonalBasis-1.0	J2ME Personal Basis Profile
CDC-1.0/PersonalJava-1.0	J2ME Personal Java Profile

A lo largo de este tutorial veremos muchos ejemplos, la mayoría sobre entornos JavaSE-1.6, pero también intentaremos realizar pruebas de la plataforma en entornos más reducidos.

- **Bundle-SymbolicName: com.acme.daffy**

Esta etiqueta define un nombre único para el Bundle. Con este nombre es con el que trabajaremos en el entorno de ejecución.

La directiva acepta los siguientes atributos:

- singleton – Significa que el bundle solo va a tener una única versión resuelta. Si ponemos el atributo a "true" indica que se trata de un bundle de tipo singleton.Por defecto, si no se indica nada se considera que esta propiedad esta a "false".

- fragment-attachment – Define como se pueden adjuntar los "fragments bundles":

  - always
  - never
  - resolve-time

Bundle-UpdateLocation: http://www.acme.com/Firewall/bundle.jar

Esta directiva especifica la URL desde la cual se puede actualizar el bundle.

- **Bundle-Vendor: OSGi Alliance**

Descripción breve del proveedor del bundle (human-readable).

- **Bundle-Version: 1.1**

Version del bundle. La versión por defecto es 0.0.0. Gracias esta característica, podemos decir que osgi permite realizar desarrollos incrementales.

- **DynamicImport-Package: com.acme.plugin.***

Esta etiqueta contiene valores separados por comas, donde se indican los paquetes que pueden ser importados dinámicamente. Generalmente esta directiva se usa para evitar dependencias cíclicas cuando cargamos clases a través de Class.forName(). No os preocupéis si no lo termináis de entender, crearemos en este tutorial un apartado con prácticas alrededor de esta directiva y otras similares relacionadas con la gestión de dependencias. (Concretamente con las directivas Export-Package, Import-Package y Require-Bundle).

- **Export-Package: org.osgi.util.tracker;version=1.3**

Un bundle puede exponer sus paquetes, para que puedan ser usados por otro bundle. Además de eso, el bundle podrá establecer que versión de paquete desea exponer. Como he mencionado anteriormente este tema lo trataremos en profundidad en futuros apartados, a

través de las practicas y junto las directivas DynamicImport-Package, Import-Package y Require-Bundle.

- **Fragment-Host: org.eclipse.swt; bundle-version="[3.0.0,4.0.0)"**

Esta directiva se utiliza para trabajar con los "Fragment Bundle". Un "Fragment Bundle" es un bundle incompleto que se adjunta a un "Host Bundle", proporcionándole nuevas funcionalidades al bundle.

Un ejemplo de uso de los fragmentos, es cuando tenemos una implementación que requiere código especifico de la plataforma sobre la que está corriendo (código nativo). El código a cargar será diferente dependiendo de la plataforma sobre la que queramos correr nuestro bundle. Siendo este caso, deberíamos tener un bundle diferente para cada sistema operativo. Para evitar esto, se usan los "Fragment Bundle", es decir, podríamos crear un fragmento de bundle por cada plataforma sobre la que queremos que corra y adjuntar estos fragmentos a un bundle principal, que dependiendo del SO, cargará un fragmento u otro.

Esta es una práctica muy interesante que realizaremos más adelante. En esa práctica, aprovecharemos también para probar la directiva que nos permite cargar librerías nativas (Bundle-NativeCode).

- **Import-Package: org.osgi.util.tracker,org.osgi.service.io;version=1.4**

Al igual que un bundle puede exportar paquetes, también puede importar los paquetes que han sido expuestos otro bundle a través de esta directiva. Esta directiva también nos permite importar versiones concretas de un paquete.

- **Require-Bundle: com.acme.chess**

Similar a "Import-Package" salvo que en vez de importar un paquete, importamos todos los paquetes de un bundle. Es una directiva que hay que usar con precaución ya que es bastante agresiva para la plataforma. También hay que tener en cuenta que en la carga de clases, prevalece la directiva import-package sobre require-bundle (ver apartado 2.1.2.1.2).

2.1.3 LifeCicle Layer

En el apartado anterior, cuando explicábamos la etiqueta Bundle-Activator, comentamos que OSGI permite llevar una gestión absoluta del ciclo de vida del software, es en este punto del documento, donde hablaremos sobre la capa responsable de ello.

Antes de comenzar a trabajar con esta capa , debemos tener claros los siguientes conceptos:

- **Bundle** – Representa un modulo instalado en el framework.

- **Bundle Activator** – Interface que deben implementar los bundles para que comience su ciclo de vida. Esta interface se usa para controlar el arranque y la parada de un bundle.
- **Bundle Context** – Contexto del bundle. Este objeto es pasado en tiempo de ejecución al bundle, concretamente en los métodos start y stop del Activator. **Bundle Event** – Evento generado por la plataforma al producirse un cambio de estado en el ciclo de vida de un bundle.
- **Framework Event** – Evento generado por la plataforma, al producirse un error en el entorno o un cambio de estado del propio framework.
- **Bundle Listener** – "Listenner" de los eventos producidos por los bundles.
- **Framework Listener** – "Listenner" de los eventos producidos por la propia plataforma.
- **Bundle Exception** – Excepción producida por un fallo del bundle.
- **System Bundle** – Bundle que representa el propio framework.

A través de los conceptos comentados anteriormente podemos gestionar el ciclo de vida de un bundle, controlando los diferentes estados por los que este puede pasar:

- **INSTALLED** – El bundle ha sido instalado correctamente.
- **RESOLVED** – Todas las clases java que el bundle necesita están disponibles. Este estado indica que el bundle está listo para ser arrancado o que el bundle ha sido detenido.
- **STARTING** – El bundle está arrancando. En caso de que hayamos establecido la propiedad Bundle-ActivationPolicy a lazy, se quedará en este estado hasta que se necesario su arranque.
- **ACTIVE** – El bundle ha sido correctamente activado y está en funcionamiento.
- **STOPPING** – El bundle ha sido detenido.
- **UNINSTALLED** – El bundle ha sido desinstalado. No se puede mover a otro estado.

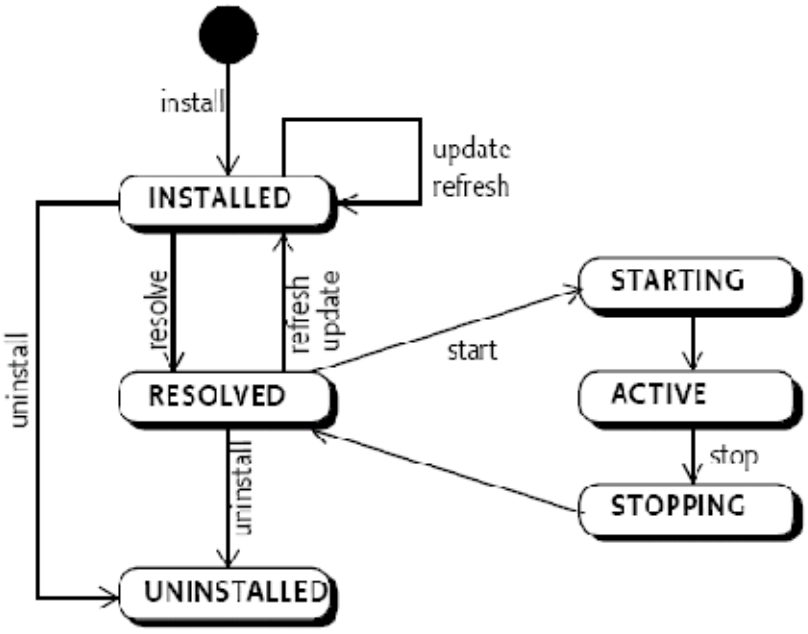

Figura-2.10 Ciclo de vida de un bundle

Más adelante realizaremos prácticas para ver el ciclo de vida de los bundles.

2.1.4 **Service Layer**

OSGI, además de ser un sistema modular, esta también orientado a servicios.

Un servicio es un objeto java que es registrado bajo una o varias interfaces. Los bundles pueden registrar servicios, buscarlos o recibir notificaciones cuando el estado de un servicio ha sufrido alguna variación.

Para comprender esta capa, debemos tener claros algunos conceptos relacionados:

- **Service** – Entendemos por servicio, un objeto registrado en la plataforma bajo una o más interfaces. Dicho objeto registrado puede ser usado por otros bundles.
- **Service Registry** – Gestiona el registro de servicios.
- **Service Reference** – Referencia a un servicio registrado. Con este objeto podemos obtener las propiedades del servicio, no el propio servicio.
- **Service Listener** – Listener para escuchar los eventos de los servicios.
- **Service Event** – Evento producido al registrar, modificar o dar de baja un servicio.
- **Filter** – Objeto para filtrar servicios.

Por lo tanto, con OSGI podremos registrar una clase java como un servicio, a través del contexto del bundle. Además con el "Bundle Context" podremos buscar servicios

registrados en la plataforma. Finalmente podremos escuchar y filtrar los eventos producidos en los cambios de estado de un servicio.

Quizá la idea de crear clases en forma de servicios, choca un poco con la idea de exportar e importar clases o paquetes, es decir para invocar a los métodos de una clase de otro bundle tenemos dos opciones:

Si la clase ha sido exportada por el otro bundle, la podríamos importar desde el nuestro e invocar a sus métodos
El otro bundle puede publicar su clase como un servicio, haciendo uso de interfaces. Nuestro bundle capturaría ese servicio e invocaría a sus métodos.

Lógicamente si registramos nuestras clases como servicios, al trabajar con interfaces, nos quedaría un código con un acoplamiento de dependencias bastante débil. También OSGI nos proporciona mecanismos para controlar por ejemplo: el número de instancias de un servicio, cuando se encuentra disponible un servicio, parar y arrancar servicios etc...

En la siguiente imagen podemos ver la diferencia entre exportar - importar clases y publicarlas y consumirlas como servicios:

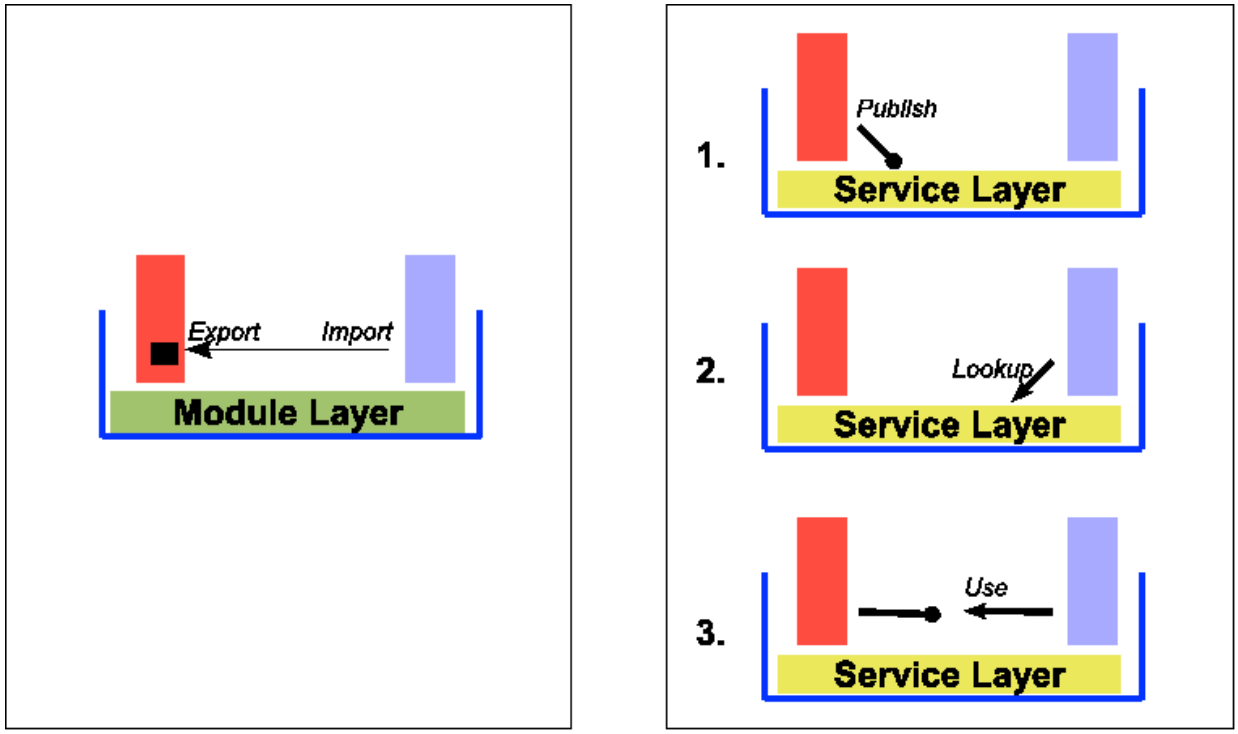

Figura-2.11 **Module Layer y Service Layer**

2.1.4.1 System Services

Además de los servicios registrados por el desarrollador, la plataforma proporciona una serie de servicios (System Services) que comentaremos muy brevemente en esta sección, para posteriormente verlos más en detalle mediante casos prácticos. Estos servicios son:

- **Log Service Specification:** Servicio de gestión de logs de la plataforma. Este servicio se divide en dos: uno para "logear" información y otro para leer trazas que se están escribiendo a través de este servicio.
- **Http Service Specification:** Permite el acceso remoto a servicios y recursos. Actualmente esta especificación permite registrar servlets y recursos estáticos (HTML, imágenes...).
- **Device Access Specification:** Permite descubrir los dispositivos conectados, así como los servicios expuestos por los mismos.
- **Configuration Admin Service:** Establece un estándar para configurar los bundles en un entorno OSGI.
- **Metatype Service Specification:** Servicio de metadatos asociados a un bundle. Este servicio examinará el bundle especificado en busca de documentos con metadatos.
- **Preferences Service Specification:** Ofrece un mecanismo de persistencia de ciertos datos o preferencias asociados a la ejecución de un bundle.
- **User Admin Service Specification:** Servicio asociado a la capa de seguridad. Ofrece mecanismos de identificación.
- **Wire Admin Service Specification**: Podemos decir que proporciona una forma flexible de conexión de consumidores de servicios y productores.
- **IO Connector Service Specification:** Servicio de conectores I/O basado en J2ME (javax.microedition.io).
- **UPnP™ Device Service Specification**: Este servicio provee de los protocolos necesarios para una red "peer-to-peer". Especifica cómo hay que conectarse a una red y como los dispositivos conectados pueden ser controlados enviando mensajes XML sobre HTTP.
- **Declarative Services Specification** : Sirve para crear nuestros propios servicios, pero a través de clases POJO y configuraciones declarativas en ficheros XML. Lo veremos junto a las prácticas de la sección III y IV.
- **Event Admin Service Specification:** Servicio de gestión de eventos de la plataforma OSGI. (Podríamos compararlo con los servicios JMS) .
- **Service Tracker Specification:** Proporciona utilidades para el registro de servicios OSGI.
- **XML Parser Service Specification:** Incluye las directrices y servicios para el procesamiento de XML dentro de la plataforma OSGI.

