OSGi

OPEN SERVICES GATEWAY INITIATIVE

OVERVIEW OF OSGi, A SERVICE PLATFORM FOR MODULAR SYSTEMS

Peter R. Egli
peteregli.net

Contents

1. What is OSGi?

OSGi = Component-based technology:

OSGi is a component-based technology, i.e. a set of components (bundles) with defined interfaces that live within a runtime environment.

OSGi = runtime / hosting environment (container):

OSGi is a runtime environment that controls the lifecycle of the components (install, start, suspend, stop, deinstall) and controls the dependencies between the components (dependency injection).

OSGi = service-based technology:

OSGi is a service-based technology. The interface of a bundle is separated from the implementation. The interface is published by the bundle in the OSGi service registry.

OSGi Implementations:

Apache Felix, Eclipse Equinox, Knopflerfish, ProSyst.

What is a component (UML definition)?

A component is a modular part of a software system with defined interfaces (ports). A component is replaceable by another equivalent component with an identical set of provided and required interfaces. A component hides its implementation to the outside world but makes its functionality available through interfaces.

2. Why OSGi?

OSGi was mainly developed because Java has only weak support for components.
The Java modularity (component) model is weak because:

- **Java selects modules (jar-files) through the classpath (selection through location instead of properties).**

- **Dependencies between jar-files are unclear.**
  java –cp foo.jar bar.jar MyApp.MainClass **(what are dependencies between foo.jar and bar.jar?)**

- **Different versions of a module (jar-file) are not directly supported.**
  java –cp foo-1.0.jar bar.jar MyApp.MainClass **(version control needs to be done on classpath level, i.e. module of a specific version needs to be selected on the classpath)**

OSGi introduces bundles (=components) with a manifest containing:
a. Defined imports (required interfaces) and exports (provided interfaces)
b. Internal classpath
c. Versioning of bundles

3. OSGi framework

Functions of OSGi framework:

1. Installation of «primary» bundle:

The primary bundle has the task to install other available bundles (as defined by OSGi initial provisioning specification).

2. Provide service registry:

The OSGi framework provides a service registry through which bundles can publish, find and bind to service.

3. Install class loader for bundles, keep track of references:

In OSGi, every bundle has its own class loader. As every Java class loader defines its own namespace, a class X loaded in class loader CL1 is different from the class X loaded in class loader CL2. Therefore the OSGi framework must make sure that package-exported classes use the same class loader.

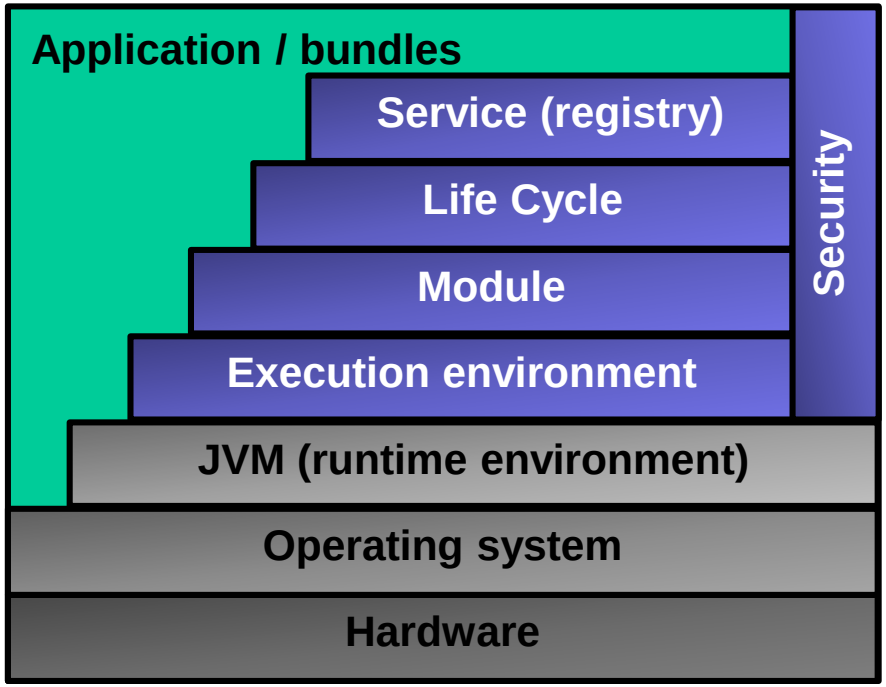
4. Provide standard services to bundles:

The OSGi framework provides standard services like:

- **Log service**
- **Event admin service**
- **Permission admin service**
- **Wire admin service**
- **Configuration admin service**

4. OSGi base architecture and layers

OSGi is based on a layered architecture that usually runs on a JVM.
OSGi may also be run on other runtime environments such as .Net (e.g. OSGi.NET), but to date OSGi is mostly used in the Java domain.



