

JSR 277, 291 and OSGi, Oh My! - OSGi and Java Modularity

Richard S. Hall

June 28th, 2005



LSR-Adèle

ApacheCon
Europe 06



Agenda

- Modularity
- Modularity in Java
- Modularity in Java + OSGi technology
 - Introduction to OSGi technology
- Apache Incubator Felix Project Overview
- JSR 291 Overview
- JSR 277 Overview
- Bonus JSR 294 Overview
- Conclusion

Modularity

What is Modularity? (1)

- “(Desirable) property of a system, such that individual components can be examined, modified and maintained [and deployed] independently of the remainder of the system. Objective is that changes in one part of a system should not lead to unexpected behavior in other parts.”

(www.maths.bath.ac.uk/~jap/MATH0015/glossary.html)

What is Modularity? (2)

- Different types of modularity
 - Logical
 - Useful during development to decompose and/or structure the system
 - Physical
 - Useful after development to simplify deployment and maintenance

Why Care About Modularity? (1)

- Simplifies the creation of large, complex systems
 - Improves robustness
 - Eases problem diagnosis
 - Enables splitting work among independent teams
- Simplifies the deployment and maintenance of systems
- Simplifies aspects of extensible and dynamic systems

Why Care About Modularity? (2)

- Java needs improvement in this area
 - Java currently lags .NET in support for modularity
 - OSGi specification deals with many of these issues and can fill that gap

Modularity in Java

Logical Modularity in Java

- Classes
 - Provide logical static scoping via access modifiers (i.e., public, protected, private)
- Packages
 - Provide logical static scoping via “package privates”
 - Namespace mechanism, avoids name clashes
- Class loaders
 - Enable run-time code loading
 - Provide logical dynamic scoping

Physical Modularity in Java

- Java class files
- Java Archive (JAR) files
 - Provide form of physical modularity
 - May contain applications, extensions, or services
 - May declare dependencies
 - May contain package version and sealing information

Java Modularity Limitations (1)

- Limited scoping mechanisms
 - No module access modifier
- Simplistic version handling
 - Class path is first version found
 - JAR files assume backwards compatibility at best

Java Modularity Limitations (2)

- Implicit dependencies
 - Dependencies are implicit in class path ordering
 - JAR files add improvements for extensions, but cannot control visibility
- Split packages by default
 - Class path approach searches until it finds, which can lead to shadowing or mixing of versions
 - JAR files can provide sealing

Java Modularity Limitations (3)

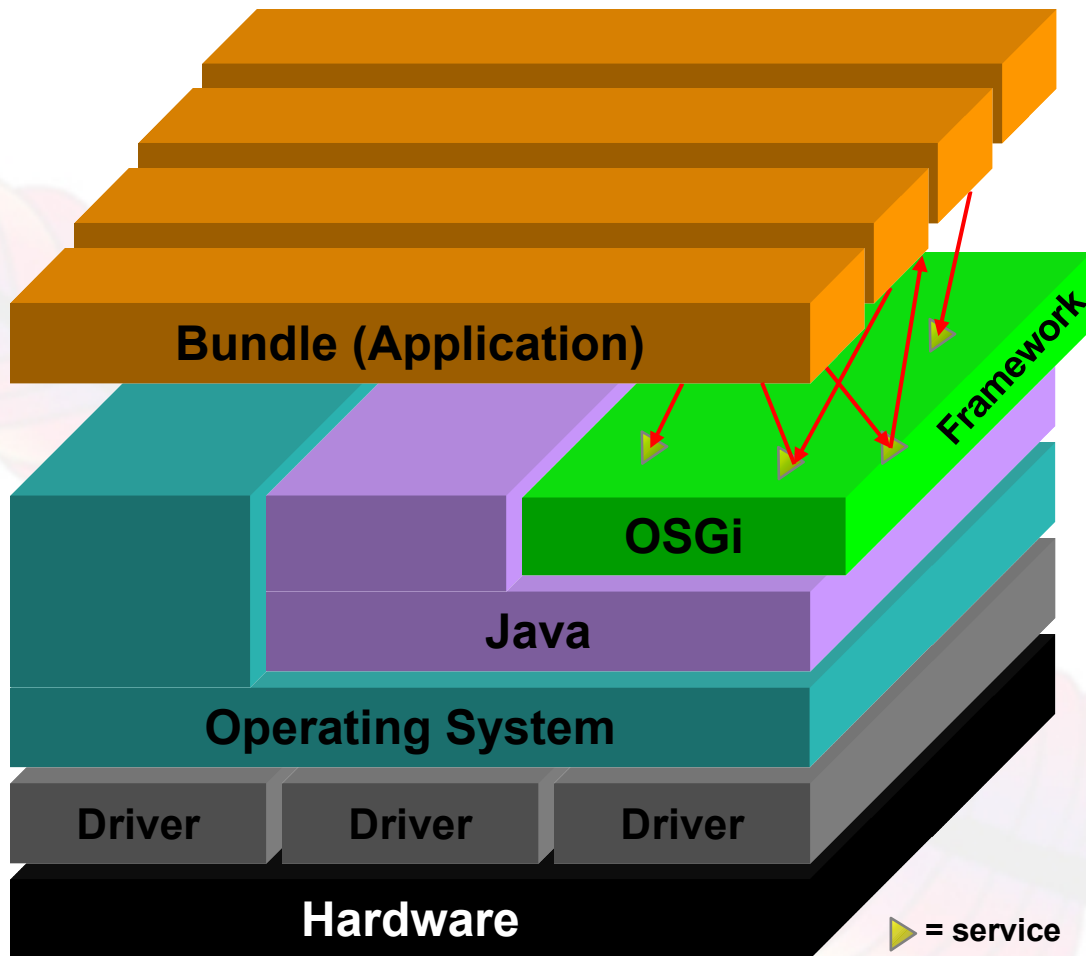
- Low-level support for dynamics
 - Class loaders are complicated to use
- Unsophisticated consistency model
 - Cuts across previous issues, it is difficult to ensure class space consistency

Java Modularity Limitations (4)

