Java Modularity Support in OSGi R4

Richard S. Hall
ApacheCon (San Diego)
December 14th, 2005
Modularity
What is it?
What is Modularity?

“(Desirable) property of a system, such that individual components can be examined, modified and maintained 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)

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?

- 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
- 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
Java Modularity

Standard Support & Limitations
Logical Modularity in Standard 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 Standard 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
Standard 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
- 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
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

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
OSGi Overview
Dynamic Service Platform
OSGi Alliance

- Formerly known as the Open Services Gateway Initiative
- Defined a framework for hosting dynamically downloadable services
- OSGi framework provides
  - Simple component model
  - Component life-cycle management
  - Service registry
  - Standard service definitions
    - Separation of specification and implementation
OSGi Overview

Hardware

Driver

Driver

Driver

Operating System

Java

Framework

OSGi

Bundle
Service Orientation

- The OSGi framework promotes a service-oriented interaction pattern
OSGi Component Model

- Simple component and packaging model
  - JAR files, called bundles, contain Java classes, resources, and meta-data
  - Meta-data explicitly defines boundaries and dependencies in terms of Java package imports(exports)
    - Dependencies and associated consistency are automatically managed

- Defines a component life cycle
- Explicitly considers dynamic scenarios
- Interaction through service interfaces
Component Life Cycle

- INSTALLED
- RESOLVED
- UNINSTALLED
- STARTING
- STOPPING
- ACTIVE

Actions:
- install
- uninstall
- resolve
- update
- start
- stop

States:
- explicit
- automatic
Bundle Dependency Resolution

- The framework automatically resolves package dependencies when a bundle is activated
  - Matches bundle’s imports to available exports
  - Ensures package version consistency
- If a bundle cannot be successfully resolved, then it cannot be activated/used
A bundle represents a single component contained in a JAR file
A bundle represents a single component contained in a JAR file.
A bundle represents a single component contained in a JAR file.
OSGi Component Model

- A bundle represents a single component contained in a JAR file
A bundle represents a single component contained in a JAR file.
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;
  specification-version=2.3
Export-Package:
  org.foo.service;
  specification-version=1.1
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;
  specification-version=2.3

Export-Package: 
  org.foo.service;
  specification-version=1.1

Life cycle entry point
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;
  specification-version=2.3

Export-Package:
  org.foo.service;
  specification-version=1.1
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;
specification-version=2.3

Export-Package:
org.foo.service;
specification-version=1.1

Native code dependencies
Bundle-Activator: org.foo.Activator
Bundle-ClassPath: .,org/foo/embedded.jar
Bundle-NativeCode:
  libfoo.so; osname=Linux; processor=x86,
  foo.dll; osname=Wind
Import-Package:
  javax.servlet;
  specification-version=2.3
Export-Package:
  org.foo.service;
  specification-version=1.1
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;
   specification-version=
Export-Package:
   org.foo.service;
   specification-version=1.1
A collection of bundles that interact via service interfaces

- Bundles may be independently developed and deployed
- Bundles and their associated services may appear or disappear at any time
OSGi R3 Modularity
Improving Standard Java
Success as a Modularity Framework

- OSGi framework is increasingly used as a modularity mechanism for Java
  - Provides logical and physical system structuring
    - Has benefits for development and deployment
  - Provides sophisticated dynamic module life-cycle management
    - Simplifies creation of dynamically extensible systems
      - Where system components can be added, removed, or rebound at run time while the system as a whole continues to function
OSGi R3 Modularity (1)

- Defines **bundle**, logical and physical modularity unit
  - Explicit boundaries
    - External interface (i.e., exports)
    - Internal class path
      - Java code, resources, and native libraries
  - Explicit dependencies
    - Package dependencies (i.e., imports)
  - Explicit versioning
    - Package version, bundle version
  - Isolation via class loaders
  - Packaging format (bundle JAR file)
OSGi R3 Modularity (2)

- Defines dynamic bundle life cycle
  - Possible to install, update, and uninstall code at run time
  - Automatic package dependency resolution
  - Replaces low-level class loaders
- Package sharing is only global
  - Cannot have multiple shared versions
- Simplistic versioning semantics
  - Always backwards compatible
- Not intended for sharing implementation packages
  - Only for specification packages, which was why the version model is simple
- Provider selection is always anonymous
  - No way to influence selection
OSGi R3 Modularity Issues (2)

- Simplistic consistency model
  - Consistency model based on single in-use version
  - No way to declare dependencies among packages
- Coarse-grained package visibility rules
  - Classes in a package are either completely visible to everyone or hidden
- Module content is not extensible
  - All content of the logical module must be included in the physical module
- Package dependencies are not always appropriate
  - Package metadata is cumbersome in large, complex systems, tightly coupled subsystems and in less structured legacy systems
In Fairness