Bundles:
This layer contains the bundles developed by users.
Security:
The security «layer» is «cross-cutting» and applies its security functions to all OSGi layers.
Service (registry):
The service layer connects bundles in a dynamic way and lets bundles publish, subscribe and bind to services.
Life Cycle:
The life cycle layer controls the entire life cycle of bundles (install, start, update, stop, deinstall).
Module:
The module layer lets bundles export and import code.
Execution environment:
This layer defines which methods and classes are available in a specific OSGi runtime environment. Possible environments are CDC-1.0 (Connected Device Configuration) or JRE-1.1.
JVM (runtime environment):
The JVM is the natural run-time environment for OSGi.
N.B.: .Net could be used as well, but OSGi chose Java as the primary platform.

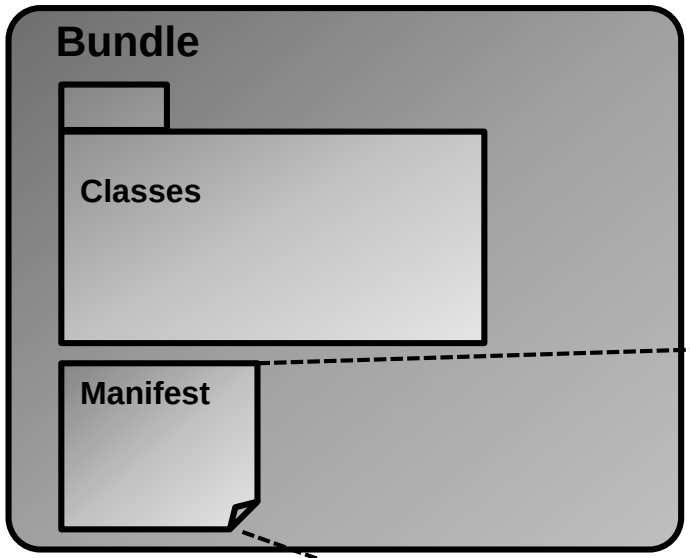
5. OSGi bundle (1/7)

Bundle = OSGi-Component:

Components in OSGi are called bundle. In Eclipse, a plugin is implemented through a bundle (since Eclipse version 3.0). Thus an Eclipse plugin is an OSGi bundle.

Contents of an OSGi bundle:

An OSGi bundle is an ordinary jar-file, i.e. contains classes plus a manifest file.
The manifest file contains OSGi-specific key-value pairs.