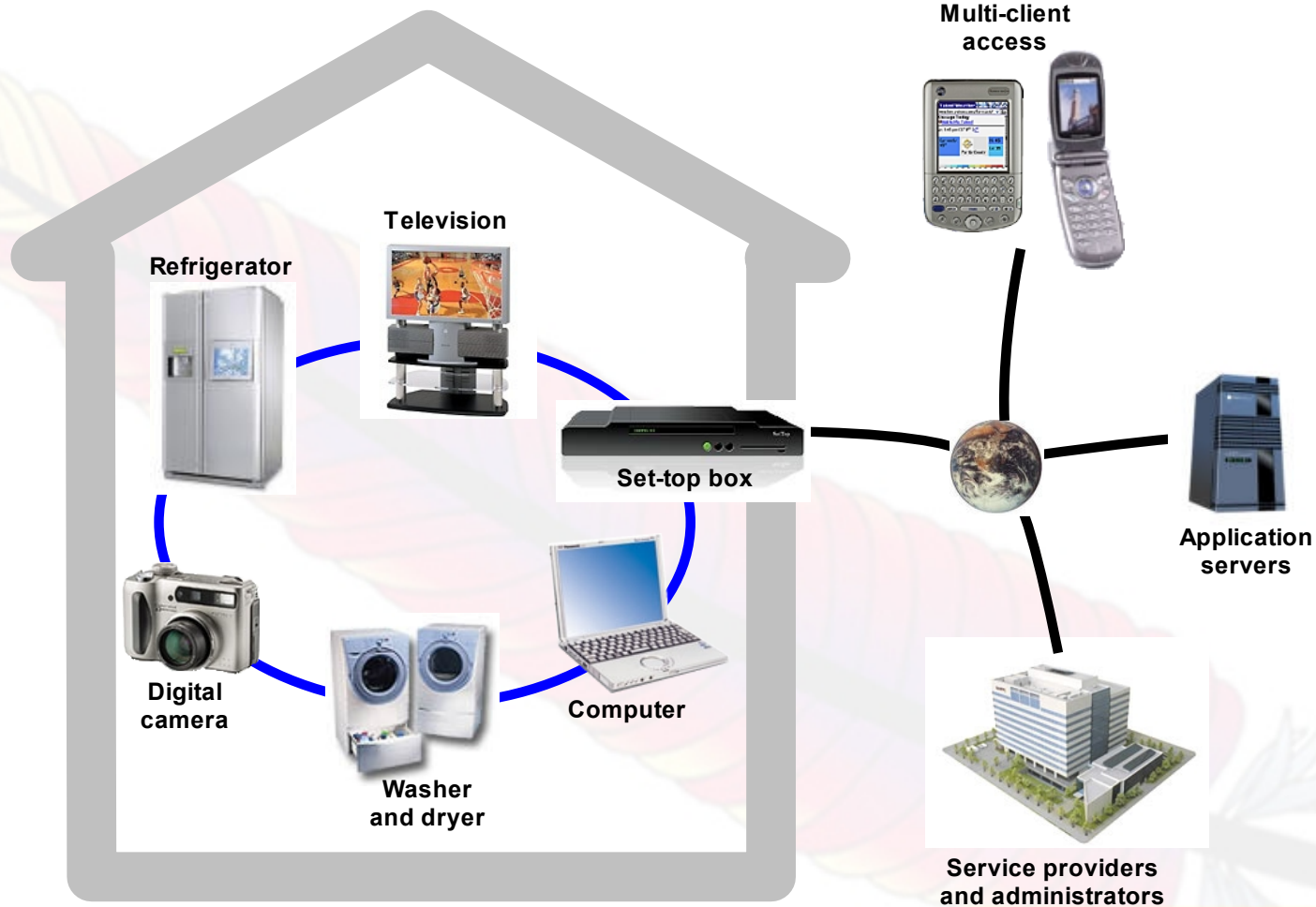
- Missing module concept
 - Classes too fine grained, packages too simplistic, class loaders too low level
 - JAR files are best candidates, but still inadequate
 - Modularity is a second-class concept as opposed to the .NET platform
 - In .NET, Assembly usage is enforced with explicit versioning rules and sharing occurs via the Global Assembly Cache

Modularity in Java + OSGi technology

OSGi Overall Architecture



OSGi Original Target



OSGi Framework (1)

- Horizontal software integration platform
- Component-oriented architecture
 - Module (Bundles)
 - Package sharing and version management
 - Life-cycle management and notification
- Service-oriented architecture
 - Publish/find/bind intra-VM service model
 - Life-cycle notification
- Open remote management architecture
 - No prescribed policy or protocol

OSGi Framework (2)

- Runs multiple applications and services
- Single VM instance
- Separate class loader per bundle
 - Class loader network
 - Independent namespaces
 - Class sharing at the Java package level
- Lifecycle management of bundles
- Intra VM publish/find/bind service model
- Java Permissions to secure framework

OSGi Framework Layering

SERVICE MODEL

L3 – Provides a publish/find/bind service model to decouple bundles – not discussed further here

LIFECYCLE

L2 - Manages the lifecycle of bundle in a bundle repository without requiring the VM be restarted

MODULE

L1 - Creates the concept of modules (aka. bundles) that use classes from each other in a controlled way according to system and bundle constraints

Execution Environment

L0 -

- OSGi Minimum Execution Environment
- CDC/Foundation
- JavaSE

OSGi Module Layer

- Generic/standardized solution for Java modularity
- Bundle is the unit of modularization
 - Contains Java classes and other resources
 - Typically a JAR file
- Bundles share Java packages in a well-defined way
 - Export a package
 - Import a package
- Bundle manifest contains module metadata
 - Bundles are resolved
 - Imports are wired to exports
 - Class loader per bundle with wires form a delegation network

OSGi Bundle-ClassPath

- Manifest header

```
Bundle-Classpath ::= entry ( ',' entry ) *  
entry ::= target ( ';' target ) * ( ';' parameter ) *  
target ::= path | '.'
```

- Defines the “inner” class path of a bundle
- `path` can refer to a directory in the bundle or a contained JAR
 - `/WEB-INF/classes`
 - `util.jar`

OSGi Export-Package

- Manifest header

```
Export-Package ::= export ( ',' export )*  
export ::= package-names ( ';' parameter )*  
package-names ::= package-name ( ';' package-name )*
```

- Directive

- uses := package list
- mandatory := attribute list
- include := class list
- exclude := class list

- Attributes

- version = version
- arbitrary

OSGi Import-Package

- Manifest header

```
Import-Package ::= import ( ',' import )*  
import ::= package-names ( ';' parameter )*  
package-names ::= package-name ( ';' package-name )*
```

- Directive

- resolution := mandatory | optional

- Attributes

- version = [low,high)
- bundle-symbolic-name = org.bundle.name
- bundle-version = [low,high)
- arbitrary

OSGi DynamicImport-Package

- Manifest header

```
DynamicImport-Package ::= dynamic-description ( ','  
dynamic-description ) *  
dynamic-description ::= wildcard-names ( ';' parameter ) *  
wildcard-names ::= wildcard-name ( ';' wildcard-name ) *  
wildcard-name ::= package-name | ( package-name '*' ) | '*'
```

- Dynamic imports are matched to export definitions during class loading
 - Does not affect module resolution
 - Apply only to packages for which no wire has been established
 - Dynamic import is used as last resort
- For `Class.forName` idiom

OSGi Require-Bundle

- Manifest header

```
Require-Bundle ::= bundle-description ( ',' bundle-  
description ) *  
bundle-description ::= symbolic-name ( ';' parameter ) *
```

- Bundles can be directly wired to the exported packages of other bundles without specifying the specific packages
- Used by Eclipse plugins as a convenience
- Not preferred approach
 - Has several drawbacks but is seen as convenient by developers

OSGi Fragments

- Manifest header

```
Fragment-Host ::= bundle-description  
bundle-description ::= symbolic-name (';' parameter)*
```

- Fragments are bundles that are attached to a host bundle as part of resolving
 - Treated as part of the host bundle
 - Must not have their own class loader.
- Localization is key use case