- It is important to point out that the preceding slides do not necessarily describe shortcomings of the OSGi framework.
  - It was not designed to be a modularity layer, so it makes sense that it does not do it perfectly.
  - It was used for a modularity layer by developers because it was simple and filled a specific need.
Modularity Requirements

- Backwards compatible with OSGi R3
- Defined in terms of Java packages
  - Well-defined concept in Java
  - Maps nicely to class loaders
- Explicitly defined boundaries
- Explicitly defined dependencies
- Support for versioning and multi-versions
- Flexible, must support
  - Small to large systems
  - Static to dynamic systems
Related Work

- **Module mechanisms**
  - Mechanisms for Secure Modular Programming in Java (L. Bauer et al – Princeton University)
  - Units: Cool Modules for HOT Languages (M. Flatt and M. Felleisen – Rice University)
  - Evolving Software with Extensible Modules (M. Zenger – École Polytechnique Fédérale de Lausanne)

- **Component and extensible frameworks**
  - EJB, Eclipse, NetBeans
  - Microsoft .NET
    - Assemblies and Global Assembly Cache
Limitation: Package sharing is only global
Limitation: Package sharing is only global

Multi-version support

- Possible to have more than one version of a shared package in memory at the same time
- General change of philosophy to the prior OSGi specifications
- Has deep impact on service aspects as well as modularity
  - For a given bundle, the service registry is implicitly partitioned according to the package versions visible to it
  - Impact on services not explored further in this presentation
Limitation: Simplistic versioning semantics
Limitation: Simplistic versioning semantics

Import version ranges
- Exporters still export a precise version, but importers may specify an open or closed version range
- Eliminates existing backwards compatibility assumption
Limitation: Simplistic versioning semantics

Import version ranges

- Exporters still export a precise version, but importers may specify an open or closed version range
- Eliminates existing backwards compatibility assumption

Import-Package: foo; version="[1.0.0,1.5.0)"
- **Limitation: Simplistic versioning semantics**

Import version ranges

- Exporters still export a precise version, but importers may specify an open or closed version range
- Eliminates existing backwards compatibility assumption

  ```
  Import-Package: foo; version="[1.0.0,1.5.0)"
  ```

- **Limitation: Not intended for sharing implementation packages**
Limitation: Simplistic versioning semantics

Import version ranges

- Exporters still export a precise version, but importers may specify an open or closed version range
- Eliminates existing backwards compatibility assumption

```
Import-Package: foo; version="[1.0.0,1.5.0)"
```

Limitation: Not intended for sharing implementation packages

- Multi-version sharing and importing version ranges make implementation package sharing possible
Limitation: Provider selection is always anonymous
Limitation: Provider selection is always anonymous

Arbitrary export/import attributes

- Exporters may attach arbitrary attributes to their exports, importers can match against these arbitrary attributes
  - Exporters may declare attributes as mandatory
    - Mandatory attributes provide simple means to limit package visibility
- Importers influence package selection using arbitrary attribute matching
OSGi R4 Modularity (3)

- Limitation: Provider selection is always anonymous

- Arbitrary export/import attributes
  - Exporters may attach arbitrary attributes to their exports, importers can match against these arbitrary attributes
    - Exporters may declare attributes as mandatory
      - Mandatory attributes provide simple means to limit package visibility
  - Importers influence package selection using arbitrary attribute matching

```plaintext
Export-Package: foo;
    version="1.0.0";
    myattr="myvalue"
```

A

foo

(myattr="myvalue")

B

foo

Export-Package: foo;
    version="1.0.0"
Limitation: Provider selection is always anonymous

Arbitrary export/import attributes

- Exporters may attach arbitrary attributes to their exports, importers can match against these arbitrary attributes
  - Exporters may declare attributes as mandatory
    - Mandatory attributes provide simple means to limit package visibility
- Importers influence package selection using arbitrary attribute matching

```
Import-Package: foo;
version="1.0.0";
myattr="myvalue"
```
Limitation: **Provider selection is always anonymous**

Arbitrary export/import attributes

- Exporters may attach arbitrary attributes to their exports, importers can match against these arbitrary attributes
  - Exporters may declare attributes as mandatory
    - Mandatory attributes provide simple means to limit package visibility
- Importers influence package selection using arbitrary attribute matching

```
Import-Package: foo;
version="1.0.0";
myattr="myvalue"
```

```
A
  foo (myattr=myvalue)

C
  foo

B
  foo
```
Limitation: Simplistic consistency model
**Limitation: Simplistic consistency model**

Sophisticated package consistency model

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies
Limitation: Simplistic consistency model

Sophisticated package consistency model

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies
Limitation: Simplistic consistency model