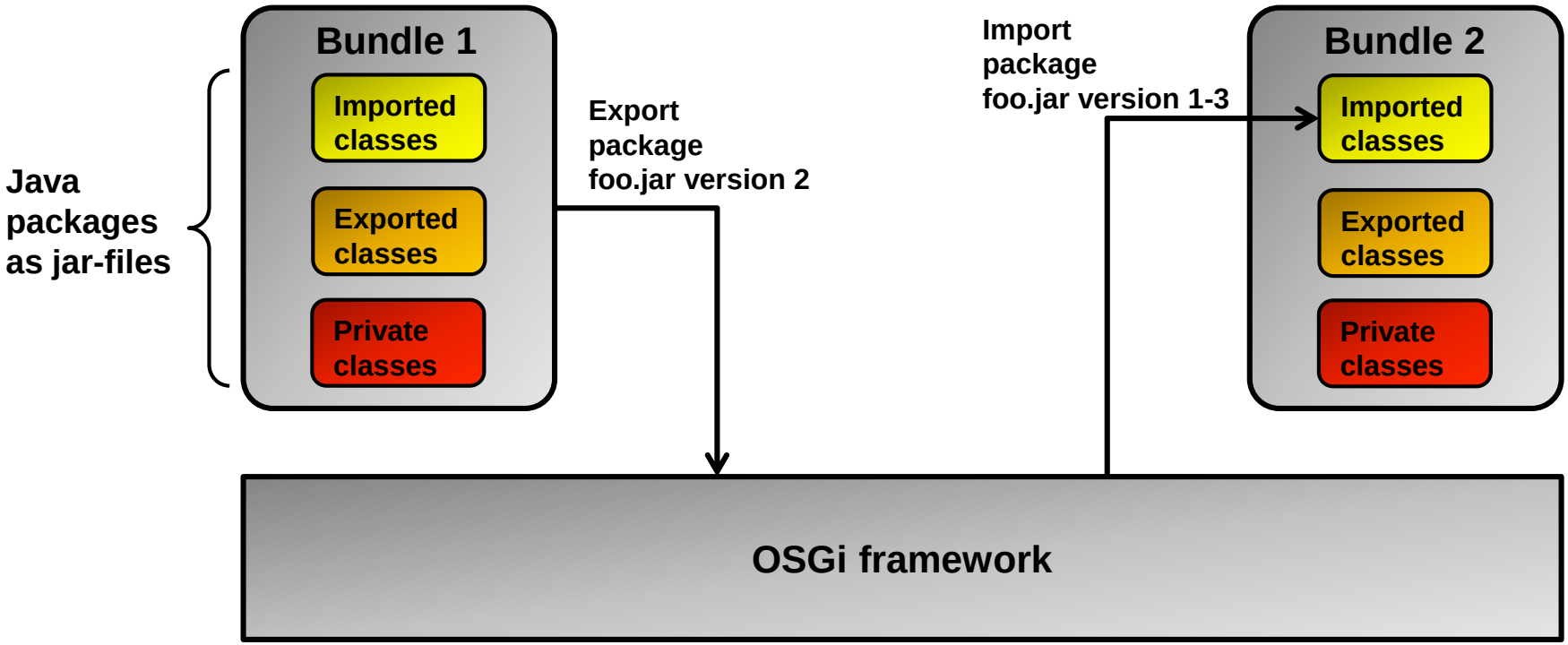



```
Manifest-Version: 1.0
Bundle-Vendor: INDIGOO
Bundle-Version: 1.0.0.201101062241
Bundle-Name: Myfirstbundle
Bundle-Activator: com.indigoo.myfirstbundle.Activator
Bundle-ManifestVersion: 2
Import-Package: org.osgi.framework;version="1.3.0"
Bundle-SymbolicName: com.indigoo.myfirstbundle
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

5. OSGi bundle (2/7)

Export and import of Java packages (set of classes) (1/2):

OSGi bundles can export and import Java packages in a controlled way.
Package export–import lets bundles share code with other bundles.
Package export–import differs from service import–export in that the package-importing bundle itself is responsible for the creation and deletion of objects (controls part of the lifecycle).



5. OSGi bundle (3/7)

Export and import of Java packages (set of classes) (2/2):

Bundles export a specific version of a package (e.g. bundle 1 exports foo.jar version 2).
Bundles define a range of acceptable versions when importing packages (e.g. bundle 2 accepts foo.jar Version 1…3).
Problem: Uninstallation of a bundle that exports a package.
Solution: OSGi framework lets importing bundles re-import another (compatible) package and restarts these bundles.

Export of package:

Packages are exported by a statement (key-value pair) in the bundle manifest file with a comma-separated list of package identifiers.
Example:

```
Export-Package: com.indigoo.myfirstbundle, com.indigoo.superbundle
```

Import of package:

Likewise, packages are imported by a statement in the bundle manifest with a comma-separated list of package identifiers. Packages can be augmented with a version identifier.
Example:

```
Import-Package: com.indigoo.myfirstbundle;version="1.0.0",
org.osgi.framework;version="1.3.0"
```

5. OSGi bundle (4/7)

Export and import of services (1/2):

Java RMI registry as a service registry:

Java provides a simple object registry service through the RMI remote registry.

Problems with Java RMI registry:

Each application or module must control its dependencies individually (usually in different ways). This leads to code duplication.

Solution with OSGi:

OSGi provides a common service registration and lifecycle service for all bundles and thus simplifies the dynamic behavior of applications.

Bundles can export functionality as a service (a registered object is a service).
When the bundle becomes active, it registers its service with the OSGi service registry.
The registry is fully dynamic.

Service registry:

→ Bundles can register objects as services with the service registry (publish).
→ Bundles can search the registry for matching objects (find).
→ Bundles can receive notifications when services become registered or unregistered.

5. OSGi bundle (5/7)

Export and import of services (2/2):

Like packages, services are exported and imported through statements in the bundle's manifest file.

Export of service with properties through bundle context:

```
public Interface IMyService {
  void hello();
}
public class MyService implements IMyService{
  public void hello() {…}
}
Hashtable props = new Hashtable();
props.put("description", "This is my first service");
bc.registerService(IMyService.class.getName(), new MyService(), props);
```

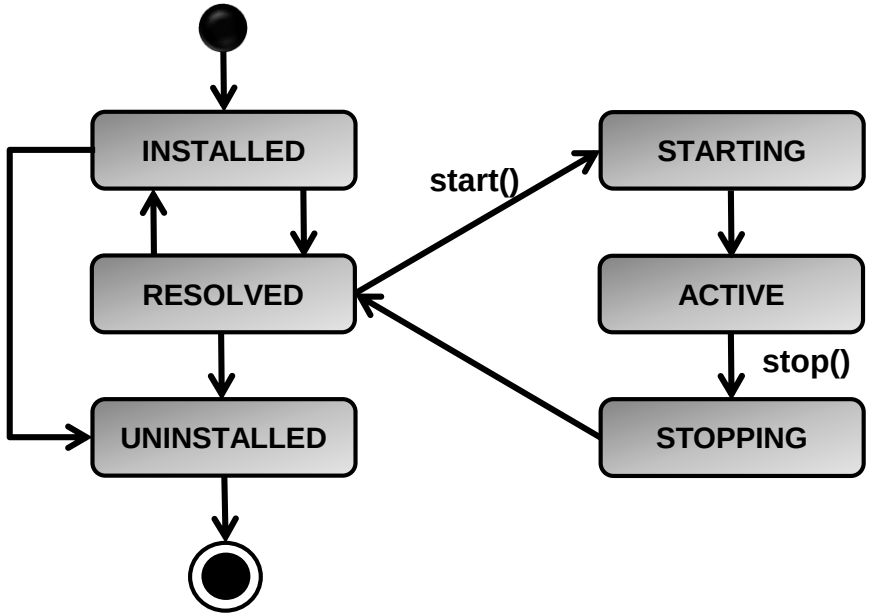
Import (find) and call of service:

```
myServiceRef = context.getServiceReference(IMyService.class.getName());
IMyService serviceObjMyService =
(IMyService)context.getService(myServiceRef);
serviceObjMyService.hello();
```

5. OSGi bundle (6/7)

Bundle life cycle:

The bundle's life cycle is controlled with 6 states.



Bundle state	Description
INSTALLED	The bundle has been successfully installed.
RESOLVED	All Java classes that the bundle requires are available. This state indicates that the bundle is ready to be started (all dependencies have been resolved).
STARTING	The bundle is being started, i.e. the OSGi platform has called Activator.start(), but the function did not yet return.
ACTIVE	The bundle has been started and is active. The bundle's expored packages and services can be used by other bundles.
STOPPING	Equivalent function to STARTING, i.e. the bundle is being stopped.
UNINSTALLED	The bundle has been uninstalled.

5. OSGi bundle (7/7)

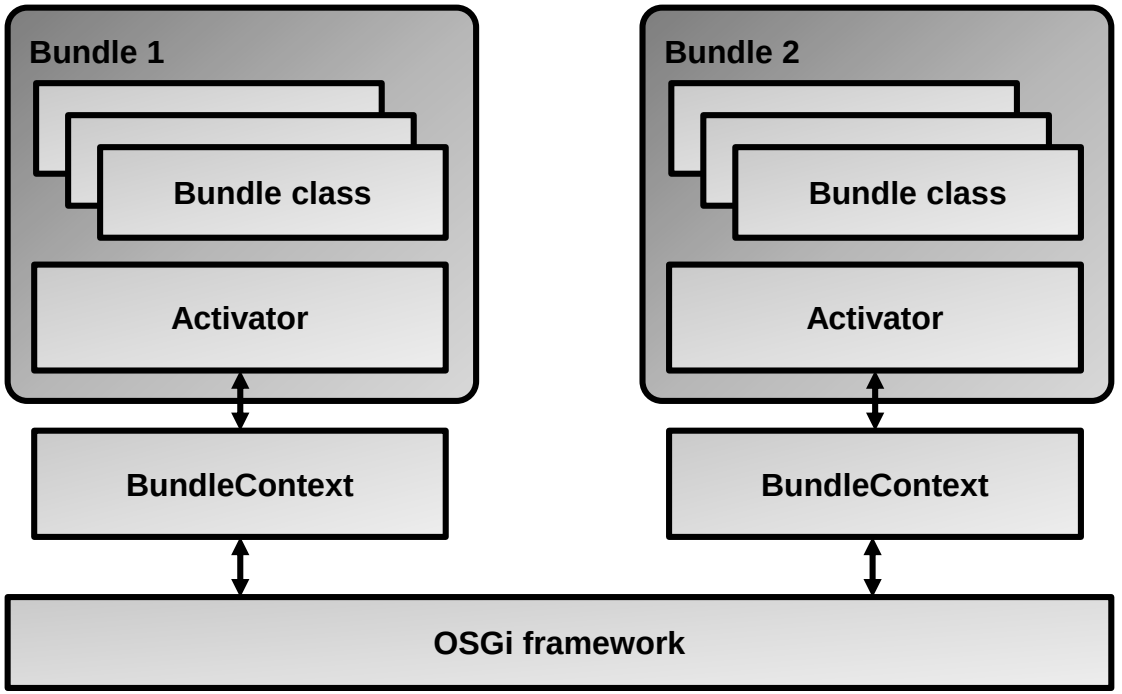
BundleContext object:

The BundleContext object is the «bridge» between a bundle and the OSGi framework.
The BundleContext API lets bundles:

a. register services
b. install service listeners (get a notification upon the availability of a service)
c. obtain references to installed bundles
d. obtain references to installed services (bundle object)
e. install bundles

Bundle Activator object:

The bundle Activator object is used by the OSGi framework to control the life cycle of the bundle (start(), stop() etc.).



6. OSGi service registry

The service registry is the central component of OSGi. Through its dynamic behavior, the service registry allows to build a SOA of loosely coupled components.

Registration of a service:

A bundle registers a service (=object) with an interface name and a set of properties.
Example:

Interface name: `org.osgi.service.log.LogService`
Property: `vendor=indigoo`

Registration is dynamic:

If a bundle is stopped, its registered services are unregistered by the OSGi framework.

Declarative service registration:

If the OSGi framework implements the «Declarative Service Spec.», classes of a registered service are only loaded when another bundle calls the service («lazy loading»).
This helps reducing the memory required for registered services.

Service discovery:

a. Notification-based discovery: A bundle is notified of the existence of a newly registered service.
b. Active discovery: A bundle actively searches for a specific service.
OSGi supports a filter language to specify what service is searched.

7. OSGi security

Security is a natural concern when using connected components, particularly in a very dynamic environment such as OSGi.

OSGi defines a multi-layer security concept that is based on the following layers:

1. Java 2 code security:

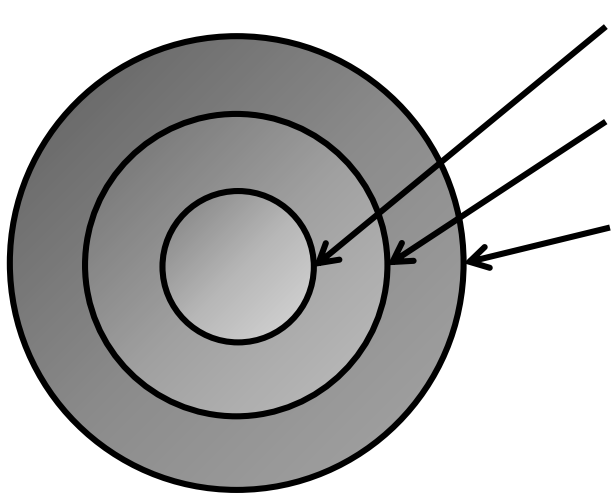
E.g. Java file permissions with security manager, Java access modifiers protected/private, Java language security (no pointers etc.).

2. Minimized bundle content exposure:

Bundle packages and services are not accessible to other bundles except the ones that the bundles explicitly exports through OSGi.

3. Managed communication links between bundles:

Services are augmented with permissions to give only specific bundles access (use, provide) to other services.



8. OSGi specifications (1/5)

OSGi is specified in a number of specifications that share some chapters. As such the documents define a kind of a profile (applicability of chapter specifications to different environments).

Chapters	Description	Core Spec.	Compendium Spec.	Enterprise Spec.	Mobile Spec.
2: Security	Optional layer that underlies the OSGi service platform. Provides fine-grained security control on top of the Java security model.	Yes	No	No	No
3-6: Core framework	Mandatory implementation. Defines the layers Module, Life Cycle, Service and framework API.	Yes	No	No	No
7: Package admin service	Service that manages the lifecycle and depencencies of packages (classes and resources).	Yes	No	No	No
8: Start level service	Controls the startup sequence of bundles (some bundles may have to be started before others).	Yes	No	No	No
9: Conditional permission admin service	Extends Permission admin service. Allows the definition of permissions that are based on conditions such as *user prompt response*, *geographic location of the bundle* (~LBS: Location Based Service) etc.	Yes	No	No	No
10: Permission admin service	Manipulation of permissions of registered or future bundles.	Yes	No	No	No
11: URL handler service	Allows to dynamically add new URL schemes (protocols) and a corresponding content handler.	Yes	No	No	No
12: Service hooks	Allows extended functionality beyond the standard service primitivies *publish*, *find* and *bind*. Allows to interact with the service engine, e.g. to provide a proxy service by a bundle as soon as another bundle requests the service → service hooks).	Yes	No	No	No

8. OSGi specifications (2/5)

Chapters	Description	Core Spec.	Compendium Spec.	Enterprise Spec.	Mobile Spec.
13: Remote service	Allows distributed components and connecting bundles in a distributed environment. Details see "Distributed OSGi" below.	No	Yes	Yes	No
101: Log service	Warning, debug or error logging.	No	Yes	Yes	Yes
102: HTTP service	Allows bundles to provide JEE-servlets that are accessible over HTTP (servlets can thus be lifecycle-managed within the OSGi framework without restart).	No	Yes	Yes	No
103: Device access service	Locate and start a driver bundle that matches a new device (used for plug'n'play scenarios).	No	Yes	No	No
104: Configuration admin service	Configuration read and write service.	No	Yes	Yes	Yes
105: Metatype service	Allows to define attributes of bundles in computer-readable form as key-value pairs.	No	Yes	Yes	Yes
106: Preferences service	Access to a hierarchical database of properties (similar to Windows registry or Java preferences class).	No	Yes	No	No
107: User admin service	User authentication and authorization.	No	Yes	Yes	No