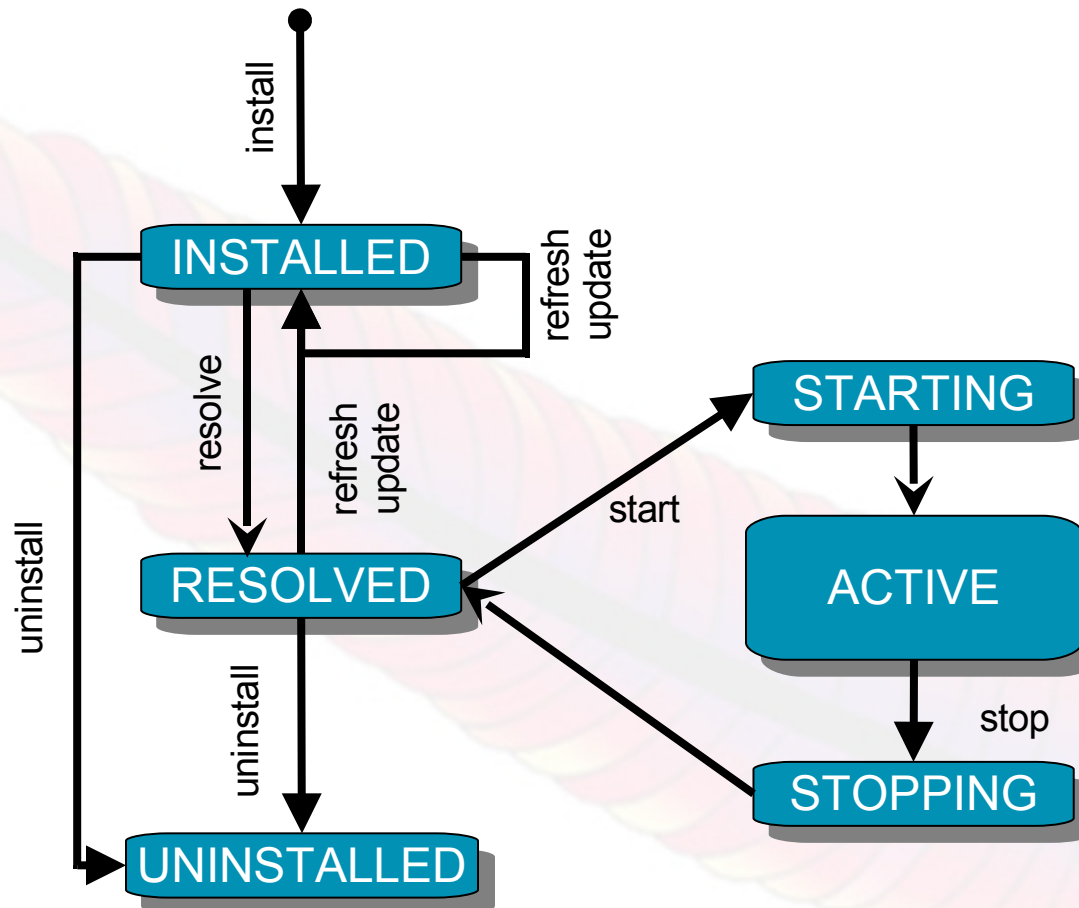
OSGi Extension Bundles

- Framework Extension
 - Loaded by framework class loader
 - Used to extend the framework implementation
- Boot Class Path Extension
 - Loaded by the boot class loader
 - Used to supply JRE extensions
- Declared as fragments of System Bundle

OSGi Lifecycle Layer (1)

- Provides APIs to control the life cycle of bundles
 - Install/Update/Uninstall
 - Start/Stop
 - Notifications
- API provides information on state of module layer
- Builds upon the module layer
- Supports dynamic management of bundles in running VM

OSGi Lifecycle Layer (2)



OSGi Modularity Best Practices

- Partition public and non-public classes into separate packages
 - Packages with public classes can be exported
 - Packages with non-public classes are not exported
- Import-Package rather than Require-Bundle
 - Allows substitutability of package providers
- Limit fragment use
- Do not use DynamicImport-Package

OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: .,org/foo/embedded.jar
Bundle-NativeCode:
    libfoo.so; osname=Linux; processor=x86,
    foo.dll; osname=Windows 98; processor=x86
Import-Package:
    javax.servlet; version="[2.0.0,2.4.0)";
    resolution:="optional"
Export-Package:
    org.foo.service; version=1.1;
    vendor="org.foo"; exclude:="*Impl",
    org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```


OSGi Bundle Manifest Example


Bundle-ManifestVersion: 2

```
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: ./embedded.jar
Bundle-NativeCode: libfoo.so; osname=Linux; processor=x86,
foo.dll; osname=Windows 98; processor=x86
Import-Package:
    javax.servlet; version="[2.0.0,2.4.0)";
    resolution:="optional"
Export-Package:
    org.foo.service; version=1.1;
    vendor="org.foo"; exclude:="*Impl",
    org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```

Indicates R4
semantics and syntax

OSGi Bundle Manifest Example


```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.activator
Bundle-ClassPath: .,org.foo.embedded.jar
Bundle-NativeCode:
    libfoo.so; osname=Linux; processor=x86,
    foo.dll; osname=Windows; processor=x86
Import-Package:
    javax.servlet; version="[2.0.0,2.4.0)";
    resolution:="optional"
Export-Package:
    org.foo.service; version=1.1;
    vendor="org.foo"; exclude:="*Impl",
    org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```



Globally unique ID

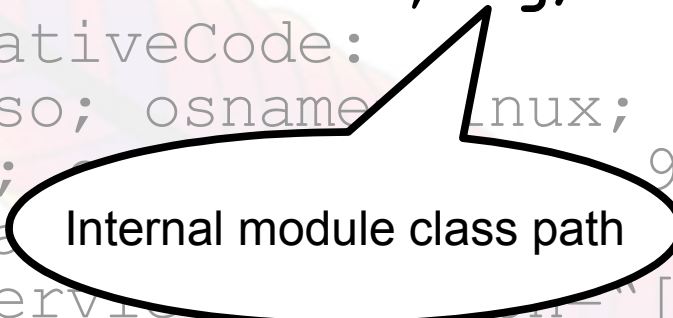
OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: ./org/foo/embedded.jar
Bundle-NativeCode:
    libfoo.so; processor=x86,
    foo.dll; processor=x86; processor=x86
Import-Package:
    javax.servlet; version="[2.0.0,2.4.0)";
    resolution:="optional"
Export-Package:
    org.foo.service; version=1.1;
    vendor="org.foo"; exclude:="*Impl",
    org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```



OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: .,org/foo/embedded.jar
Bundle-NativeCode:
    libfoo.so; osname=linux; processor=x86,
    foo.dll; osname=win32; processor=x86
Import-Package:
    javax.service; version="[2.0.0,2.4.0)";
    resolution:="optional"
Export-Package:
    org.foo.service; version=1.1;
    vendor="org.foo"; exclude:="*Impl",
    org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```