2.2 Ejemplos Básicos

Existen tres ejemplos básicos que se pueden tomar como punto de partida para iniciarse en la plataforma OSGI.

Los ejemplos que vamos a ver son:

- Una clase activator
- Un bundle que expone un servicio
- Un bundle que consume el servicio publicado por el modulo anterior

Estos dos ejemplos, nos vendrán bien para utilizarlos en las demostraciones de los apartados de la sección II.

2.2.1 Clase Activator

A continuación, veremos el código fuente de nuestro primer bundle: "Hola Mundo".

Para que nuestro bundle nos salude al arrancar y se despida al parar la aplicación, solo necesitamos dos archivos:

- El archivo manifest.mf, que como punto clave tendrá la definición de la clase Activator.
- Clase java que implementa a la interfaz de OSGI BundleActivator

```
Manifest-Version : 1.0

Bundle-ManifestVersion : 2

Bundle-Name : Holamundo

Bundle-SymbolicName:org.javahispano.osgi.holamundo

Bundle-Version : 1.0.0

Bundle-Activator : org.javahispano.osgi.holamundo.Activator

Import-Package : org.osgi.framework;version="1.3.0"
```

Listado-2.3 Manifest.mf Holamundo bundle

La mayoría de las directivas que vemos en el fichero anterior, las hemos visto en los apartados anteriores. En este ejemplo queremos destacar la directiva Bundle-Activator que

apunta a la clase org.javahispano.osgi.holamundo.Activator que veremos a continuación.

```
package org.javahispano.osgi.holamundo;

import org.osgi.framework.BundleActivator;

import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

   public void start(BundleContext context) throws Exception {

      System.out.println("HOLA MUNDO");

   }

   public void stop(BundleContext context) throws Exception {

      System.out.println("ADIOS MUNDO CRUEL");

   }

}
```

Listado-2.4 Activator HolaMundo Bundle

En el listado anterior vemos como se definen los métodos start y stop de la clase activadora. Al arrancar, el bundle invocará al método start y cuando lo paremos al método stop. Para poder parar completamente la ejecución del bundle cuando invocamos al método Stop, se suele hacer uso de los Threads, esto lo entenderemos mejor con un ejemplo de otra clase Activator:

```
 package org . osgi . tutorial ;

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

public class MiClaseActivator implements BundleActivator {

     private Thread thread ;

 public void start ( BundleContext context ) throws Exception {
       thread = new Thread ( new MiClaseQueHaceCosas ( ) ) ;
       thread . start ( ) ;
```

```
}

public void stop ( BundleContext context ) throws Exception {
        thread . interrupt ( ) ;
}
}

   class MiClaseQueHaceCosas implements Runnable {

      public void run ( ) {

      }

   }
```

Listado-2.5 Activator con threads

2.2.2 Bundle que expone un servicio

En apartados anteriores, hemos comentado como OSGI además de exportar paquetes y clases, puede registrar sus objetos como servicios para que sean usados por otros bundles. Ahora desarrollaremos un servicio OSGI que simplemente tiene un método saluda.

El primer bundle org.javahispano.osgi.holamundoservice, tiene la siguiente estructura:

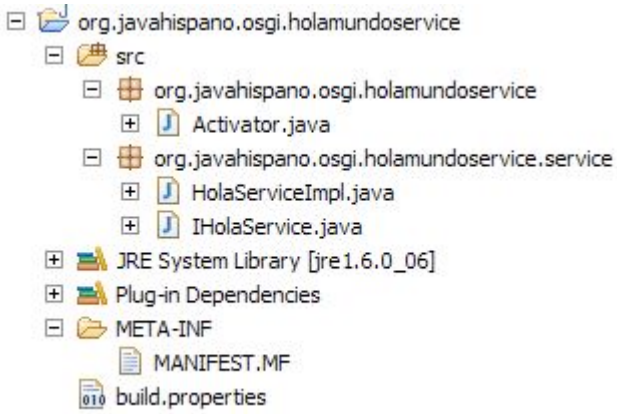

Figura-2.12 Bundle HolaMundoService

Las partes más importantes de este bundle son:

- IHolaService: Interface del Servicio.

- HolaServiceImpl: Implementación de la interfaz del servicio.
- Activator: Se ejecuta al arrancar el bundle y registra el servicio (IHolaService).
- Manifest.mf: Exporta el paquete donde se encuentra la interfaz, para que pueda ser usado por otros bundles.

```
package org.javahispano.osgi.holamundoservice.service;
public interface IHolaService {
    public void saluda(String nombre);
}
```

Listado-2.6 IHolaService

```
package org.javahispano.osgi.holamundoservice.service;
public class HolaServiceImpl implements IHolaService{
    @Override
    public void saluda(String nombre) {
       System.out.println("El servicio te saluda: Hola " + nombre);
   }
}
```

Listado-2.7 HolaServiceImpl

```
package org.javahispano.osgi.holamundoservice;

import org.javahispano.osgi.holamundoservice.service.HolaServiceImpl;
import org.javahispano.osgi.holamundoservice.service.IHolaService;
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(IHolaService.class.getName(),
   new HolaServiceImpl(), null);
 }
 public void stop(BundleContext context) throws Exception {
  registration.unregister();
 }
}
```

Listado-2.8 Activator HolaMundoService

```
Manifest-Version: 1.0
Bundle-ManifestVersion : 2
Bundle-Name : Holamundoservice Plug-in
Bundle-SymbolicName : org.javahispano.osgi.holamundoservice
```

```
Bundle-Version : 1.0.0
Bundle-Activator : org.javahispano.osgi.holamundoservice.Activator
Import-Package: org.osgi.framework;version="1.3.0"
Eclipse-LazyStart : true
Export-Package : org.javahispano.osgi.holamundoservice.service
```

Listado-2.9 Manifest.mf HolaMundoService

2.2.3 Bundle que consume un servicio

Ahora veremos el bundle que consume el servicio anterior. Al arrancar recogerá del registro de servicios OSGI, la interfaz del servicio e invocará al método "saluda".

El bundle que consume el servicio, estaria compuesto por:

- **Activator:** Consume el servicio al arrancar el bundle.
- **Manifest.mf:** Importa el paquete del servicio a consumir.

```
 package org.javahispano.osgi.holamundoconsumer;

import org.javahispano.osgi.holamundoservice.service.IHolaService;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

public class Activator implements BundleActivator {

  public void start(BundleContext context) throws Exception {

    ServiceReference ref = context.getServiceReference(IHolaService.class
.getName());

    IHolaService holaService = (IHolaService) context.getService(ref);
    holaService.saluda( "JavaHispano");
  }

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

Listado-2.10 Activator HolaMundoConsumer

```
Manifest-Version : 1.0
Bundle-ManifestVersion : 2
Bundle-Name: Holamundoconsumer Plug-in
Bundle-SymbolicName : org.javahispano.osgi.holamundoconsumer
Bundle-Version : 1.0.0
Bundle-Activator : org.javahispano.osgi.holamundoconsumer.Activator
Import-Package :
org.javahispano.osgi.holamundoservice.service,org.osgi.framework;
version="1.3.0"
```

Listado-2.11 Manifest.mf HolaMundoConsumer

SECCIÓN II

Implementaciones OSGI y entornos de desarrollo

3 Entornos de desarrollo y ejecución OSGI.

Dentro del mundo OpenSource podemos encontrar varias plataformas certificadas en OSGI, algunas de ellas son:

- **Apache Felix:** Es un proyecto de la fundación apache. Es una implementación de osgi R4. El proyecto inicialmente partió del código fuente donado por el proyecto OSCAR de ObjectWeb.
- **Eclipse Equinox**: Plataforma que implementa la R4 de OSGI. Equinox, podemos decir que es núcleo sobre el que se basa el famoso IDE de desarrollo Eclipse. Una de las razones por las que eclipse se ha extendido tanto, es por su carácter modular, ofreciendo al desarrollador múltiples plugins (bundles) .
- **Knopflerfish:** Implementación de R3 y R4 de OSGI.
- **FUSE ESB 4:** Proyecto OpenSource ESB (Enterprise Service Bus) basado en Apache ServiceMix 4 y que tiene una implementación completa de OSGI R4.
- **Newton Project:** Es una implementación distribuida de OSGI R4. Newton describe los sistemas distribuidos utilizando el estándar de SCA. Newton hace uso de OSGi para la conexión de componentes dentro de una sola JVM y la tecnología Jini para el seguimiento y la conexión de las dependencias entre los diferentes componentes en diferentes JVM.
- **Concierg:** Implementación compacta de OSGI R3. Esta especialmente pensada para dispositivos con poca capacidad. Al tratarse de una implementación de R3, y no tener soporte para la R4, no entraremos en detalles sobre el, en este articulo.

En esta sección aprenderemos a configurar cada una de estas implementaciones, para conseguir un entorno de ejecución de nuestros bundles.

Tenemos diferentes formas de desarrollar código para las plataformas anteriormente comentadas. Las más comunes son:

Eclipse: Eclipse es uno de los IDEs de java más extendidos en la comunidad de desarrolladores. El propio eclipse está basado en Equinox, esa es la razón por la que este IDE, ofrece una integración completa al desarrollo de Bundles, especialmente al desarrollo sobre su propia implementación: Equinox. Con Eclipse IDE, también podremos ejecutar nuestros bundles en otros contenedores diferentes a equinox.

Maven: Maven es una herramienta de gestión de despliegues y dependencias. Mediante plugins de maven podemos compilar y desplegar nuestros bundles. Recordemos que la gestión de dependencias de maven, tiene cierta similitud con la gestión de dependencias de OSGI. Podremos crear nuestro bundles, con cualquier IDE que tenga soporte para maven.

En mi opinión eclipse facilita enormemente el trabajo a la hora de trabajar con OSGI, ya que podemos desplegar directamente nuestros bundles sobre la plataforma OSGI, aunque también nos da la opción de empaquetarlos y exportarlos a la ubicación que prefiramos.

3.1 Entornos de Ejecución

Hemos mencionado en la introducción a esta sección algunas de las implementaciones osgi más conocidas. A continuación comenzaremos a indagar sobre las tres más famosas Equinox, Apache Felix y Knopflerfish.

3.1.1 Eclipse Equinox

Es obvio que antes de la instalación de esta plataforma necesitaremos tener instalado una maquina virtual java. Una vez instalada la JVM, nos descargaremos la ultima implementación estable de OSGI Equinox, lo haremos accediendo a la URL: *http://download.eclipse.org/equinox/*

Una vez en la página de equinox, nos encontraremos con diferentes tipos de descarga:

- **Descarga de la implementación OSGI base:** Se trata de un fichero .jar (bundle), que contiene los elementos básicos para poder ejecutar nuestro entorno. Esta descarga no incorpora por ejemplo, las implementaciones de los servicios OSGI.
- **Descarga completa de equinox:** Se trata de un fichero .zip que contiene el bundle base comentado anteriormente, más todas las configuraciones e implementaciones de los servicios OSGI.

- **Descarga de los bundles opcionales:** Mediante esta opción podremos descargar uno a uno los bundles de equinox que implementan los diferentes servicios de OSGI.

Por el momento y para este tutorial, nos bajaremos la opción de equinox al completo. Para usuarios más avanzados se podrían plantear la opción de descargarse la instalación base e ir añadiendo bundles de equinox a medida que sean necesarios para su desarrollo especifico.

Una vez descargado el zip con todos los bundles, lo descomprimiremos en una carpeta, quedando una estructura de directorios similar a la siguiente:

Figura-3.1 Estructura de directorios Equinox

De momento nos centraremos en la carpeta plugins, que es donde se encuentra la implementación base de equinox:

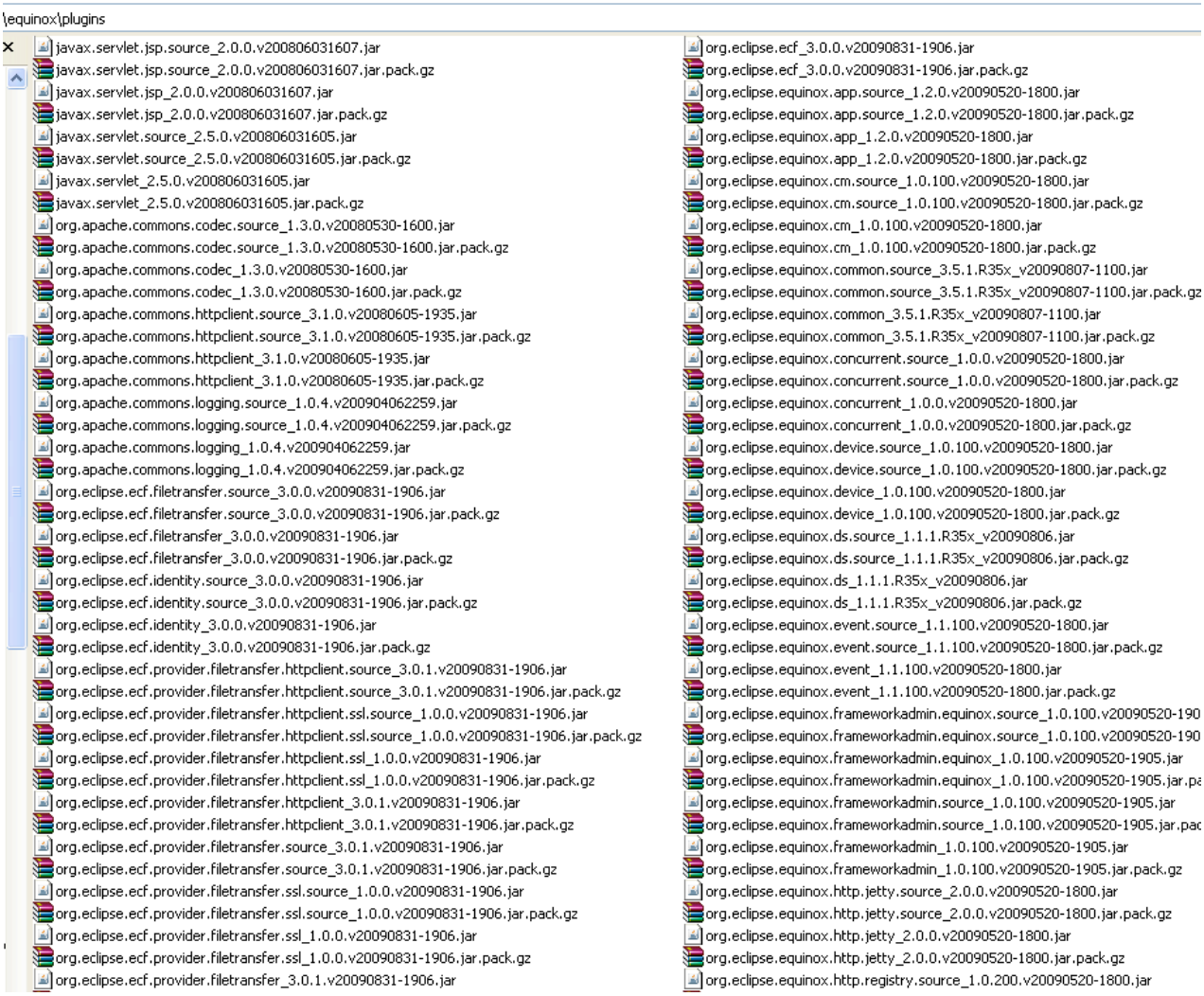

Figura-3.2 Contenido del directorio plugins

No debemos asustarnos al ver la cantidad de librerías que alberga este directorio. Por ahora solo trabajaremos con uno de estos jars: org.eclipse.osgi_3.x.x_xxx.jar.

Para arrancar la consola osgi-equinox deberemos ejecutar este jar, mediante el siguiente comando: java -jar org.eclipse.osgi_3.x.x_xxx.jar -console.

Una vez ejecutado este comando entraremos a la consola osgi. Algunas de las acciones que podemos realizar desde esta consola son:

- Listar los bundles instalados sobre la plataforma.
- Instalar / Desinstalar bundles.
- Arrancar / Parar bundles.
- Diagnosticar bundles.
- Revisar la configuración de los bundles.

3.1.1.1 Línea de Comandos Equinox

Comenzaremos listando los bundles instalados en nuestra plataforma:

```
osgi>ss

Framework is launched.

id      State       Bundle

0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827
```

Al ejecutar el comando "ss" veremos que solo tenemos un único bundle, que es el encargado de gestionar la plataforma. Al ejecutar este comando podremos observar que la salida resultante se divide en tres columnas:

- Id: Identificador único del bundle en cuestión.
- State: Estado en el que se encuentra el modulo listado (En este primer caso se encuentra activo).
- Bundle: Nombre del bundle.

Como el ejemplo es un poco pobre, vamos a instalar un nuevo bundle. Este primer bundle, mostrará un "Hola Mundo". De momento usaremos el bundle listado en el apartado 2.2.1 ya compilado adjunto a este artículo.

Para instalar este bundle ejecutaremos desde la consola OSGI ejecutamos:

```
osgi> install file:///bundleInstall/org.javahispano.osgi.holamundo_1.0.0.jar

Bundle id is 8

osgi>
```

Si ahora volvemos a listar los bundles:

```
osgi> ss

Framework is launched.

id      State       Bundle

0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827
```

```
8      INSTALLED   org.javahispano.osgi.holamundo_1.0.0

osgi>
```

Vemos como nuestro nuevo bundle ha sido instalado, pero no está activo. Lo activaremos ejecutando el comando START pasándole como parámetro el id del bundle que queremos arrancar:

```
osgi> start 8

HOLA MUNDO

osgi>
```

Por fin!!!, ya hemos ejecutado nuestro primer "Hola Mundo". Si observamos el código fuente del bundle listado en el apartado 2.2.1, podremos deducir que se ha ejecutado el método start(BundleContext context) de la clase Activator.

Listamos de nuevo los bundles instalados en la plataforma:

```
osgi> ss

Framework is launched.

id      State       Bundle

0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827

8       ACTIVE      org.javahispano.osgi.holamundo_1.0.0

osgi>
```

Ahora nos aparece el bundle activo. Bien, pues vamos a detenerlo, para ello ejecutamos el comando STOP pasándole como parámetro el id del bundle que queremos detener:

```
osgi> stop 8

ADIOS MUNDO CRUEL
```

Si observamos el código fuente del bundle listado anteriormente, podremos deducir que se ha ejecutado el método stop(BundleContext context) de la clase Activator.

Si listamos nuevamente los bundles, veremos cómo ha vuelto a cambiar su estado:

```
osgi> ss

Framework is launched.

id      State       Bundle

0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827

8       RESOLVED    org.javahispano.osgi.holamundo_1.0.0
```

En los primeros apartados de este tutorial vimos como además del "Activator", todos los bundles tienen un fichero descriptor del propio bundle y de sus relaciones con otros bundles (MANIFEST.MF). Podremos ver las propiedades de nuestro bundle, ejecutando desde la consola:

```
osgi> bundle 8

org.javahispano.osgi.holamundo_1.0.0 [8]

  Id=8, Status=RESOLVED    Data Root=F:\tutoriales\osgi-
spring\manual_montero\eq

uinox\plugins\configuration\org.eclipse.osgi\bundles\8\data

  No registered services.

  No services in use.

  No exported packages

  Imported packages

    org.osgi.framework; version="1.5.0"<org.eclipse.osgi_3.5.1.R35x_v20090827 [0

]>

  No fragment bundles

  Named class space

    org.javahispano.osgi.holamundo; bundle-version="1.0.0"[provided]
```

```
No required bundles

osgi>
```

Esta forma de instalar bundles es útil, para realizar instalaciones en caliente, pero imagínense que tenemos muchos bundles, sería poco práctico tener que instalarlos de uno en uno mediante comandos. Equinox nos ofrece una forma de cargar todos los bundles en el arranque de la plataforma, a través del fichero de configuración config.ini. Este fichero lo ubicaremos en el directorio EQUINOX_HOME/plugins/configuration. Hay que tener en cuenta que el directorio "configuration" lo crea equinox al arrancar por primera vez.

Para ilustrar un ejemplo sobre el uso del config.ini, vamos a instalar dos nuevos bundles: org.javahispano.osgi.holamundoservice_1.0.0.jar y org.javahispano.osgi.holamundoconsumer_1.0.0.jar que se encuentran adjuntos a este articulo. Se trata de los bundles listados en los apartados 2.2.2 y 2.2.3, pero en este caso los tenemos ya compilados y empaquetados.

Ahora queremos arrancar la plataforma con estos dos nuevos bundles mas el bundle anterior:

- org.javahispano.osgi.holamundo_1.0.0.jar
- org.javahispano.osgi.holamundoservice_1.0.0.jar
- org.javahispano.osgi.holamundoconsumer_1.0.0.jar

Nuestro config.ini tendría este aspecto:

```
osgi.clean=true
eclipse.ignoreApp=true
osgi.bundles=org.javahispano.osgi.holamundo_1.0.0.jar@start, \
org.javahispano.osgi.holamundoservice_1.0.0.jar@start, \
org.javahispano.osgi.holamundoconsumer_1.0.0.jar@start, \
```

Listado-3.1 config.ini

El directorio por defecto donde buscará los bundles indicados en el config.ini será EQUINOX_HOME/plugins, si por ejemplo queremos que nuestros bundles estén en la carpeta EQUINOX_HOME/mis bundles, el config.ini cambia un poco:

```
osgi.clean=true
eclipse.ignoreApp=true
osgi.bundles=../misbundles/org.javahispano.osgi.holamundo_1.0.0.jar@start, \
../misbundles/org.javahispano.osgi.holamundoconsumer_1.0.0.jar@2:start,\
../misbundles/org.javahispano.osgi.holamundoservice_1.0.0.jar@start, \
```

Listado-3.2 config.ini II

Aprovechando este segundo ejemplo del config.ini, hemos introducido los niveles de arranque de los bundles. En osgi podemos clasificar el orden de arranque de los bundles, permitiéndonos también iniciar varios bundles en un mismo nivel de arranque. En este caso nos conviene arrancar el bundle consumer después de que el bundle servicio este arrancado.

Cuando lancemos de nuevo la plataforma, realizaremos un listado de los bundles y aparecerá algo similar a esto:

```
osgi> ss

Framework is launched.

id      State       Bundle

0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827

1       ACTIVE      org.javahispano.osgi.holamundo_1.0.0

2       ACTIVE      org.javahispano.osgi.holamundoservice_1.0.0

3       ACTIVE      org.javahispano.osgi.holamundoconsumer_1.0.0

osgi>
```

El último comando que vamos a ver es DIAG, este comando nos permitirá diagnosticar el estado de las dependencias de un bundle. Si de nuestro ejemplo quitamos el bundle helloService del config.ini, el consumer no arrancará:

```
osgi> ss

Framework is launched.

id      State       Bundle

0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827

1       ACTIVE      org.javahispano.osgi.holamundo_1.0.0

2       INSTALLED   org.javahispano.osgi.holamundoconsumer_1.0.0

osgi>
```

Si en la consola escribimos "diag 2", se mostrarán las dependencias que están fallando del bundle 2:

```
osgi> diag 2

initial@reference:file:org.javahispano.osgi.holamundoconsumer_1.0.0.jar/ [2]

  Direct constraints which are unresolved:

    Missing imported package
org.javahispano.osgi.holamundoservice.service_0.0.0
```

3.1.2 **Apache Felix**

Para descargar esta implementación de Osgi, accederemos a la siguiente URL:

http://felix.apache.org/site/downloads.cgi

Al igual que en el caso de equinox, se nos presentan diferentes tipos de descarga:

- Descarga de la instalación mínima de Apache Felix. (Felix Framework Distribution)
- Descarga de las diferentes implementaciones de los servicios osgi y otros servicios de Felix.

Para comenzar descargaremos la distribución básica. Una vez la tengamos, simplemente la descomprimiremos obteniendo un sistema de archivos similar al siguiente:

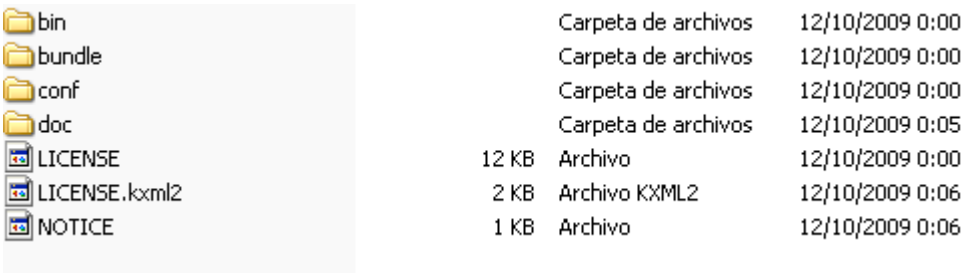

Figura-3.3 Estructura de directorios Apache Felix

Dentro del directorio bin encontraremos el jar: felix.jar, que será el encargado de arrancar la plataforma.

La carpeta "bundle" es el directorio donde depositaremos nuestros bundles. (Ejemplo, nuestro bundle hola mundo).

En el directorio conf, encontraremos el fichero config.properties, que es similar al config.ini de equinox.

Para arrancar la plataforma ejecutamos:

java -jar bin/felix.jar

Seguidamente aparecerá la consola de Felix:

```
Welcome to Felix
==============
→
```

3.1.2.1 Repositorios de Apache Felix

En el directorio "bundle" comentado anteriormente, podemos encontrar tres bundles que forman parte de la instalación por defecto de la plataforma. Dos de ellos son para la consola y uno más que incluye servicios osgi para acceder a los repositorios de bundles.

¿Que es el servicio de acceso al repositorio de Felix? Se trata de un servicio que facilita la instalación de bundles que se encuentran en repositorios remotos desde la ventana de comandos. El comando que se usa para acceder a estos repositorios es OBR. Por ejemplo si en la consola de Felix ejecutamos el comando "obr list-url" veremos los repositorios que tenemos registrados en la instalación. Por defecto nos viene registrado un único repositorio el de Apache Felix.

```
Welcome to Felix
==============
→ obr list-url
http://felix.apache.org/obr/releases.xml
→
```

También podemos listar los bundles del repositorio ejecutando el comando "obr list":

```
Welcome to Felix
==============
→ obr list-url
http://felix.apache.org/obr/releases.xml
→obr list
Apache Felix Bundle Repository (1.4.2, …)
```

```
Apache Felix Configuration Admin Service (1.0.4, …)
Apache Felix Declarative Services (1.0.8, …)
Apache Felix EventAdmin (1.0.0)
Apache Felix File Install (2.0.0)
Apache Felix Gogo Shell Commands (0.2.0)
...
```

Podéis consultar todos los comandos OBR en la página oficial de Apache Felix, concretamente en http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html.

3.1.2.2 Línea de comandos en Apache Felix

Al igual que en la sección de Equinox, en este apartado realizaremos las mismas prácticas sobre Apache Felix.

Comenzaremos realizando un listado de los bundles instalados, para ellos usamos el comando "ps":

```
Welcome to Felix
==============
→ ps
START LEVEL 1
   ID   State         Level   Name
[   0] [Active     ] [    0]  System Bundle (2.0.1)
[   1] [Active     ] [    1]  Apache Felix Bundle Repository (1.4.2)
[   2] [Active     ] [    1]  Apache Felix Shell Service (1.4.1)
[   3] [Active     ] [    1]  Apache Felix Shell TUI (1.4.1)
```

En esta ocasión podemos ver cuatro columnas:

- ID: Identificador único para cada bundle.
- State: Estado del bundle.
- Level: Nivel de arranque del bundle. Vemos como el bundle del sistema se arranca en nivel 0 y los demás se arrancan en nivel 1. Podemos especificar tantos niveles como queramos para arrancar nuestros módulos, también podemos arrancar varios en un mismo nivel.
- Name: Nombre del bundle.

A continuación vamos a instalar nuestro bundle "Hola Mundo", para ello utilizaremos el mismo comando usado para Equinox:

```
 Welcome to Felix
==============
→ install file:///bundleInstall/org.javahispano.osgi.holamundo_1.0.0.jar
Bundle ID: 4
→ ps
START LEVEL 1
    ID    State            Level    Name
[    0]   [Active      ]  [     0]   System Bundle (2.0.1)
[    1]   [Active      ]  [     1]   Apache Felix Bundle Repository (1.4.2)
[    2]   [Active      ]  [     1]   Apache Felix Shell Service (1.4.1)
[    3]   [Active      ]  [     1]   Apache Felix Shell TUI (1.4.1)
[    4]   [Installed   ]  [     1]   Holamundo Plug-in (1.0.0)
```

Si realizamos un listado de los bundles, veremos que el nuevo bundle se ha quedado como "installed", ahora solo tenemos que arrancarlo:

```
Welcome to Felix
==============
→ install file:///bundleInstall/org.javahispano.osgi.holamundo_1.0.0.jar
Bundle ID: 4
→ ps
START LEVEL 1
    ID    State            Level    Name
[    0]   [Active      ]  [     0]   System Bundle (2.0.1)
[    1]   [Active      ]  [     1]   Apache Felix Bundle Repository (1.4.2)
[    2]   [Active      ]  [     1]   Apache Felix Shell Service (1.4.1)
[    3]   [Active      ]  [     1]   Apache Felix Shell TUI (1.4.1)
[    4]   [Installed   ]  [     1]   Holamundo Plug-in (1.0.0)
→ start 4
HOLA MUNDO
→
```

Puede que dentro de nuestro proyecto tengamos tantos bundles que sea demasiado laborioso instalarlos de uno en uno desde la consola, para estos casos tenemos el directorio "bundle" donde podemos trasladar todos nuestros módulos para que se "auto desplieguen" en el arranque:

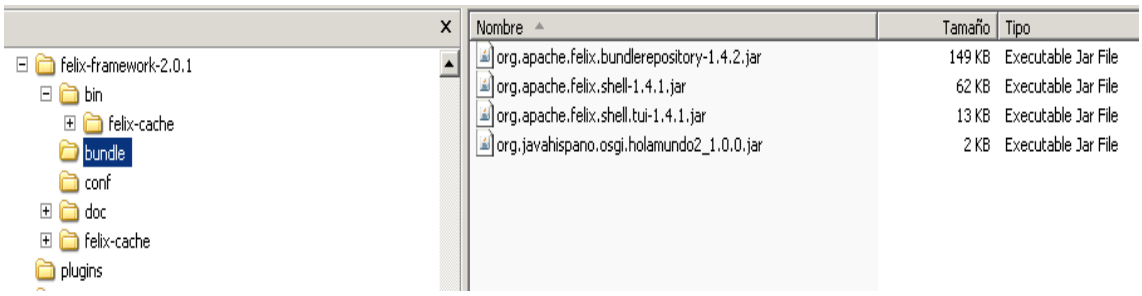

Figura-3.4 Directorio Bundle de Apache Felix

Si volvemos a listar los bundles, ahora nos aparecerá nuestro segundo bundle:

```
Welcome to Felix
==============
HOLA MUNDO
HOLA MUNDO
→ ps
START LEVEL 1
   ID   State         Level   Name
[   0] [Active     ] [    0]  System Bundle (2.0.1)
[   1] [Active     ] [    1]  Apache Felix Bundle Repository (1.4.2)
[   2] [Active     ] [    1]  Apache Felix Shell Service (1.4.1)
[   3] [Active     ] [    1]  Apache Felix Shell TUI (1.4.1)
[   4] [Active     ] [    1]  Holamundo Plug-in (1.0.0)
[   5] [Active     ] [    1]  Holamundo Plug-in  - 2(1.0.0)
```

3.1.2.3 Consola Web de Administración Felix

En ocasiones la administración de la plataforma mediante comandos puede ser un poco "dura", por ello, Apache Felix distribuye una utilidad Web para instalar en la plataforma y poder administrarla a través de un navegador web.

El único requisito o dependencia de esta utilidad es el servicio HTTP de OSGI. Para instalar este servicio, lo haremos a través del repositorio de Apache Felix mediante la utilidad OBR.

```
Welcome to Felix
==============
→ obr deploy “Apache Felix HTTP Service Jetty”
```

```
Target resource(s):
-----------------------
    Apache Felix HTTP Service Jetty (1.0.1)
Required Resource(s):
----------------------------
    OSGI R4 Compendium Bundle (4.0.0)
Deploying…done.
```

Si realizamos un listado de los bundles, veremos que se han instalado nuevos módulos, ahora solo tenemos que arrancarlos:

```
Welcome to Felix
==============
→ ps
START LEVEL 1
   ID   State          Level   Name
[   0] [Active     ] [    0]  System Bundle (2.0.1)
[   1] [Active     ] [    1]  Apache Felix Bundle Repository (1.4.2)
[   2] [Active     ] [    1] Apache Felix Shell Service (1.4.1)
[   3] [Active     ] [    1] Apache Felix Shell TUI (1.4.1)
[   4] [Active     ] [    1] Holamundo Plug-in (1.0.0)
[   5] [Active     ] [    1] Holamundo Plug-in – 2 (1.0.0)
[   6] [Installed  ] [    1] HTTP Service (1.0.1)
[   7] [Installed  ] [    1] OSGI R4 Compendium Bundle (4)
```

```
Welcome to Felix
==============
→ start 6 7
→ org.mortbay.log:Logging to org.mortbay.log  via org.apache.felix.http.jetty.
org.mortbay.log:Init SecureRandom.
org.mortbay.log:started org.mortbay.jetty.servlet.HashSessionIdManager…
…
org.mortbay.log: Started SelectChannelConnector@0.0.0.0:8080
….
```

Hemos arrancado los bundles 7 y 8, veremos que ahora tenemos el Apache Felix escuchando por el puerto 8080.

Una vez que tenemos instalado el servicio "HTTP", procedemos a instalar la consola, para ello:

```
Welcome to Felix
```

```
==============
→ install
http.//mirror.ibiblio.org/pub/mirrors/maven2/org/apache/felix/org.apache.felix.webco
nsole/1.0.0/org.apache.felix.webconsole-1.0.0.jar
Bundle ID:10
→ start 10
org.mortbay.log.started /system/console/*
org.mortbay.log:started /system/console/res
```

Finalmente, ya podemos acceder vía web a la administración de Felix, para ello accedemos a la siguiente url: http://localhost:8080/system/console/list

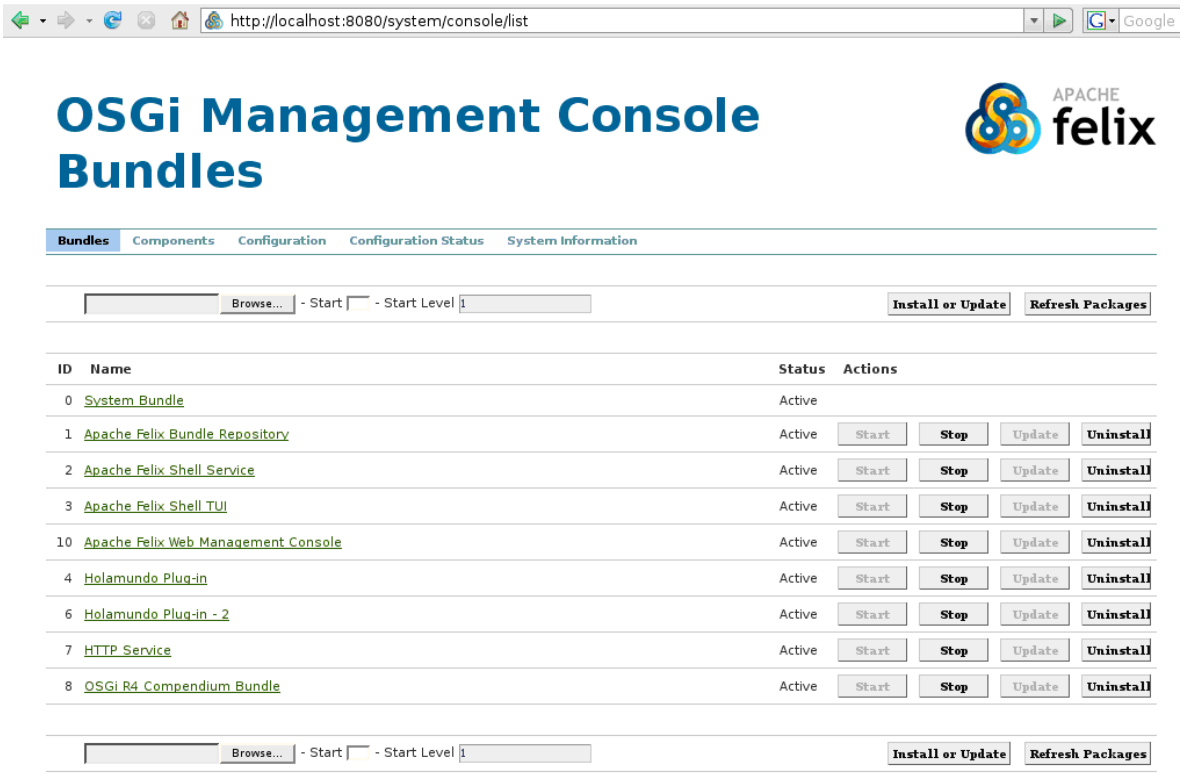

Figura-3.5 Consola de Administración Apache Felix

Podemos obtener más información acerca de la consola de administración desde la web: http://felix.apache.org/site/apache-felix-web-console.html.

Un último apunte en este apartado, es comentaros que podéis instalar esta consola en otros contenedores OSGI diferentes al Apache Felix, por ejemplo en equinox. Rebuscando por internet seguro que encontráis muchas referencias a ello.

3.1.3 **Knopflerfish**

Para descargarnos esta implementación accedemos a la página web del producto: http://www.knopflerfish.org. Desde el "home" vamos a la sección de descargas. Existen distintas versiones, pero es recomendable bajar la última versión estable.

Knopflerfish dispone de diferentes tipos de descarga, las tres descargas más importantes:

- Versión con los fuentes y documentación.
- Versión binaria completa sin fuentes ni documentación.
- Version binaria del core y algunos bundles, sin fuentes ni documentación.

Comenzamos por descargarnos la versión binaria completa, en mi caso knopflerfish_fullbin_osgi_2.1.1.jar.

AL ejecutar este jar con la maquina virtual java, se abrirá el wizard de instalación (lo típico...siguiente, siguiente y Finalizar...)

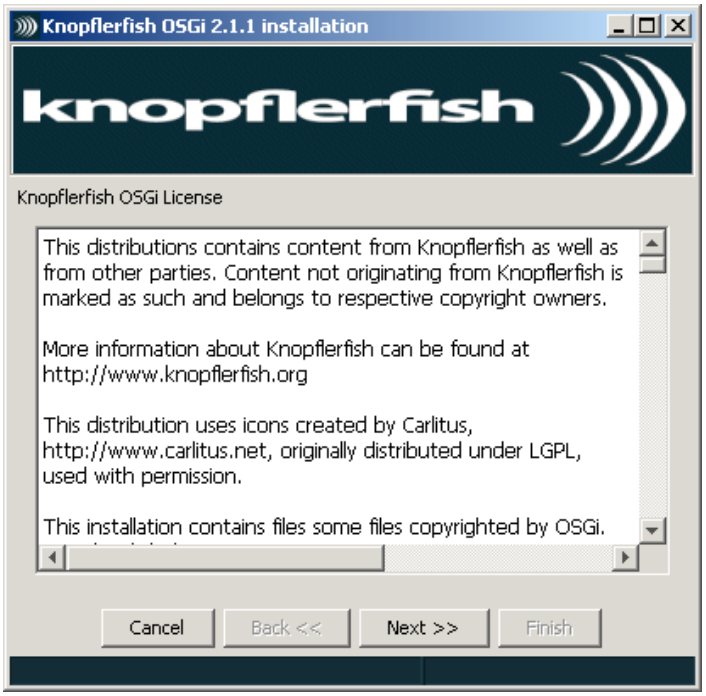

Figura-3.6 Instalación Knopflerfish

Figura-3.7 Instalación Knopflerfish II

Figura-3.8 Instalación Knopflerfish III

Una vez finalizada la instalación podremos ejecutar el jar: framework.jar, que se encuentra dentro del directorio de instalación, seguidamente se abrirá un consola totalmente grafica, desde la cual se administra la plataforma. Como se trata de un entorno muy grafico, lo explicaremos mediante una imagen:

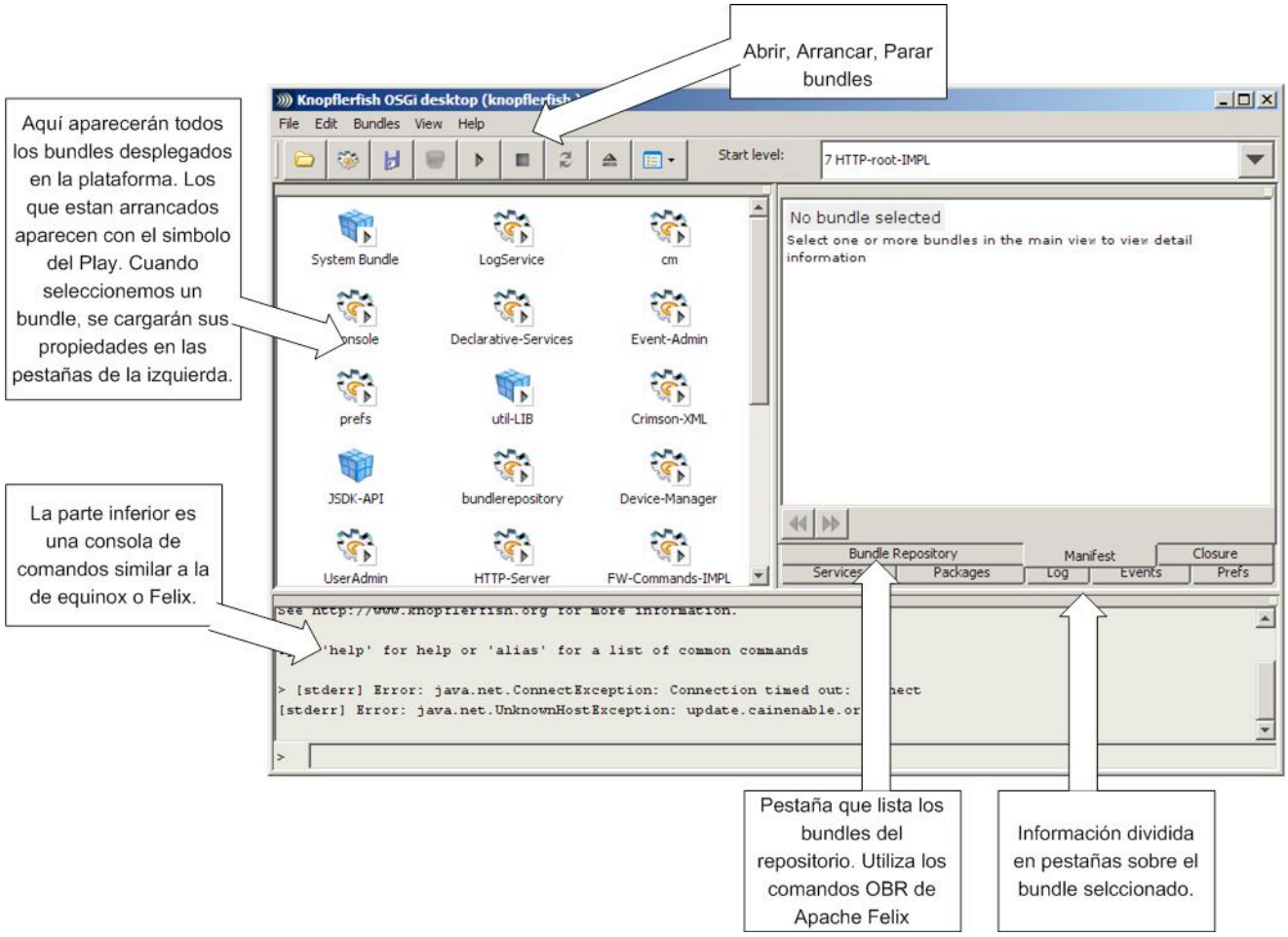

Figura-3.9 Consola de Administración Knopflerfish

3.2 **Entornos de desarrollo**

En la introducción hemos nombrado los entornos de desarrollo más comunes en ambientes OSGI. A continuación vamos a verlos más en detalle.

3.2.1 Eclipse

A mi parecer Eclipse es una de las herramientas más cómodas para trabajar con OSGI, ya que eclipse en si mismo está construido sobre la plataforma Equinox, e incorpora herramientas para el desarrollo, pruebas y ejecución de bundles. No es objeto de este

tutorial explicar cómo se trabaja con eclipse, en este caso solo nos centraremos en las herramientas que incorpora para el desarrollo de bundles.

3.2.1.1 Instalando Eclipse

Lo primero que vamos a hacer es descargarnos la herramienta de la página web de eclipse, pero en este caso nos bajaremos la version "RCP/Plug-ins developers": http://www.eclipse.org/home/categories/index.php?category=equinox.

Una vez lo tengamos instalado (para instalar solo hay que descomprimir), creamos nuestro nuevo espacio de trabajo y manos a la obra!

3.2.1.2 Creando un nuevo proyecto

Comenzaremos creándonos nuestro bundle "Hola Mundo", para ello seleccionamos File -> New -> Project y escogemos "Plug-in Project":

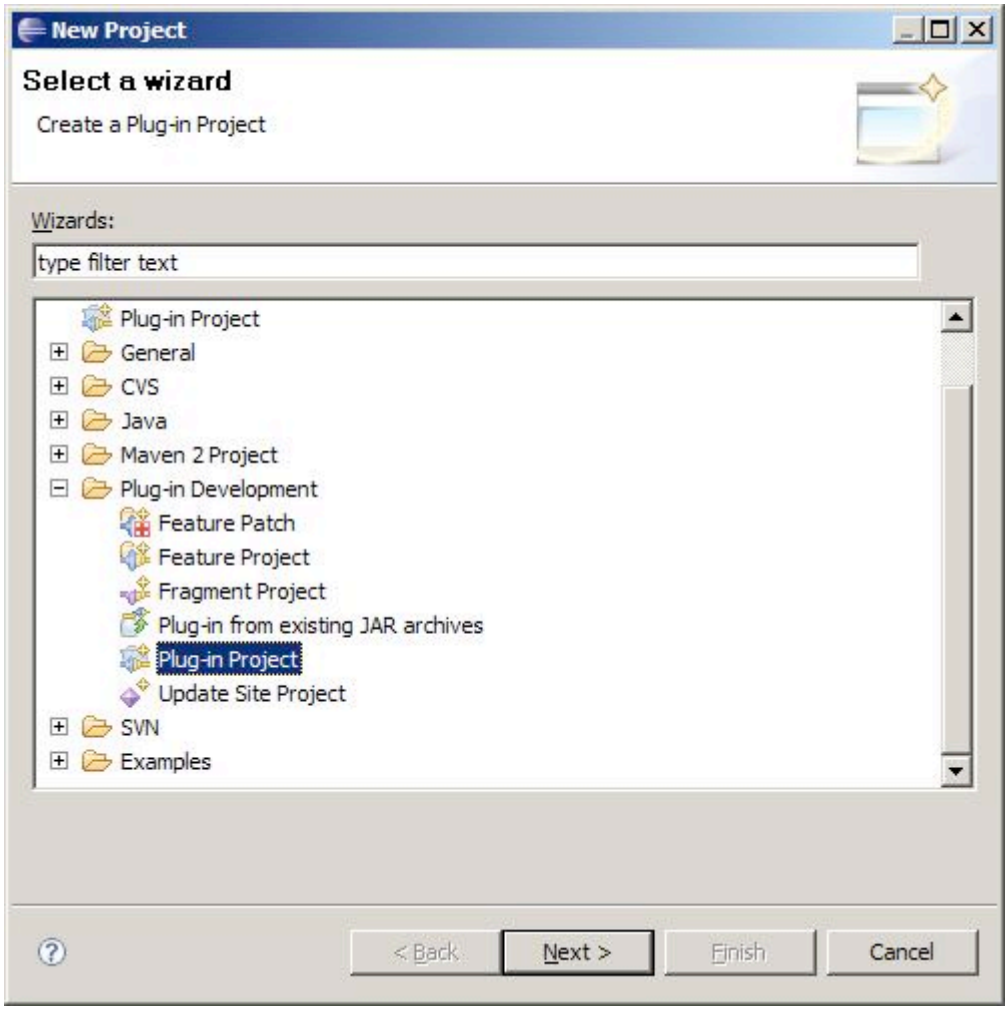

Figura-3.10 Nuevo Proyecto Eclipse

En la siguiente pantalla, le daremos un nombre al bundle, generalmente igual al paquete "base" de nuestro proyecto (org.javahispano.osgi.holamundo). En la parte inferior de la misma pantalla podremos seleccionar el entorno de ejecución del bundle. Elegiremos la versión de eclipse si estamos desarrollando plug-ins de eclipse, en caso de desarrollar módulos de Osgi, tendremos la opción de desarrollar en OSGI estándar o específico para la plataforma equinox.

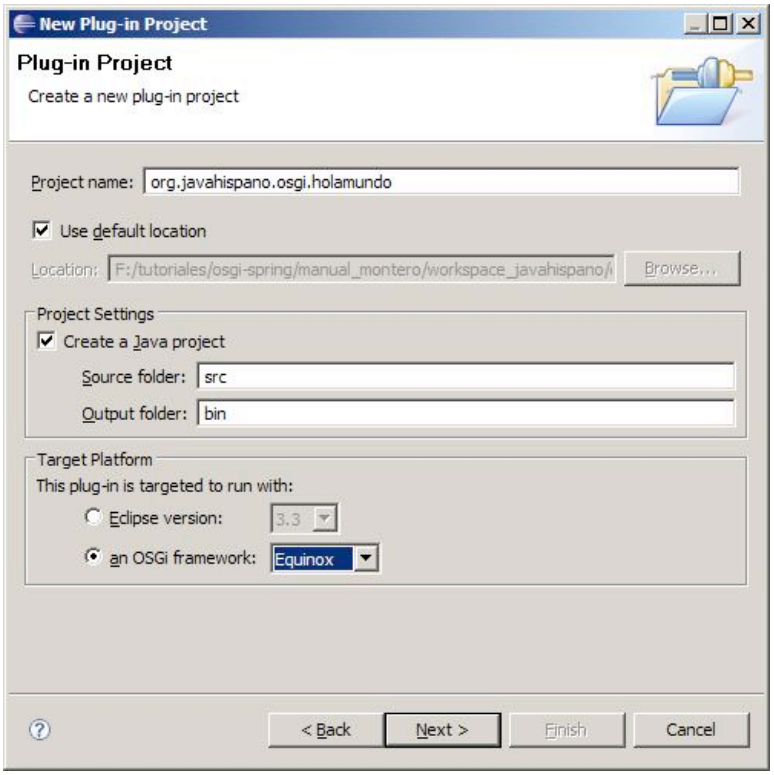

Figura-3.11 Nuevo Proyecto Eclipse II

En la siguiente pantalla se configura el nombre o id del bundle y otros atributos como la version o el nombre del proveedor. También permitirá darle nombre a la clase activadora del bundle (El Activator con sus métodos Start y Stop).

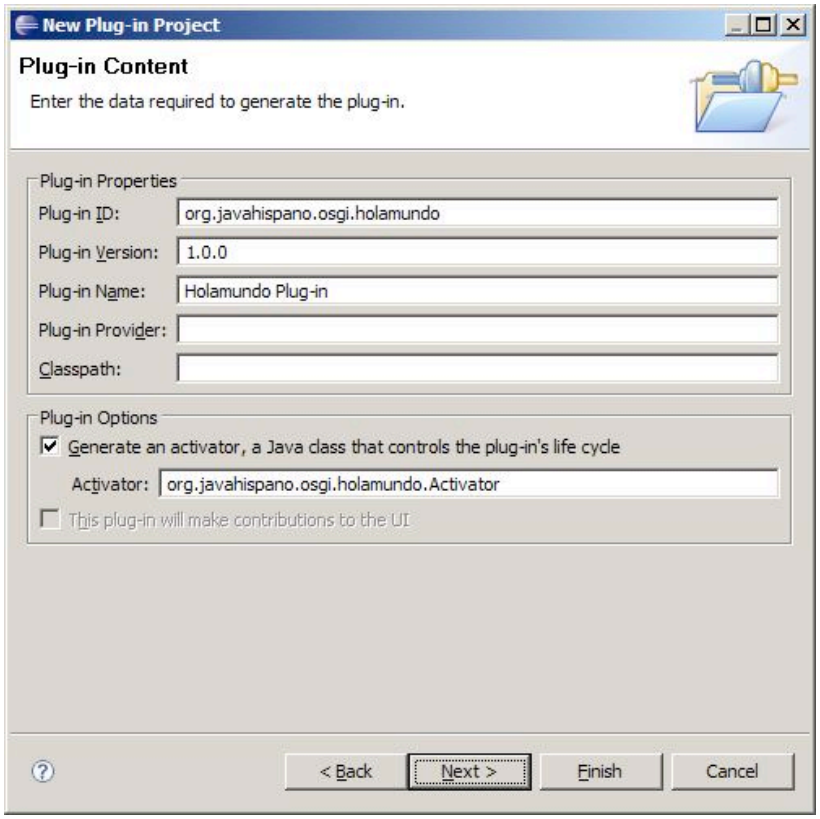

Figura-3.12 Nuevo Proyecto Eclipse III

En la última pantalla del wizard, podemos crear nuestro proyecto a partir de unas plantillas de ejemplo, muy útil para principiantes, por ello seleccionaremos la plantilla "Hello OSGI Bundle":

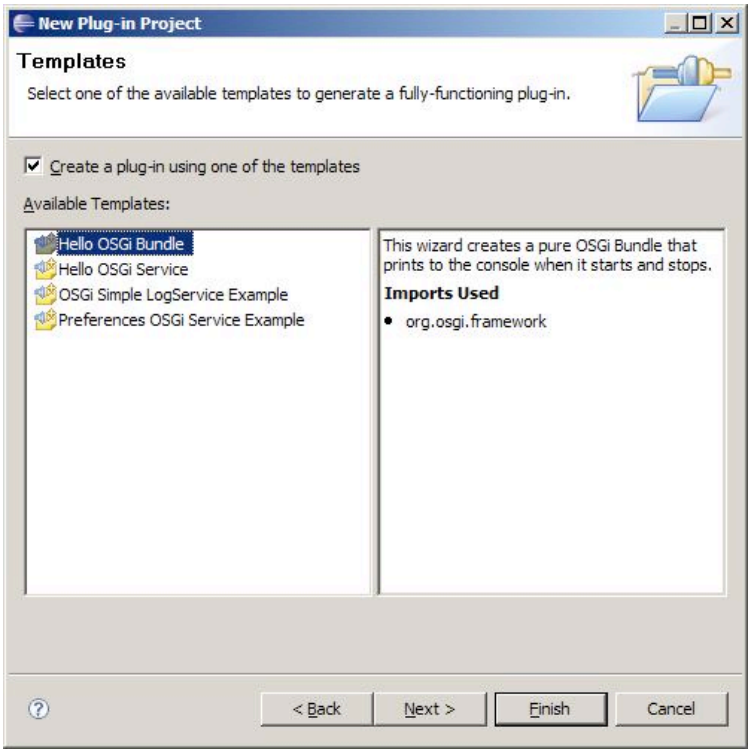

Figura-3.13 Nuevo Proyecto Eclipse IV

Al seleccionar este tipo de template se habilita una nueva pantalla, donde se puede especificar los mensajes ("System.out") que se mostraran al arrancar y parar el bundle.

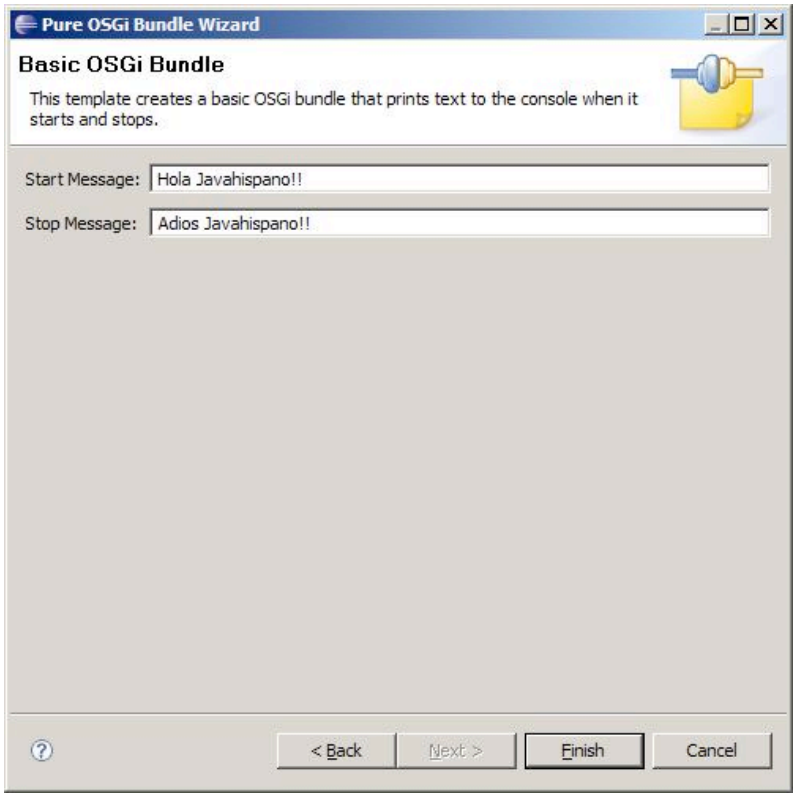

Figura 3.14 Nuevo proyecto Eclipse V

Al pulsar Finish ya tendremos nuestro bundle listo para ejecutarse. La clase activator se habrá creado con los mensajes especificados en la última pantalla del wizard.

3.2.1.3 Manifest File Editor

Una utilidad interesante de Eclipse, es el editor de archivos manifest.mf, desde la cual podemos modificar este fichero descriptor de manera grafica.

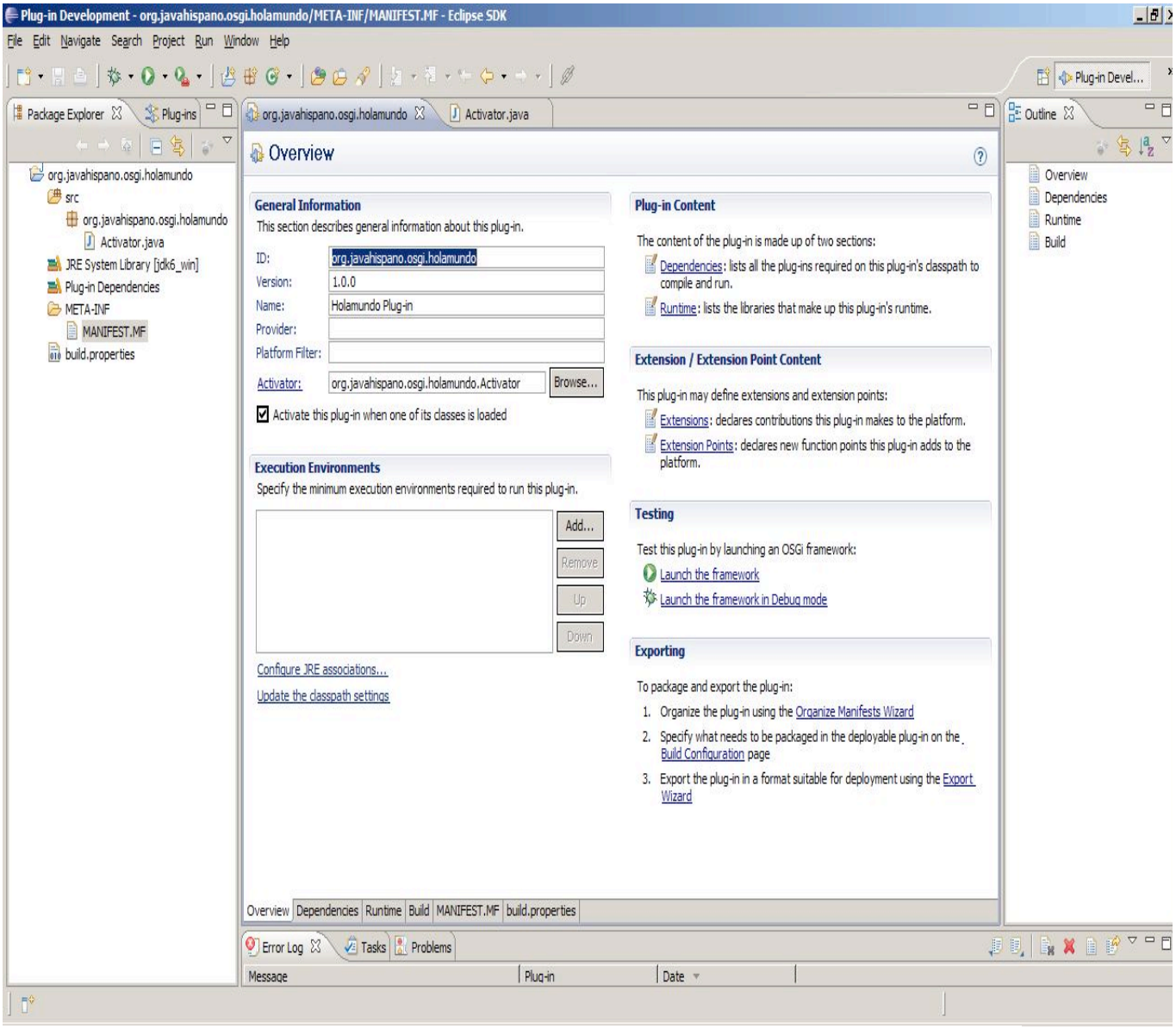

Figura 3.15 Manifest Editor I

Este editor se compone de seis pestañas:

- Overview
- Dependencies
- Runtime
- Build
- MANIFEST.MF
- build.properties

Solapa Overview:

En la primera pestaña "Overview", podemos diferenciar diferentes zonas:

- General Information: Muestra las cabeceras básicas del manifest: Id, nombre, proveedor, versión..., además del nombre de la clase Activator.
- Execution Environmnets: Versión mínima de la JVM sobre la que funcionará el bundle.
- Plugin Content: Enlaces a las pestañas de dependencias y runtime.
- Extension / Extension Point Content: Elementos específicos para el desarrollo de plug-ins de Eclipse.
- Testing: Utilidades para lanzar el bundle en modo debug. Esto nos permitiría especificar puntos de corte en nuestro código para poder depurarlo.