108: Wire admin service	Normally bundles define the rules to locate other bundles they use. Alternatively the wire admin services allows to connect different services together as defined by a configuration (deployment configuration).	No	Yes	No	No
109: IO connector service	Alternative scheme (protocol) registration service for the JME generic connection framework.	No	Yes	No	Yes

8. OSGi specifications (3/5)

Chapters	Description	Core Spec.	Compendium Spec.	Enterprise Spec.	Mobile Spec.
110: Initial provisioning	Allows to install management agents with a specific management protocol for remote management of an OSGi service platform.	No	Yes	Yes	No
111: UPnP™ device service	a. Mapping of UPnP devices to the service registry. b. Mapping of OSGi services to the UPnP network.	No	Yes	No	No
112: Declarative services	Allows defining service dependencies (import, export service) with an XML-file instead of programmatically importing and exporting services.	No	Yes	Yes	Yes
113: Event admin service	Publish – subscribe service for synchronous and asynchronous events.	No	Yes	Yes	Yes
114: Deployment admin service	Allows managing of deployment packages = group of resources. Sets of packages can be lifecycle-managed as a unit.	No	Yes	No	Yes
115: Auto configuration	Allows the definition of configuration resources that are packed into a deployment package. The configuration resources are processed by an Autoconf Resource Processor.	No	Yes	No	Yes
116: Application admin	Service to register application bundles and start these on demand.	No	Yes	No	Yes
117: DMT admin service	Device management tree service (abstract tree containing management information).	No	Yes	No	Yes
118: Mobile conditions spec.	Defines conditions in mobile environments that can be used in conjunction with the conditional permission admin service.	No	No	No	Yes
119: Monitor admin service	Standard performance monitoring for bundles.	No	Yes	No	Yes
120: Foreign application access	Service for deploying non-OSGi applications (e.g. Midlets) into OSGi.	No	Yes	No	Yes

8. OSGi specifications (4/5)

Chapters	Description	Core Spec.	Compendium Spec.	Enterprise Spec.	Mobile Spec.
121: Blueprint container	Dependency injection framework derived from the Spring Dynamic Modules project.	No	Yes	Yes	No
122: Remote service admin	Administration of services in a distributed environment, i.e. which services should be available in an OSGi-network.	No	No	Yes	No
123: JTA service	Transaction service based on JTA (Java Transaction API).	No	No	Yes	No
124: JMX™ management model	Provides a JMX-compliant OSGI-platform management interface to connect OSGi to JMX-compliant implementation. JMX: Java Management Extensions	No	No	Yes	No
125: JDBC service	Defines interfaces for JDBC drivers to connect an RDBMS to OSGi.	No	No	Yes	No
126: JNDI service	Defines how JNDI (Java Naming and Directory Interface) can be used from within an OSGi platform.	No	No	Yes	No
127: JPA service	Defines how persistent units can be published in an OSGi-platform (ORM: Object Relational Mapping).	No	No	Yes	No