Internal module class path

OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: ./org/foo/embedded.jar
Bundle-NativeCode:
    libfoo.so; osname=Linux; processor=x86,
    foo.dll; osname=Windows 98; processor=x86
Import-Package:
    javax.servlet; version="[2.0.0,2.4.0)";
    resolution:=optional;
Export-Package:
    org.foo.service,
    vendor="org.foo"; exclude:="*Impl",
    org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```

Native code dependencies

OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: .; libfoo.jar
Bundle-NativeCode: libfoo.so; osname=Linux; processor=x86,
foo.dll; osname=Windows; processor=x86
```

Optional dependency on a
package version range

Import-Package:

```
javax.servlet; version="[2.0.0,2.4.0)";  
resolution:="optional"
```

Export-Package:

```
org.foo.service; version=1.1;  
vendor="org.foo"; exclude:="*Impl",  
org.foo.service.bar; version=1.1;  
uses:="org.foo.service"
```


OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: ./org/foo/embedded.jar
Bundle-NativeCode:
    libfoo.so; osname=Linux; processor=x86,
    foo.dll; osname=Windows; processor=x86
Import-Package:
    javax.servlet; version=1.0; resolution:=optional
```

Provided package with
arbitrary attribute and
excluded classes

Export-Package:

```
org.foo.service; version=1.1;
    vendor="org.foo"; exclude:="*Impl",
org.foo.service.bar; version=1.1;
    uses:="org.foo.service"
```

OSGi Bundle Manifest Example

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.simplebundle
Bundle-Version: 1.0.0
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: ./org/foo/embedded.jar
Bundle-NativeCode:
    libfoo.so; osname=Linux; processor=x86,
    foo.dll; osname=Windows 98; processor=x86
Import-Package:
    javax.servlet; version="
    resolution:="optional
```

Export-Package:

```
org.foo.service; version=
vendor="org.foo"; exclude:="*Impl",
```

```
org.foo.service.bar; version=1.1;  
uses:="org.foo.service"
```

Provided package with
dependency on exported
package



Apache Incubator **Felix** Project Overview

Felix Project

- Currently in the Apache Incubator
- Apache licensed open source implementation of OSGi R4
 - Framework (in progress and functional)
 - Services (in progress and functional)
 - Package Admin, Start Level, URL Handlers, Declarative Services, UPnP Device, HTTP Service, Configuration Admin, Preferences, User Admin, Wire Admin, Event Admin, and Log
 - OSGi Bundle Repository (OBR), Dependency Manager, Service Binder, Shell (TUI and GUI), Mangan

Felix Project

- Felix community is growing strong
 - 15 committers, 7 in the last quarter
 - Code granted and contributed from several organizations and communities
 - Grenoble University, ObjectWeb, CNR-ISTI, Ascert, Luminis, Apache Directory
- Other project interest in Felix and/or OSGi
 - Apache Directory
 - Apache Cocoon
 - Apache Harmony
 - Apache Derby

Felix Project

- Felix bundle developer support
 - Apache Maven2 OSGi plugin
 - Merges OSGi bundle manifest with Maven2 POM file
 - Automatically generates Bundle-Activator, Bundle-ClassPath, and Import-Package headers as well as verifies the Export-Package header
 - Greatly simplifies bundle development by eliminating error-prone manual header creation process
 - Automatically creates final bundle JAR file, automatically embedded required library JAR files

Felix Project

- Felix bundle deployment support
 - OSGi Bundle Repository (OBR) service implements support for the new, OSGi supported bundle repository format
 - Supports a generic capability/requirement XML-based metadata model for deployment bundles and their transitive set of dependencies
 - Supports both install and update with mix-and-match versioning
 - In the process of setting up an official Felix bundle repository

Felix Project

- Roadmap
 - Incubator graduation hopefully soon
 - Version 0.8 public release after graduation
 - Snapshot of framework that is stable and includes major portions of R4 specification functionality
 - Version 0.9 developer release Summer 2006
 - Focusing on security aspects and R4 Require-Bundle
 - Version 1.0 release
 - Stable snapshot of version 0.9 by the end of Summer/early Fall 2006

JSR 291 Overview

JSR 291 Rationale

- Provide dynamic component support for existing Java SE platforms
- Align OSGi and JCP
- Align SE and ME
- Follow-on activity
 - Add Dolphin to the list of compatible supported platforms
 - Exploit Dolphin technology for static modules

JSR 291 Early Draft Review

- Start with OSGi R4 core specification
- List use cases and requirements
- Discuss, clarify, etc.
- Raise issues
- Produce early draft review specification
- Resolve issues and process review feedback
- Finish as soon as possible
 - After expert group has it say

JSR 291 Current Discussion (1)

- Packaging of JSR 291 output
 - Scoped to existing platforms, not future Java SE