- Exporting: Herramientas de eclipse para empaquetar los bundles.

Solapa Dendencies:

La siguiente pestaña es la de dependencias, desde la cual podremos controlar las dependencias tanto en ejecución como en desarrollo:

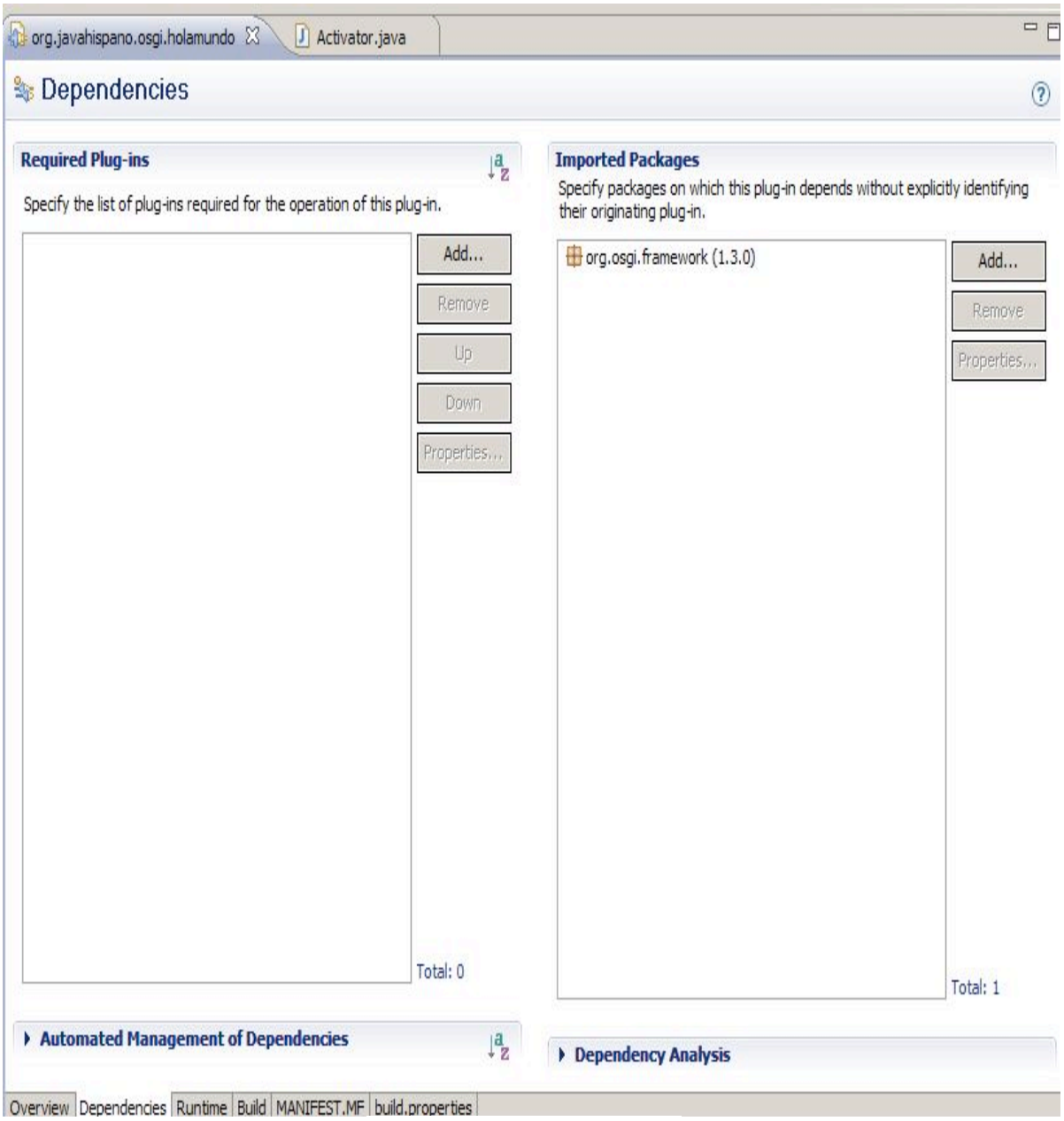

Figura-3.16 Manifest Editor II

Podremos diferenciar las siguientes áreas dentro esta pestaña:

- **Required Plug-ins:** equivale a las cabeceras Required-bundle comentadas en la I sección de este documento.
- **Imported Packages:** Nos permite importar paquetes exportados por otros bundles.
- **Automated Management of Dependencies:** Con esta opción eclipse analizará las dependencias del bundle y las incluirá automáticamente en el manifest.mf.
- **Dependency Analysis:** Utilidad que ayuda a comprender mejor las dependencias del bundle, también nos ayudará a optimizarlo.

Solapa Runtime:

Esta pestaña permite modificar configuraciones del bundles, que entrarán en vigor cuando dicho módulo este en ejecución.

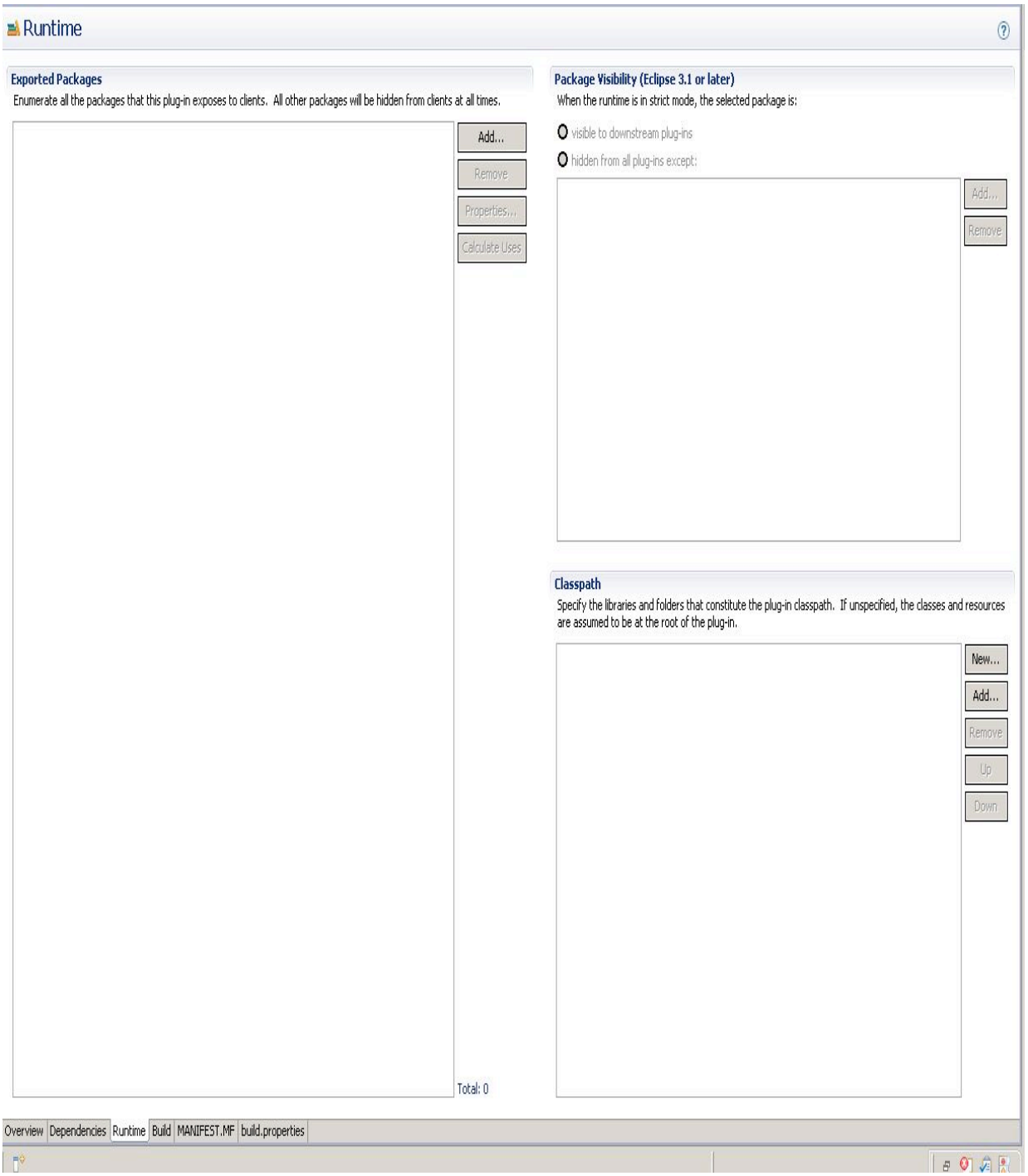

Figura-3.17 Manifest Editor III

Podemos diferenciar tres secciones:

- **Exported Packages:** Aquí añadiremos los paquetes de nuestro bundle que queremos que estén visibles por otros bundles que estén corriendo sobre el mismo framework.
- **Package Visibility:** Se trata de una utilidad para crear plugins para eclipse, por ello no entraremos en detalles.
- **Classpath:** Aquí incluiremos las librerías necesarias para que el bundle se ejecute. Por ejemplo, nuestro bundle puede necesitar de la librería de Hibernate para ejecutarse, para ello, dentro de la estructura de nuestro bundle crearemos una carpeta lib y copiaremos en ella la librería de Hibernate, posteriormente volveremos a esta pestaña y añadiremos la librería al manifest.mf:

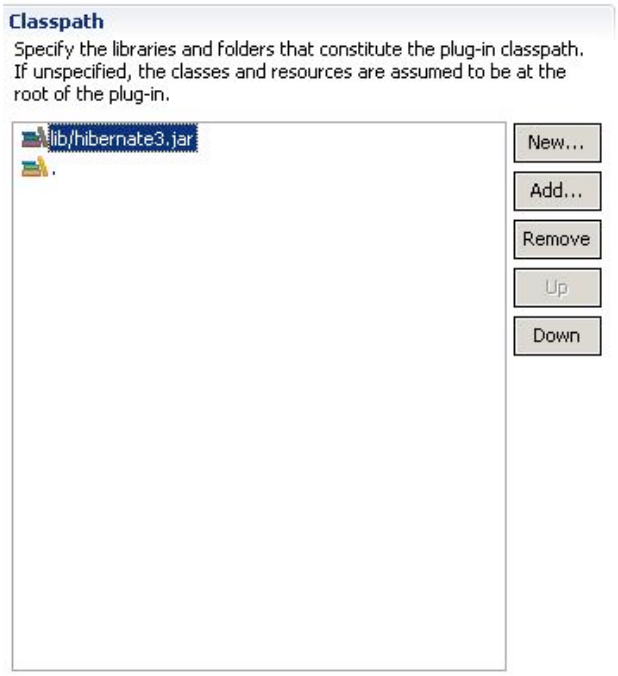

Figura-3.18 Manifest Editor IV

Solapa Build

En esta pestaña configuraremos como queremos que eclipse empaquete nuestro bundle

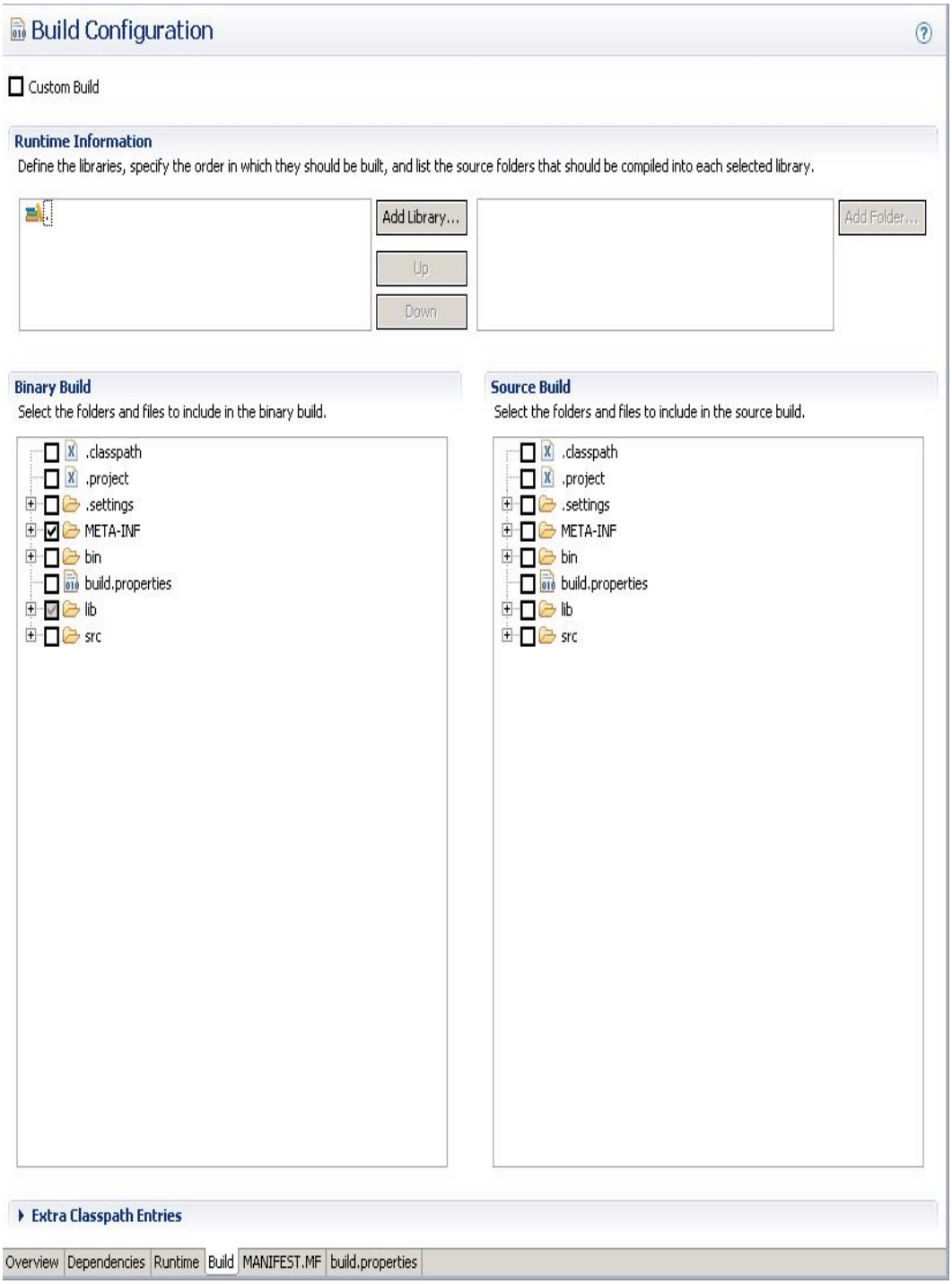

Figura-3.19 Manifest Editor V

Solapa MANIFEST.MF

Desde esta pestaña podremos editar el manifest.mf en modo texto.

Solapa build.properties

Desde aquí se configura la ubicación de los fuentes del proyecto y donde se dejaran los binarios generados en tiempo de compilación

3.2.1.4 Ejecutando nuestros bundles

Una vez llegado a este punto, tenemos nuestro bundle creado en el Workspace, la última acción seria ejecutarlo dentro del entorno de ejecución de Eclipse.

Antes de intentar ejecutar el bundle, configuraremos el entorno desde Window --> Preferences:

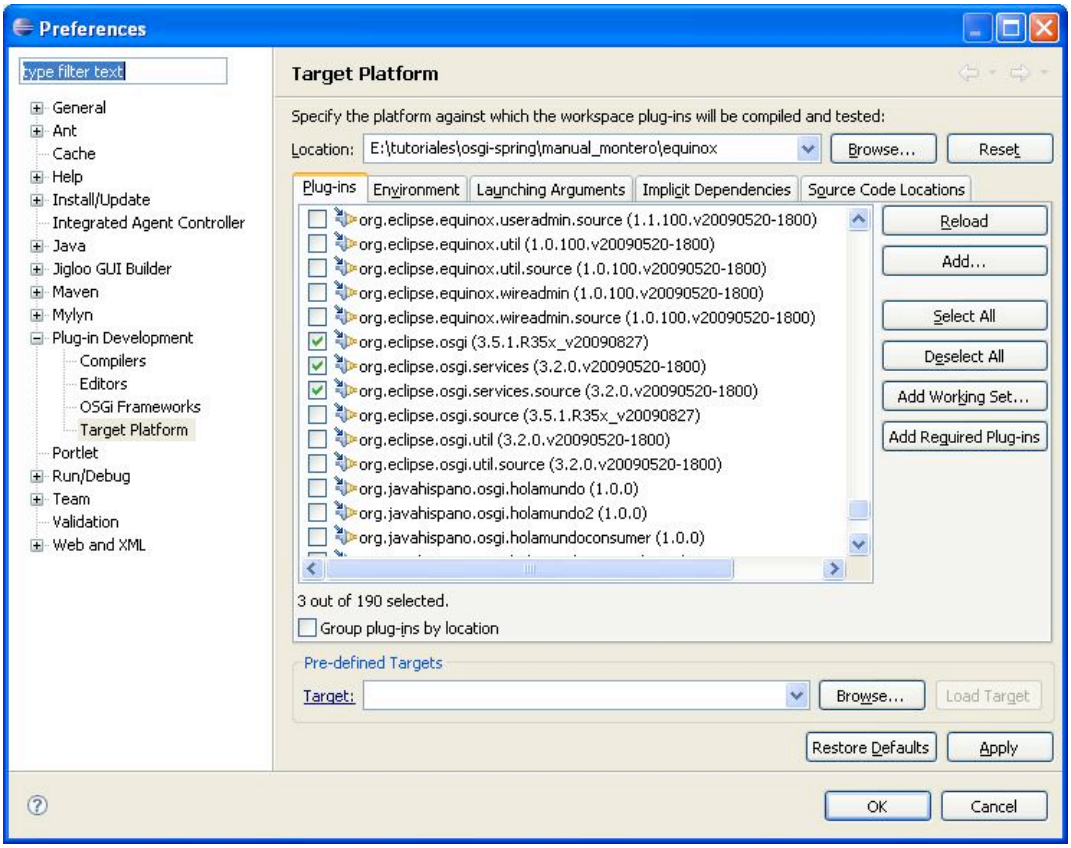

Figura 3.20 Ejecución de Equinox en Eclipse

Desde la opción Plug-In Development, podremos configurar el entorno de ejecución y el entorno de compilación de nuestros bundles del Workspace (es decir todos lo bundles que añadamos al entorno de ejecución se configuran en el classpath del entorno de desarrollo).

En la parte superior de la pantalla, podremos indicarle donde se encuentra la plataforma Equinox, por defecto aparecerá la carpeta Eclipse donde estamos ejecutando el ID (podríamos decir que eclipse tiene empotrado un entorno Equinox). A mí como no me gusta mezclar las cosas, he configurado otro entorno de Equinox. He utilizado una instalación limpia de Equinox, y en el campo Location de la pantalla de preferencias, le indicado el directorio raíz de mi instalación de Equinox.

En la pestaña de plugins, deseleccionamos todos los bundles e iremos añadiendo poco a poco los plugins necesarios para nuestro desarrollo, como por ahora lo que queremos ejecutar es un simple "hola mundo", solo dejamos marcados los tres bundles de la imagen. Cuando pulsemos OK, ya tendremos nuestro entorno de ejecución configurado.

Para ejecutar nuestro bundle, pulsamos con el botón derecho sobre el proyecto --> Run as --> OSGI framework:

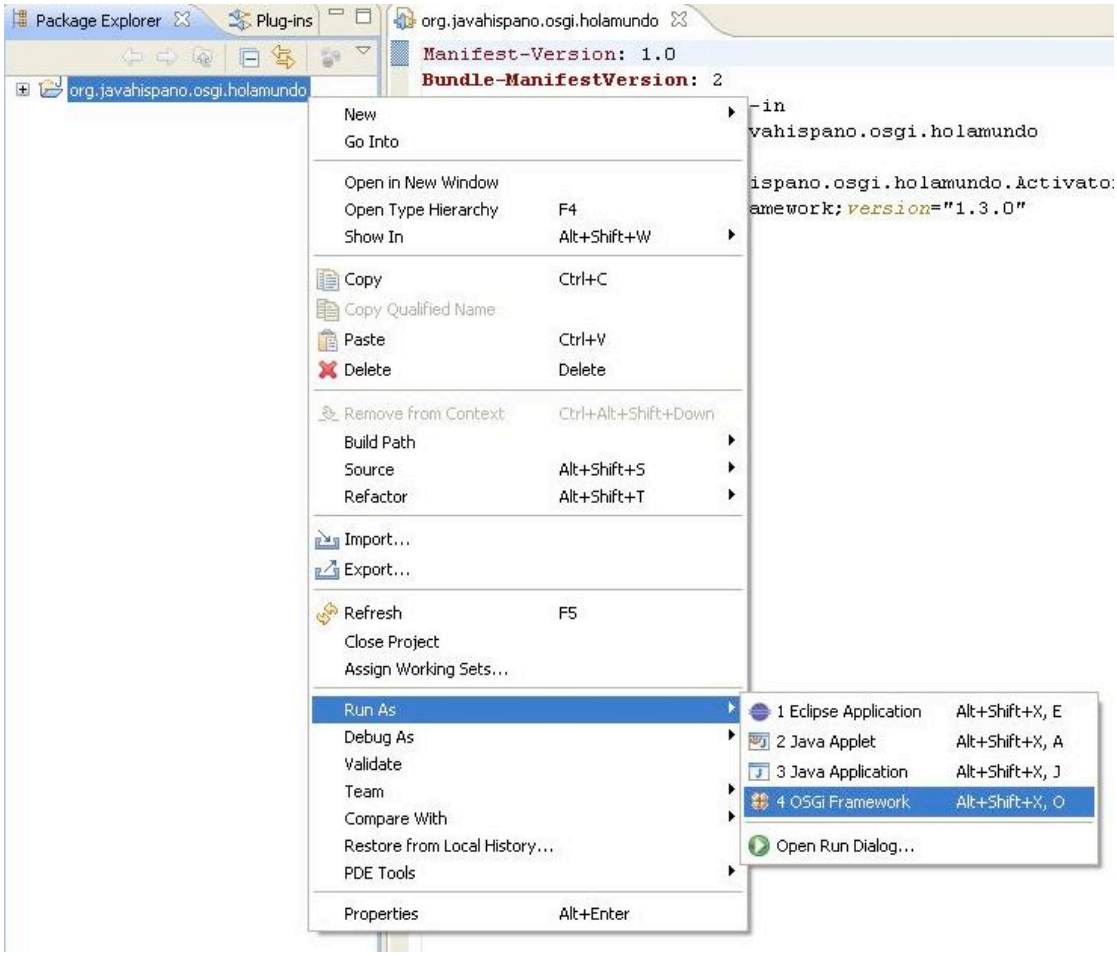

Figura-3.21 Ejecución de Equinox en Eclipse II

La consola de Equinox y la salida producida por nuestro bundle se mostrará en la parte inferior de la pantalla:

Figura-3.22 Ejecución de Equinox en Eclipse III

3.2.2 Maven: Pax Contructor Plugin

Como hemos comentado anteriormente existen ciertas similitudes en la gestión de dependencias que realiza maven, con la gestión de dependencias que viene realizando OSGI.

No es objetivo de este tutorial detallar el funcionamiento de la herramienta maven, ni realizaré una pequeña introducción acerca de esta increíble herramienta, por ello si desconocéis por completo maven, antes de seguir con este tutorial, os podéis informar en la misma WIKI, desde la cual podréis acceder a multitud de enlaces relacionados: http://es.wikipedia.org/wiki/Maven. También podéis consultar algunos de los siguientes libros sobre el tema (algunos de ellos más actualizados que otros...):

- **Apache Maven 2 Effective Implementation**
- ***Maven: A Developer's Notebook***
- **Maven: The Definitive Guide**
- **Professional Java Tools for Extreme Programming: Ant, XDoclet, JUnit, Cactus, and Maven**

"Pax Contructor" es un plugin de Maven que nos facilita enormemente desde la creación, gestión y compilación de bundles hasta su propio despliegue en algunas de las plataformas de ejecución descritas en el punto anterior.

Este plugin es uno de tantos proyectos que envuelven el proyecto PAX. Este proyecto incluye subproyectos que implementan utilidades para el desarrollo de bundles OSGI.

En este tutorial nos centraremos en el PAX Constructor, pero debéis saber que existen otros proyectos dentro de esta rama bastante interesantes:

- **Pax Exam:** Extensión de JUnit para realizar pruebas sobre bundles de osgi.
- **Pax Logging:** Una implementación de "OSGI Logging Service"
- **Pax Runner:** Utilidad para comenzar con el framework de OSGI que incluye una serie de bundles predefinidos ya instalados.
- **Pax ConfMan:** Una implementación de "OSGI Configuration Admin Service".
- **Pax LDAP Server:** Proporciona un servidor LDAP basado en OSGI.
- **Pax Shell:** Proporciona un entorno de comandos para OSGI.

Podéis ver todos los proyectos de PAX en la siguiente URL:

http://wiki.ops4j.org/display/ops4j/Pax