128: Web applications	Defines the web application bundle which performs the same function as a WAR-file (Web ARchive) in JEE.	No	No	Yes	No
129: SCA configuration type	SCA: Service Configuration Architecture. Defines a concrete configuration type for distributed OSGi applications. SCA is based on JCA (Java Connector Architecture).	No	No	Yes	No

RDBMS: Relational DataBase Management System

8. OSGi specifications (5/5)

Chapters	Description	Core Spec.	Compendium Spec.	Enterprise Spec.	Mobile Spec.
701: Tracker specification	The OSGi platform is very dynamic and pinpointing problems may thus be very difficult (e.g. race conditions). Allows tracking of bundles and services for monitoring and debugging purposes.	No	Yes	Yes	Yes
702: XML parser specification	Defines a common XML parser to be used by all services and bundles in order to reduce memory requirements.	No	Yes	Yes	Yes
703: Position specification	Standard geographic location service.	No	Yes	No	No
704: Measurement and state specification	Standard measurement class that handles issues like measurement units conversion (e.g. [m]→[in]) in a consistent way .	No	Yes	No	No
999: Execution environment specification	Defines 2 execution environments: a. JRE-based execution environment (server platform) b. CDC (Connected Device Configuration, mobile device platform)	No	Yes	No	No

9. When is OSGi applicable?

OSGi is generally suited for applications that:

a. need a dynamic environment (dynamic lifecycle).
b. consist of loosely coupled components, possibly from different vendors.
c. are based on the Java environment (OSGi may be extended to non-Java environments in the future).

Target environments of OSGi:

Initially OSGi was targeted at mobile and home networking applications.
Later OSGi was adopted in the desktop market (RCP: Rich Client Platform of Eclipse).
OSGi is increasingly used in enterprise environments in distributed environments.

Where is OSGi employed (examples):

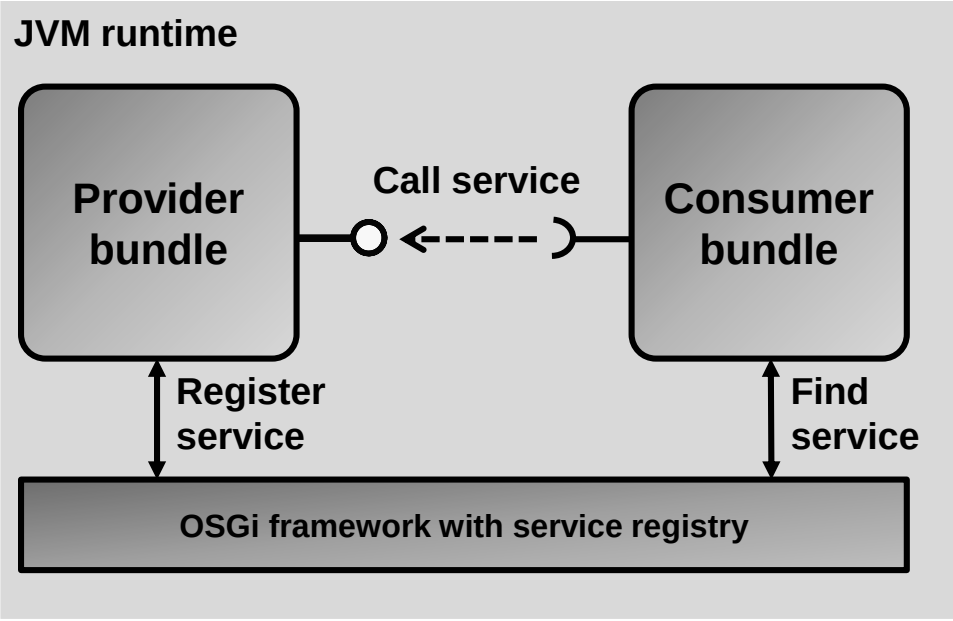
- **Glassfish JEE application server (OSGi kernel as core of Glassfish)**
- **JOnAS JEE application server**
- **IBM WebSphere application server**
- **NetBeans IDE**
- **SIP communicator (Java VoIP and IM client)**
- **Eclipse 3.0 (OSGi = Eclipse runtime)**

10. OSGi remote services (1/2)

Remote services for distributed applications:

OSGi remote services (available since OSGi 4.2, formerly known as Distributed OSGi) allows to connect different OSGi framework runtimes (different OSGi JVMs) over a network.
Distributed OSGi is achieved through OSGi distribution providers, basically proxy objects that connect service provider and consumer.

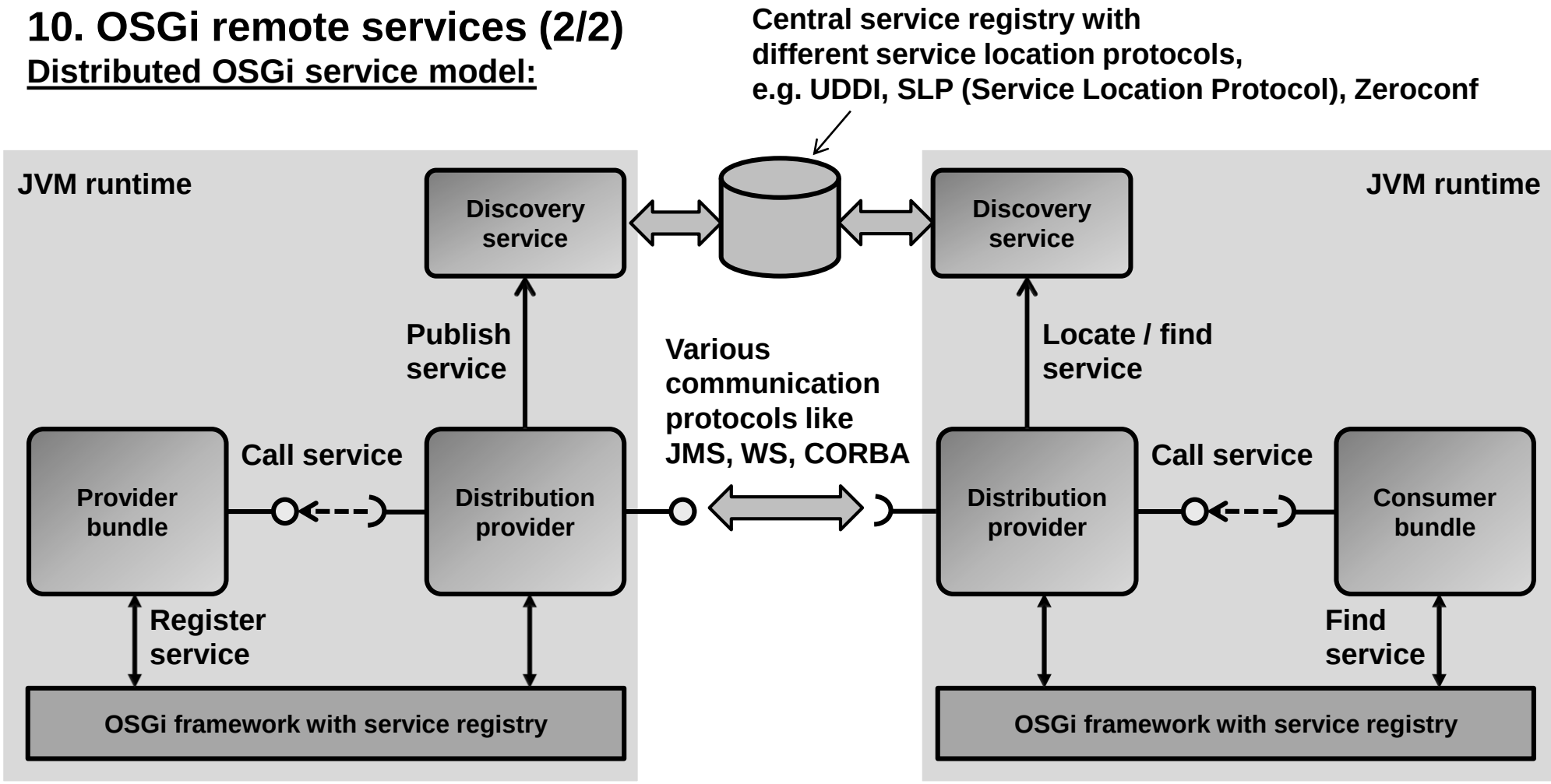
Single-JVM OSGi service model:



The OSGi framework and all bundles run in one single JVM.

10. OSGi remote services (2/2)

Distributed OSGi service model:



Criticism of OSGi remote services:

OSGi remote services extends the OSGi OOP (OO programming paradigm) to distributed computing, thus implementing something similar to DCOM, CORBA, RMI, EJB.