- Use cases and requirements
 - Interoperability between JSR 291 and OSGi
 - JSR 291 bundles on OSGi R4, OSGi R4 bundles on JSR 291
 - Requires OSGi R4 service layer, sub-setting R4 problematic
 - Optional aspects of OSGi R4
 - Avoid optionality in modularity layer, do not mention services

JSR 291 Current Discussion (2)

- SPI
 - Some parts of OSGi currently depend on framework implementation
 - Alternatives
 - 291 spec. could require these ‘extras’
 - Bloat
 - A 291 implementation could include certain ‘extras’
 - Acceptable where broad support is not required
 - ‘Extras’ can be written to a SPI and support multiple 291 implementations
 - Ideal where broad support is required, e.g. Declarative Services

JSR 291 Current Discussion (3)

- JSR 277 & 291 interoperability
 - JSR 277 provides a foundation on which a 291 solution could be built
 - Two-way class sharing (?)
 - JSR 291 runtime (i.e., OSGi R4 framework) is present when JSR 291 modules are used

JSR 291 Raised Issues

- OSGi issue 277
 - Reference content external to the bundle JAR file on the bundle class path
- OSGi issue 288
 - Standard way to gain access to arbitrary bundle contexts
 - Currently, specific to Declarative Services

JSR 291 Further Information

- JSR home page
 - <http://www.jcp.org/en/jsr/detail?id=291>
- Expert Group mailing list and archives
 - <http://bundles.osgi.org/mailman/listinfo/jsr-291-eg>

JSR-277 Overview

JSR 277 Rationale

- JAR files are the distribution/execution format for Java applications, but...
 - Not well-suited to these tasks
 - Hard to distribute
 - Hard to version
 - Hard to reference
- Java extension mechanism is inadequate
 - Version and namespace conflicts
 - Only for a single JRE installation
- Resolve these issues

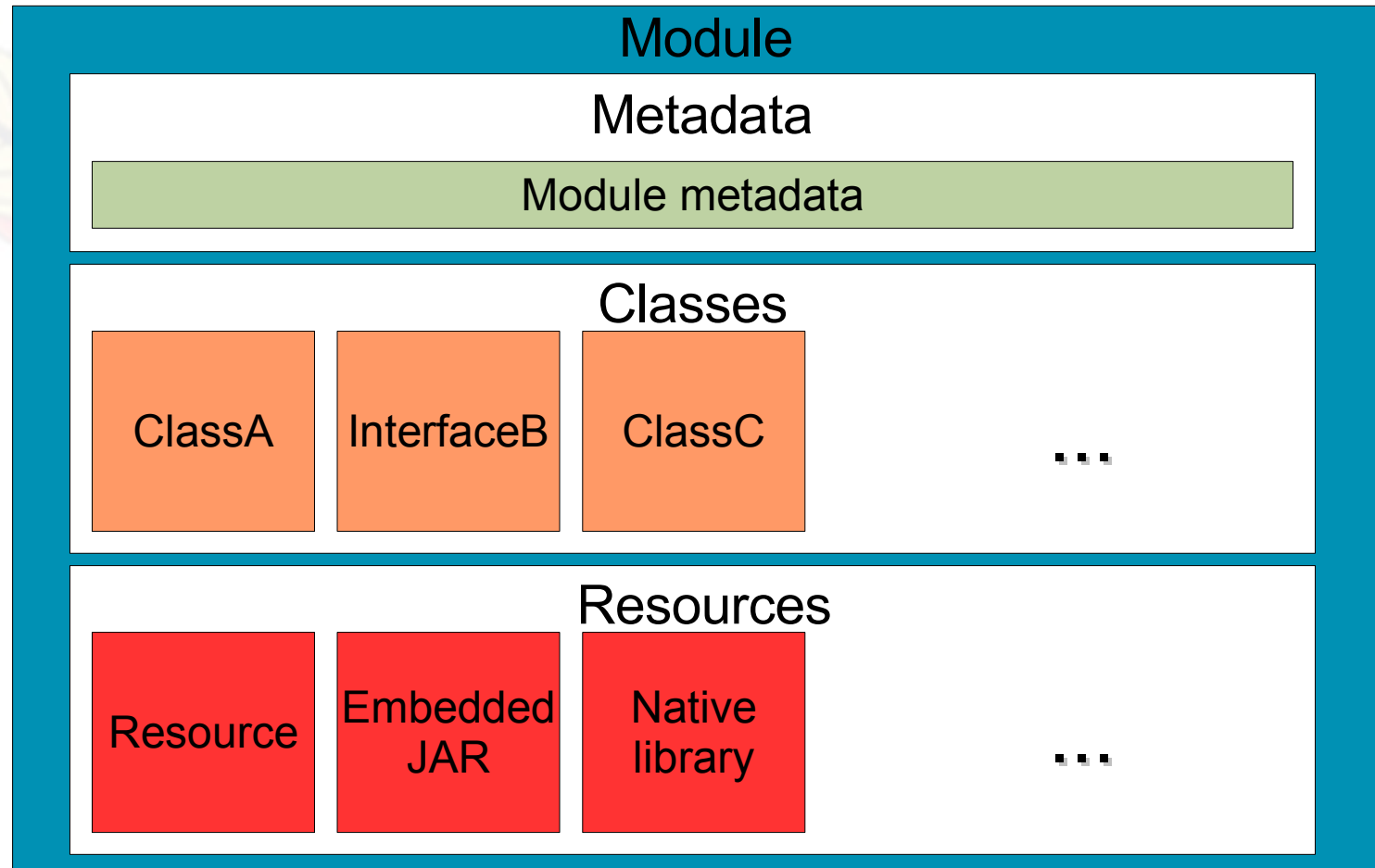
JSR 277 Focus

- Deployment modules
 - Versioning
 - Distribution and packaging
 - Repositories
 - Dependency resolution
 - Handled by tools on top of reflective run-time API

JSR 277 Concepts

- Module definition
 - Identify logical module
 - Specify content (i.e., code and resources)
 - Specify imports/exports
 - Inherently stateless
- Module instance
 - Instantiation of a module instance
 - Multiple instances can co-exist at run time

JSR 277 Logical Module View



JSR 277 Versioning

- Simple version
 - major.minor[.micro[_patch]][-qualifier]
 - e.g., 1.7.0_01-b32-beta
- Version ranges
- Version policies
 - Major element change is not compatible
 - Minor element change is generally compatible
 - Remaining changes are compatible

JSR 277 Distribution Format (1)

- Physical representation of a module definition
- Unit of packaging and deployment
- Optimized for size
- Contents
 - Module metadata, classes, resources, embedded JAR files, native libraries
- Code signing

JSR 277 Distribution Format (2)

- JAM (JAVa Module) format

org.foo.Xml-1.2.3.jam:

/META-INF

MANIFEST.MF

module/org.foo.Xml.module

/ClassA.class

/InterfaceB.class

/ClassC.class

/icon/graphics.jpg

/bin/xml-windows.dll