3.2.2.1 Instalando Pax-Construct

Lo primero que haremos para comenzar a trabajar con este plugin, será lógicamente instalarlo en nuestro PC, para ello nos descargaremos el conjunto de script que componen esta utilidad desde la página: http://wiki.ops4j.org/display/paxconstruct/Download.

Una vez descargado, lo descomprimiremos en nuestra carpeta de trabajo. Podremos observar que este paquete se compone de una serie de scripts ejecutables que explicaremos a continuación:

- **pax-add-repository** Añade un repositorio Maven a la lista de repositorios del proyecto.
- **pax-clone** Clona un proyecto Pax-Constructor.
- **pax-create-bundle** Crea un nuevo bundle dentro de un proyecto OSGI.
- **pax-create-module** Crea un nuevo modulo dentro de otro modulo o dentro de un proyecto OSGI. Pax Contructor considera un modulo como una entidad de organización que agrupa bundles relacionados entre sí.
- **pax-create-project** Crea un nuevo proyecto principal OSGI. Este será el primer script que utilizaremos para iniciar nuestro proyecto.
- **pax-embed-jar** No ayuda a insertar ficheros JAR dentro de un bundle añadiendo las referencias al classpath en su manifest.mf y sus ficheros maven de proyecto.
- **pax-import-bundle** Añade directivas Import-Package a nuestro bundle.
- **pax-move-bundle** Permite mover un bundle a un nuevo directorio, actualizando los ficheros POM de Maven.
- **pax-provision** Arranca el framework OSGI, por defecto Apache Felix. Instala todos los bundles asociados al proyecto, así como sus dependencias. Una vez correctamente instalados, comenzará la inicialización de cada uno de ellos.
- **pax-remove-bundle** Elimina un Bundle de un proyecto OSGI.
- **pax-update** Actualiza un Pax Construct.
- **pax-wrap-jar** Construye un proyecto bundle a partir de una librería de tercero.

Antes de empezar, deberemos asegurarnos que tenemos correctamente instalado Maven en nuestro equipo, ya que como hemos mencionado anteriormente, estos script "corren" sobre maven.

Comenzaremos ejecutando el comando "pax-create-project" con el cual crearemos nuestro proyecto para este tutorial. Este proyecto podrá albergar subproyectos que corresponderán a los bundles que vayamos creando. Al ejecutar este script se piden los siguientes parámetros:

- GroupId: Es un concepto de Maven. Podríamos definirlo como una forma de agrupar artefactos bajo una denominación común, bajo una identificación de grupo que es el groupId. El nombre suele ser el nombre de la organización, el proyecto o el equipo de trabajo o bien la raíz del package de java que contendrá las clases del artefacto, ej: com.xyz.foo. En nuestro caso usaremos el group-id: org.javahispano.osgi

- ArtifactId: Es un concepto de Maven. Podríamos definirlo como el identificador de un proyecto concreto dentro de los proyectos que pertenecen al mismo groupId. Este término se utiliza para denominar a la unidad mínima que maneja Maven en su repositorio. Puede ser por ejemplo un jar, un ear, un zip, etc. En nuestro ejemplo el artifactId será: tutorial.
-
- Versión: Versión de nuestro proyecto. Comenzaremos por la 1.0.0

Una vez ejecutado este comando ya tendremos creada una estructura de proyecto similar a la siguiente:

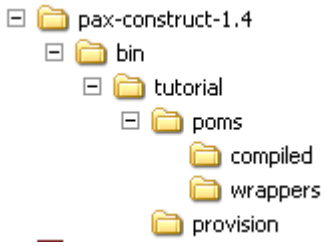

Figura 3.23 Estructura de directorios pax-construct

Navegando por este conjunto de directorios, podemos encontrarnos con diferentes archivos pom.xml. Según los conceptos de maven, un fichero POM (Project Object Model), es un fichero XML que representa un proyecto maven. Los pom que nos encontraremos en esta fase inicial son:

- POM de nivel superior, albergado en la carpeta "tutorial". De momento será el más importante para nosotros, a partir de este POM construiremos el proyecto.

- Los archivos POM ubicados bajo el directorio "poms", servirán como padres de los subproyectos de tipo bundle que crearemos mas tarde.

- El fichero POM de la carpeta "provision", lo usaremos para seleccionar los bundles que queremos desplegar dentro de la plataforma OSGI.

3.2.2.2 Configurando nuestro proyecto con Pax-Construct

Es evidente que maven nos permite multitud de configuraciones a través del archivo POM, tales como tareas de compilación, despliegue o pruebas unitarias.

De momento comentaremos un par de configuraciones muy interesantes que nos presenta el plugin de PAX Contructor:

Si abrimos el pom.xml del directorio principal de nuestro proyecto veremos algo similar a:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://maven.apache.org/POM/4.0.0"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <!-- | put company details here
  <organization>
    <name>...</name>
    <url>http://...</url>
  </organization>
  -->
  <!-- put license details here
  <licenses>
    <license>
      <name>...</name>
      <url>http://...</url>
      <comments>
      </comments>
    </license>
  </licenses>
  -->
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.javahispano.osgi</groupId>
  <artifactId>tutorial</artifactId>
  <version>1.0.0</version>
  <name>org.javahispano.osgi.tutorial (OSGi project)</name>

  <!--    describe your project here  -->
  <description>Generated using Pax-Construct</description>
```

```
 <properties>
  <!-- some example OSGi runtime properties  -->
  <org.osgi.service.http.port>8080</org.osgi.service.http.port>
  <org.osgi.service.http.port.secure>8443</org.osgi.service.http.port.secure>
 </properties>
 <packaging>pom</packaging>
 <modules>
  <module>poms</module>
  <module>provision</module>
 </modules>
 <build>
  <plugins>
   <plugin>
    <groupId>org.ops4j</groupId>
    <artifactId>maven-pax-plugin</artifactId>
    <version>1.4</version>
    <configuration>
     <!--
      | some example Pax-Runner settings
     -->
     <provision>
      <param>--platform=felix</param>
     </provision>
    </configuration>
    <executions>
     <!--
      | uncomment to auto-generate IDE files
     <execution>
      <id>ide-support</id>
      <goals>
       <goal>eclipse</goal>
      </goals>
     </execution>
     -->
    </executions>
   </plugin>
  </plugins>
 </build>
</project>
```