/bin/xml-linux.so

/lib/xml-parser.jarc

JSR 277 Distribution Format (2)

- JAM (JAVa Module) format

org.foo.Xml-1.2.3.jam:

/META-INF

MANIFEST.MF

module/org.foo.Xml.module

/ClassA.class

/InterfaceB.class

/ClassC.class

/icon/graphics.jpg

/bin/xml-windows.dll

/bin/xml-linux.so

/lib/xml-parser.jar



Module metadata

JSR 277 Distribution Format (2)

- JAM (JAVa Module) format

org.foo.Xml-1.2.3.jam:

/META-INF

MANIFEST.MF

module/org.foo.Xml.module

/ClassA.class

/InterfaceB.class

/ClassC.class

/icon/graphics.jpg

/bin/xml-windows.dll

/bin/xml-linux.so

/lib/xml-parser.jarc



Classes and resources

JSR 277 Distribution Format (2)

- JAM (JAVa Module) format

org.foo.Xml-1.2.3.jam:

/META-INF

MANIFEST.MF

module/org.foo.Xml.module

/ClassA.class

/InterfaceB.class

/ClassC.class

/icon/graphics.jpg

/bin/xml-windows.dll

/bin/xml-linux.so

/lib/xml-parser.jarc



Native libraries

JSR 277 Distribution Format (2)

- JAM (JAVa Module) format

org.foo.Xml-1.2.3.jam:

/META-INF

MANIFEST.MF

module/org.foo.Xml.module

/ClassA.class

/InterfaceB.class

/ClassC.class

/icon/graphics.jpg

/bin/xml-windows.dll

/bin/xml-linux.so

/lib/xml-parser.jarc



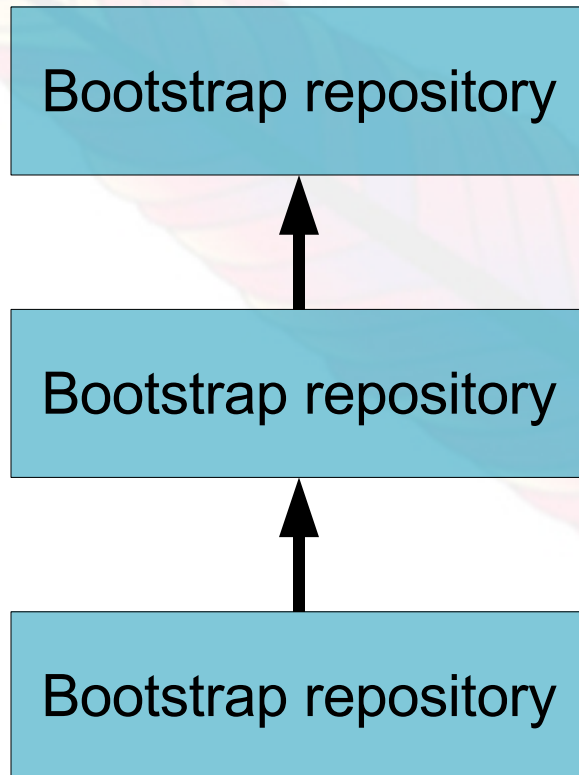
Embedded JAR files

JSR 277 Module Repository (1)

- Discovery, storage, and retrieval of module definitions
- Enable side-by-side deployment
 - More than one version of a module definition at a time to be installed
- Multiple repositories
 - Bootstrap, shared, and user repositories
- Delegation model

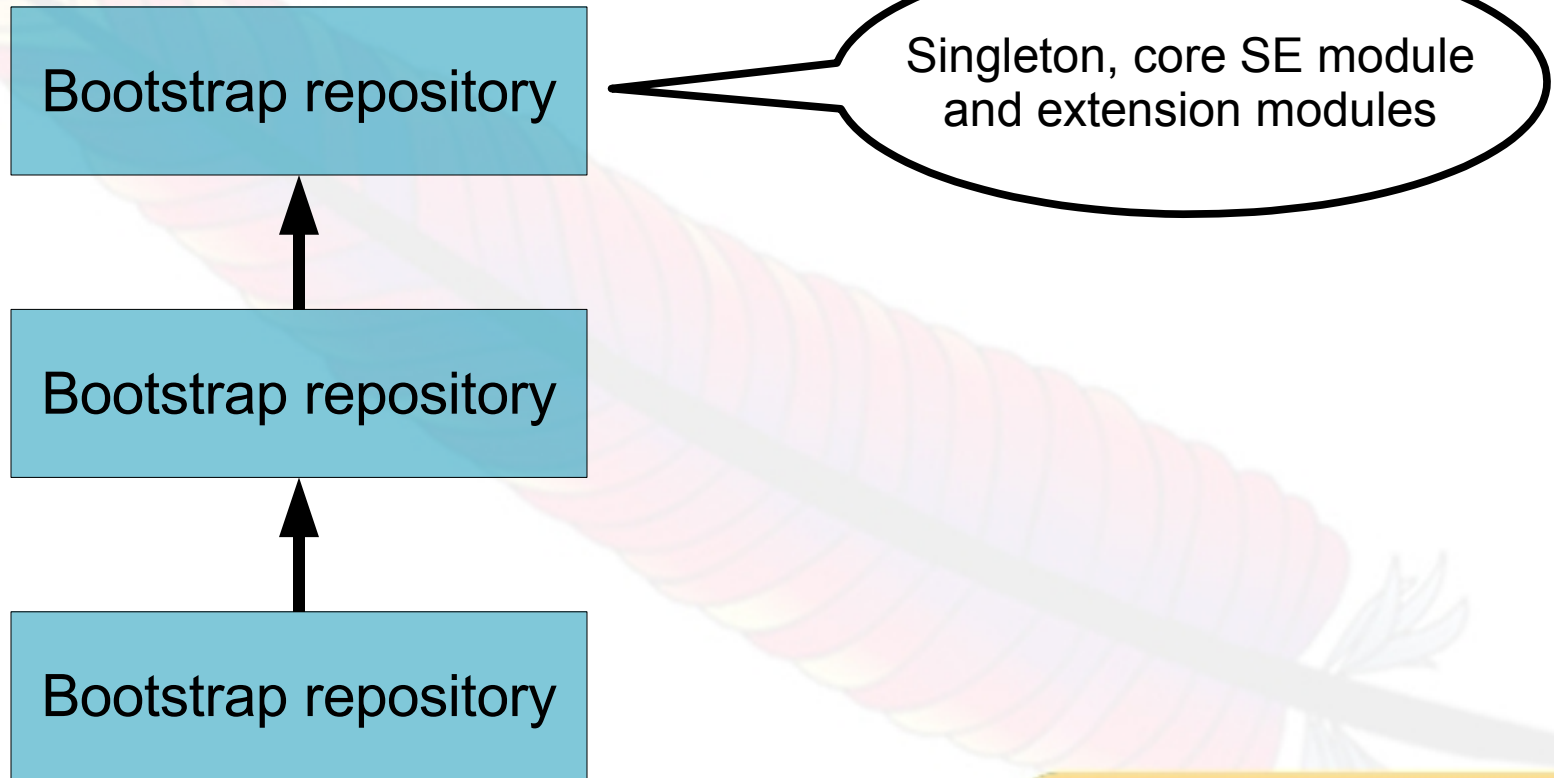
JSR 277 Module Repository (2)

- Repository delegation



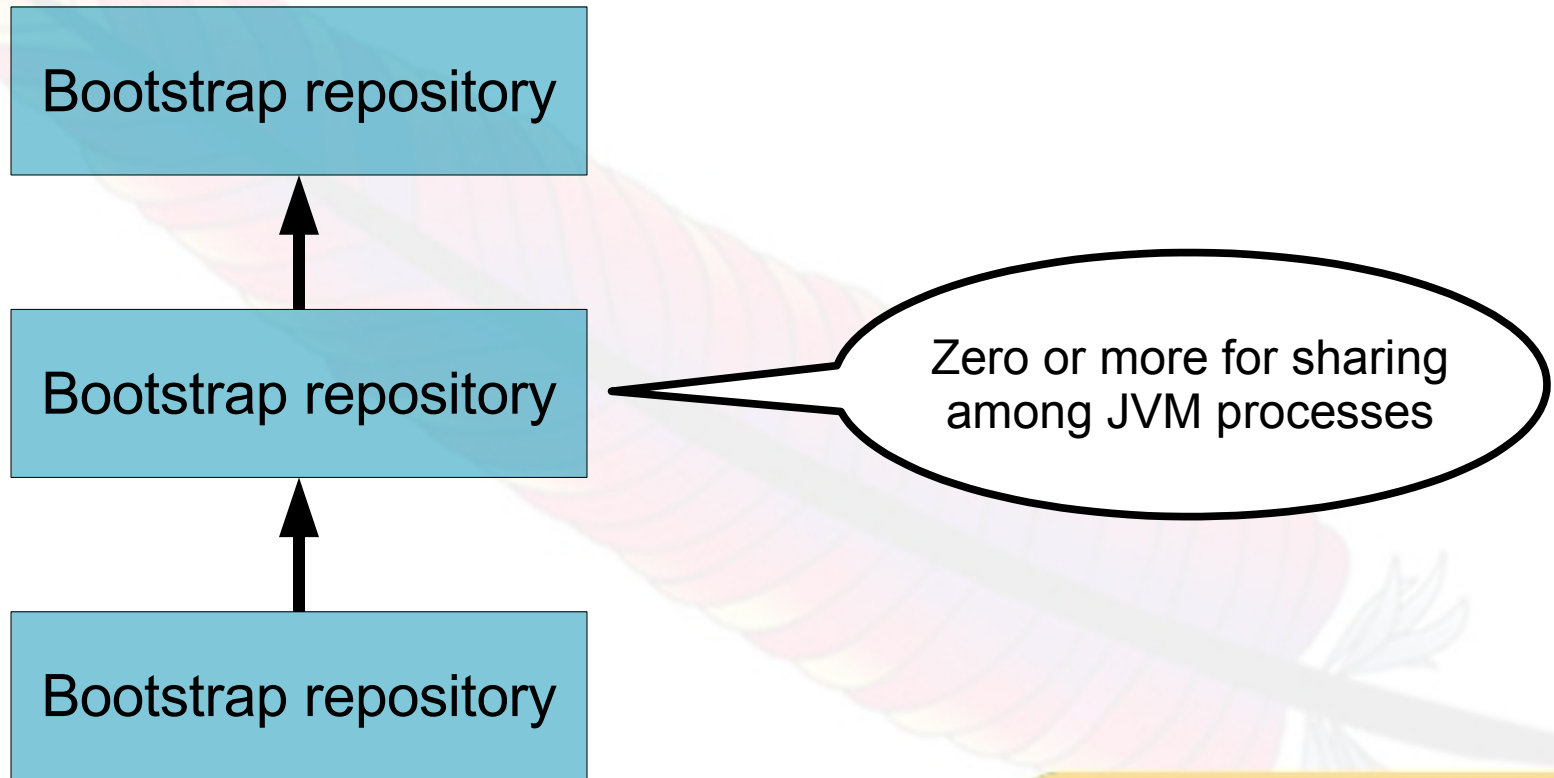
JSR 277 Module Repository (2)

- Repository delegation



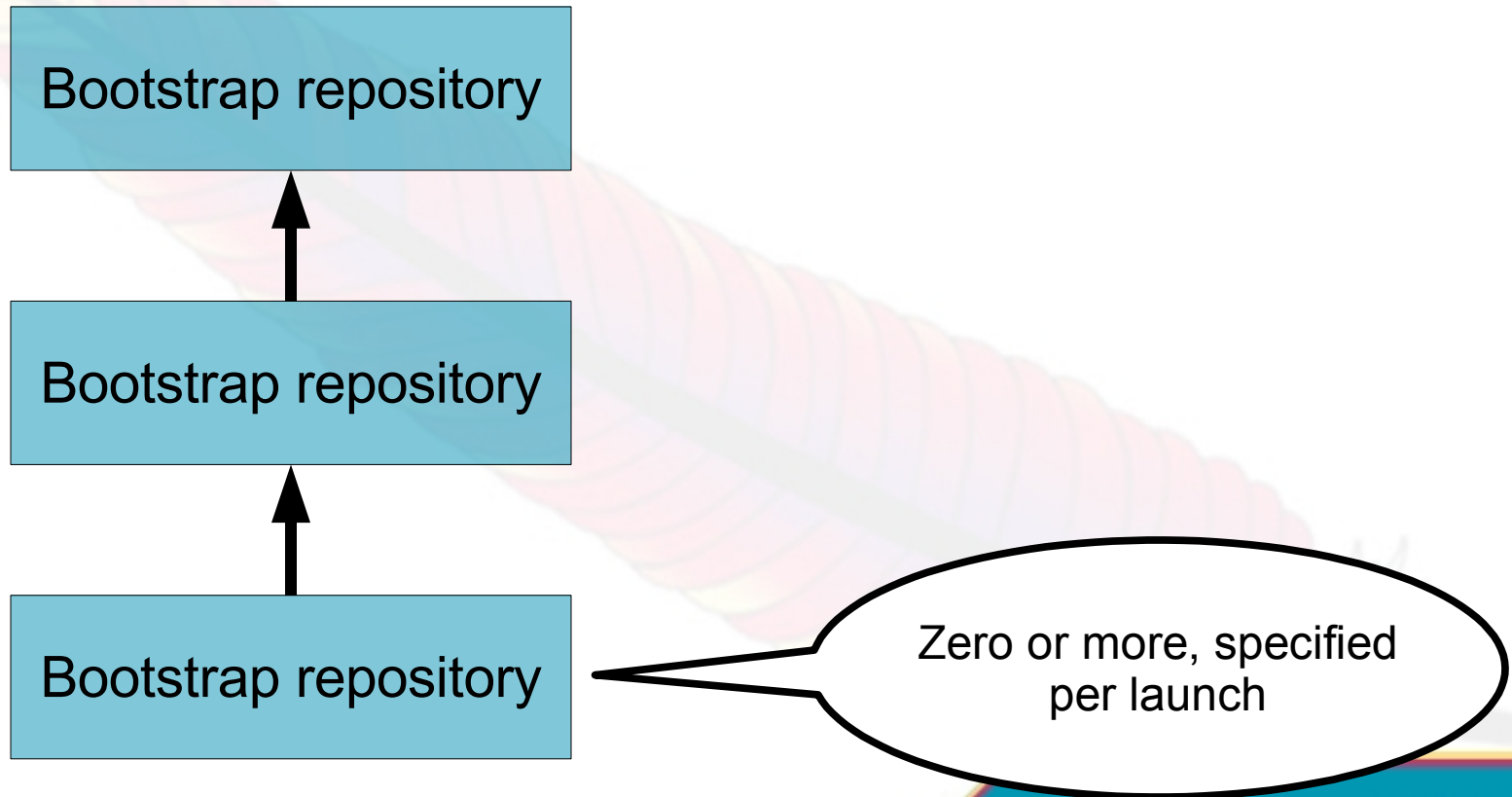
JSR 277 Module Repository (2)

- Repository delegation



JSR 277 Module Repository (2)

- Repository delegation



JSR Runtime Support

- Interconnection
 - Version constraints in imports
 - Construction script
- Validation
- Class loading
- Life cycle

Bonus JSR-294 Overview

JSR 294 Rationale

- Java programming language needs better support for hierarchical, modular organization
 - Primarily to support information hiding
 - Java packages inadequate for this purpose
 - Only two levels of visibility: internal to the package or public to everyone

JSR 294 Focus

- Development modules
 - A language construct
 - Requires direct VM support to enforce access control semantics
 - Support information hiding
 - Avoid exposing system internals
 - Support separate compilation
 - Compile against a module without an implementation of it

JSR 294 Concepts (1)

- Module “files”
 - Not necessarily a file
 - Authoritative binary definition of a module
 - Name, membership, imports, exports, metadata
 - Class files can claim membership in a module
 - Claims must be cross-checked with module file
 - VM uses membership and export information to enforce access control
 - Corresponds to the metadata of the module definition in JSR 277

JSR 294 Module File Example