Listado 3.3 pom.xml

Por ejemplo, lo primero que haremos es cambiar el entorno de ejecución de nuestros futuros bundles. Si nos fijamos en el pom.xml anterior, podemos ver que por defecto está configurado para la ejecución sobre la plataforma FELIX:

```
<provision>
    <param>--platform=felix</param>
</provision>
```

Como uno de mis entornos de ejecución OSGI preferidos es equinox, lo cambiaremos modificando las líneas anteriores por las siguientes:

```
<provision>
    <param>--platform=equinox</param>
    <param>--profiles=minimal</param>
</provision>
```

Cuando en este tutorial presentamos el IDE Eclipse, os dije que este IDE es uno de mis entornos de desarrollo favoritos, es por ello que a través de Pax Contruct intentaremos que los bundles generados sean proyectos compatibles con eclipse, para ello descomentaremos las líneas:

```
<!--
    | uncomment to auto-generate IDE files
    <execution>
      <id>ide-support</id>
      <goals>
        <goal>eclipse</goal>
      </goals>
    </execution>
    -->
```

3.2.2.3 Crear bundles con Pax Construct

Como hemos mencionado anteriormente, para crear un nuevo bundle utilizaremos el script pax-create-bundle. Este script nos ayudará a crear el esqueleto de nuestro nuevo modulo y debe ser ejecutado desde dentro del directorio de trabajo creado al ejecutar la tarea "pax-create-project" comentada anteriormente.

Al ejecutar este script, nos pedirá una serie de parámetros:

```
pax-create-bundle -p package [-n bundleName] [-g bundleGroupId] [-v version] [-o] [-- mvnOpts ...]

 package (org.example.pkg) ? com.javahispano.osgi.holamundo

bundleName () ? com.javahispano.osgi.holamundo

bundleGroupId () ? com.javahispano.osgi

version (1.0-SNAPSHOT) ? 1.0-SNAPSHOT
```

Una vez ejecutado el comando, habremos creado nuestro primer bundle con Pax Construct!. (Si nos hemos equivocado en alguno de los pasos, en el nombre del bundle o en su configuración, este será un buen momento para borrarlo y volverlo a crear con el script pax-remove-bundle).

Dentro de la carpeta del proyecto, se debería haber creado un nuevo directorio llamado com.javahispano.osgi.holamundo. Esta carpeta contendrá los fuentes del nuevo bundle cuya estructura interna se corresponderá con la arquitectura de proyectos maven. Por lo tanto dentro del directorio del nuevo bundle encontraremos:

- **pom.xml:** Archivo de proyecto maven. Si lo editamos podremos ver que depende del pom.xml que se encuentra dentro de la carpeta "tutorial" (Depende del proyecto padre que hemos creado con el script pax-create-project).

- **osgi.bnd:** Archivo similar al manifest.mf de OSGI. En vez de escribir las directivas de OSGI sobre el archivo manifest.inf, las escribiremos sobre este fichero con algunas peculiaridades que comentaremos más adelante.

- **src/main/java:** A partir de este directorio comenzarán los paquetes java de nuestro proyecto.

- **src/main/resources:** En este directorio ubicaremos los recursos de nuestro nuevo bundle. De momento no lo usaremos.

3.2.2.3.1 BND - Bundle Tool

La herramienta bnd, se trata de una utilidad opensource que nos ayuda a crear y diagnosticar bundles. Sus principales funcionalidades son:

- Mostrar el fichero manifest.inf y el contenido de un jar.
- Crear bundles osgi a partir de un jar
- Verificar las entradas del manifest.inf

Se puede trabajar con esta herramienta con alguna de las siguientes "interfaces":

- Línea de comando
- Plugin de Eclipse
- Plugin Maven
- Plugin Ant

En nuestro caso, Pax Construct utiliza esta herramienta BND a través del plugin de Maven. Entre otras operaciones que realiza, destacamos que en vez de escribir las directivas directamente sobre el manifest.inf, las escribiremos sobre el fichero osgi.bnd, que en tiempo de compilación será parseado y convertido en manifest.mf. A pesar que el formato del fichero es muy similar al manifest.mf, existen algunas diferencias. Podéis encontrar la especificación del formato de este fichero en la siguiente URL: http://www.aqute.biz/Code/Bnd#format.

3.2.2.4 Despliegue bundles con Pax Construct

En la sección anterior habíamos creado un nuevo bundle dentro de nuestro proyecto, ahora modificaremos las clases de ejemplo autogeneradas, las compilaremos, empaquetaremos y desplegaremos, para finalmente conseguir nuestra salida favorita: Hola Mundo.

La clase autogenerada, a la que apunta la directiva bundle-activator la encontraremos en el siguiente directorio: [Espacio de Trabajo]/tutorial/com.javahispano.osgi.holamundo/src/main/java/com/javahispano/osgi/holamundo/internal/ExampleActivator.java.

Si editamos esta clase veremos que se trata de una clase que expone un servicio de ejemplo también autogenerado. Para este ejemplo, solamente añadiremos la línea "System.out.println("Hola Mundo");", en el método start de la clase.

Una vez realizados los cambios en el código fuente, procederemos a compilar y empaquetar el proyecto, a la vez de ubicar el proyecto compilado en el repositorio local de maven. Al tratarse de un proyecto maven usaremos el comando mvn install.

```
------------------------------------------------------------------------

[INFO] Reactor Summary:

[INFO] ------------------------------------------------------------------------

[INFO] org.javahispano.osgi.tutorial (OSGi project) ..... SUCCESS [1:00.822s]

[INFO] tutorial - plugin configuration ..... SUCCESS [0.037s]

[INFO] tutorial - wrapper instructions ..... SUCCESS [2.401s]

[INFO] tutorial - bundle instructions ..... SUCCESS [0.027s]

[INFO] tutorial - imported bundles ..... SUCCESS [0.038s]

[INFO] com.javahispano.osgi.holamundo ..... SUCCESS [47.150s]

[INFO] ------------------------------------------------------------------------

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] ------------------------
```

Una vez compilado nuestro proyecto, el último paso será desplegar y ejecutar el proyecto en un entorno OSGI, para ello usaremos el comando: pax-provision. Este comando descargará la plataforma Equinox (ya que así se lo hemos indicado antes en el pom.xml) y ejecutará nuestro bundle en dicho entorno, logrando así nuestro esperado saludo:

```
Pax Runner (1.3.0) from OPS4J - http://www.ops4j.org

-------------------------------------------------

-> Using config [classpath:META-INF/runner.properties]

-> Using only arguments from command line

-> Scan bundles from [/home/roberto/Documentos/osgi-tutorial/pax-construct-
1.4/bin/tutorial/runner/deploy-pom.xml]
```

```
 -> Scan bundles from [scan-pom:file:/home/roberto/Documentos/osgi-tutorial/pax-construct-1.4/bin/tutorial/runner/deploy-pom.xml]

 -> Scan bundles from [scan-composite:mvn:org.ops4j.pax.runner.profiles/minimal//composite]

 -> Provision bundle [mvn:com.javahispano.osgi/com.javahispano.osgi.holamundo/1.0-SNAPSHOT, at default start level, bundle will be started, bundle will be loaded from the cache]

 -> Preparing framework [Equinox 3.5.1]

 -> Downloading bundles...

 -> mvn:com.javahispano.osgi/com.javahispano.osgi.holamundo/1.0-SNAPSHOT : conne -> mvn:com.javahispano.osgi/com.javahispano.osgi.holamundo/1.0-SNAPSHOT : 5586 bytes @ [ 2793kBps ]

 -> Using execution environment [J2SE-1.6]

 -> Runner has successfully finished his job!

osgi> STARTING com.javahispano.osgi.holamundo

REGISTER com.javahispano.osgi.holamundo.ExampleService

Hola Mundo
```

4 ANEXO I: INDICE DE FIGURAS

	construct		

5 ANEXO II: INDICE DE LISTADOS DE CODIGO FUEN TE