```
super package com.foo.moduleA {  
    // Exported classes/interfaces  
    export com.foo.moduleA.api.*;  
    export com.foo.moduleA.ifc.InterfaceC;  
    // Imported modules  
    import org.bar.moduleD;  
    // Module membership  
    com.foo.moduleA.api;  
    com.foo.moduleA.ifc;  
    org.apache.stuff;  
}
```


JSR 294 Separate Compilation

```
package interface com.foo.moduleA;  
// Implicitly public types and members  
class C implements com.foo.ifc.InterfaceC {  
    String someMethod();  
    C(int i);  
    protected Object aFieldName;  
}
```

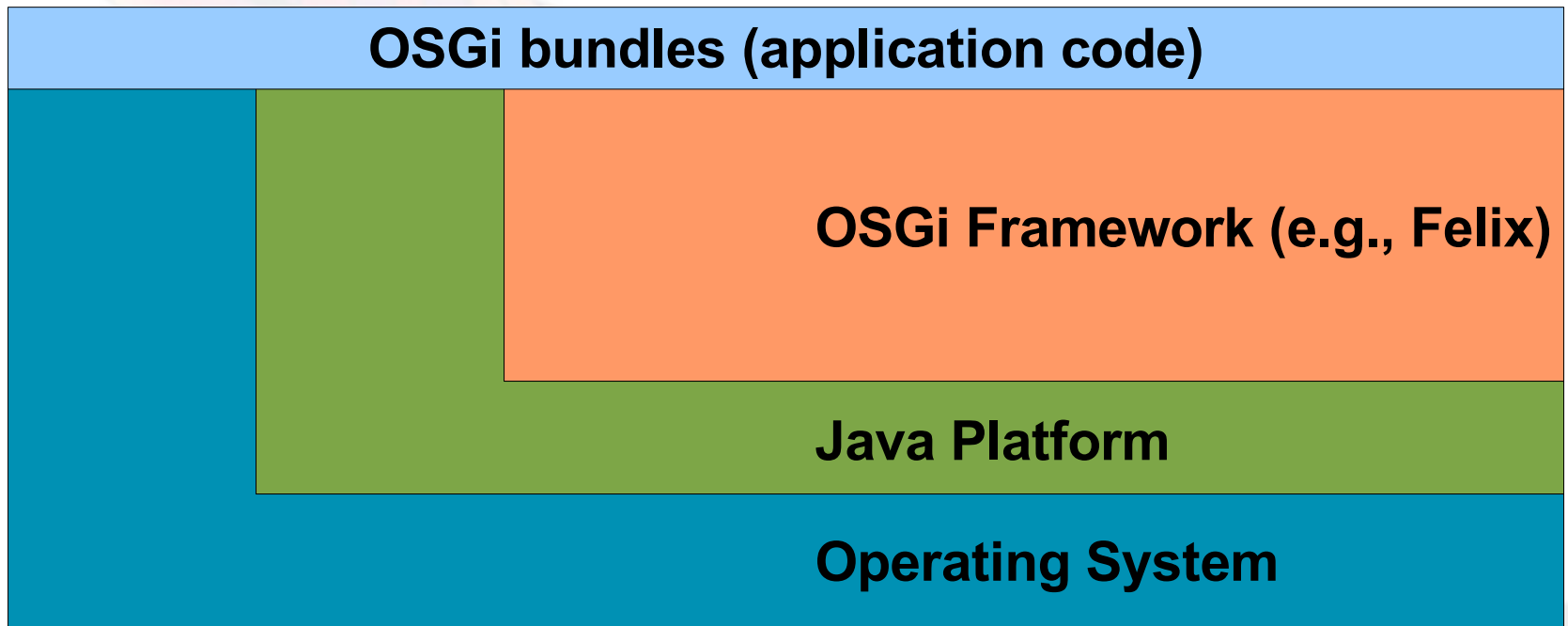
Conclusion

Issues

- OSGi specification pre-dates all existing modularity JSRs
- JSR 277, 291, and 294 have similar concepts
 - Modules with metadata for membership, imports, exports
 - Used for information hiding
- JSR 277 and 291 overlap in some areas, but differ in others
 - Overlap packaging and deployment
 - Differ in dynamics/life cycle, support for existing JREs

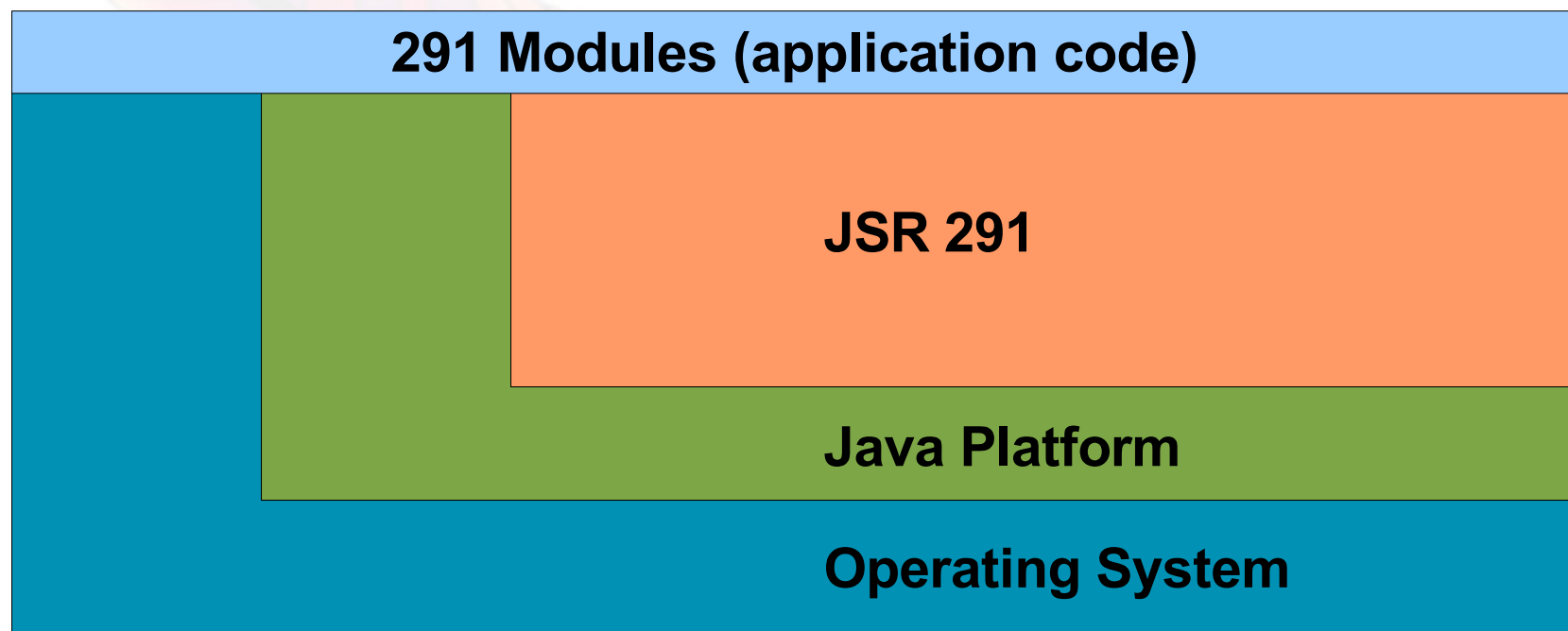
Current Solution

- OSGi framework on Java ME/SE
 - Available now and very mature



Near-term Solution

- JSR 291 framework on Java ME/SE
 - Takes EG concerns into consideration



Long-term Solution

- JSR 291 re-based onto of JSR 277