Sophisticated package consistency model

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies

Import-Package: foo, bar
Limitation: Simplistic consistency model

Sophisticated package consistency model

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies
Limitation: Simplistic consistency model

Sophisticated package consistency model

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies

```
Import-Package: foo
Export-Package: bar;
uses:="foo"
```
**Limitation: Simplistic consistency model**

**Sophisticated package consistency model**

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies

```
Import-Package: foo
Export-Package: bar;
uses:="foo"
```
Limitation: Simplistic consistency model

Sophisticated package consistency model

- Exporters may declare package “uses” dependencies
  - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
- The framework must ensure that importers do not violate constraints implied by “uses” dependencies

**Example:**

```
Import-Package: foo
Export-Package: bar;
uses:="foo"
```
**Limitation: Simplistic consistency model**

- Sophisticated package consistency model
  - Exporters may declare package “uses” dependencies
    - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
  - The framework must ensure that importers do not violate constraints implied by “uses” dependencies

```
Import-Package: foo
Export-Package: bar;
uses:="foo"
```
Limitation: Coarse-grained package visibility rules
Limitation: Coarse-grained package visibility rules

Package filtering

- Exporters may declare that certain classes are included/excluded from the exported package
Limitation: Coarse-grained package visibility rules

Package filtering

Exporters may declare that certain classes are included/excluded from the exported package

```
Export-Package: foo;
   exclude:="*Impl",
   foo; friend="yes";
   mandatory:="friend"
```
Limitation: Coarse-grained package visibility rules

Package filtering

- Exporters may declare that certain classes are included/excluded from the exported package.

```
Import-Package: foo

B
  foo
(exclude:="*Impl")

A
  foo
  friend="yes"

C
  foo
  friend="yes"
  (include:="*")
```
Limitation: Coarse-grained package visibility rules

Package filtering

- Exporters may declare that certain classes are included/excluded from the exported package

```
Import-Package: foo

B
   foo
(exclude:="*Impl")

A
   foo

C
   foo
friend="yes"
```

```java
A
friend="yes"
```

(include:="*")

friend="yes"
Limitation: Coarse-grained package visibility rules

Package filtering

- Exporters may declare that certain classes are included/excluded from the exported package

```
Import-Package: foo
B foo
(exclude:="*Impl")
\(\rightarrow\)
A foo
\(\rightarrow\)
C foo
friend="yes"
```

```
Import-Package: foo;
friend="yes"
```
Limitation: Module content is not extensible
OSGi R4 Modularity (6)

- **Limitation:** Module content is not extensible

- Bundle fragments
  - A special bundle that attaches to a host bundle and uses the same class loader
    - Conceptually becomes part of the host bundle, allowing a logical bundle to be delivered in multiple physical bundles
**Limitation:** Module content is not extensible

**Bundle fragments**

- A special bundle that attaches to a host bundle and uses the same class loader
  - Conceptually becomes part of the host bundle

```
Fragment-Host: B
Export-Package: foo
Import-Package: baz

Bundle-SymbolicName: B
Export-Package: bar
Import-Package: woz
```
Limitation: Module content is not extensible

Bundle fragments
- A special bundle that attaches to a host bundle and uses the same class loader
  - Conceptually becomes part of the host bundle
Limitation: Module content is not extensible

Bundle fragments

- A special bundle that attaches to a host bundle and uses the same class loader
  - Conceptually becomes part of the host bundle
Limitation: Module content is not extensible

Bundle fragments

- A special bundle that attaches to a host bundle and uses the same class loader
  - Conceptually becomes part of the host bundle
- **Limitation:** *Module content is not extensible*

- Bundle fragments
  - A special bundle that attaches to a host bundle and uses the same class loader
  - Conceptually becomes part of the host bundle
Limitation: Package dependencies are not always appropriate
Limitation: Package dependencies are not always appropriate

Bundle dependencies
- Import everything that another, specific bundle exports
- Allows re-exporting
Limitation: Package dependencies are not always appropriate

Bundle dependencies

- Import everything that another, specific bundle exports
- Allows re-exporting

```
Bundle-SymbolicName: A
Export-Package: bar, foo

Require-Bundle: A
Export-Package: bar
```
Limitation: Package dependencies are not always appropriate

Bundle dependencies

- Import everything that another, specific bundle exports
- Allows re-exporting

```
Bundle-SymbolicName: A
Export-Package: bar, foo

Require-Bundle: A
Export-Package: bar
```
OSGi R4 Run-time Class Search Order

load class or find resource

belongs to a java.* package?
  yes
  delegate to parent class loader
  yes
  load class or find resource

no
  belongs to an imported package?
  yes
  delegate to exporter's class loader
  yes
  load class or find resource

no
  belongs to a package exported by a required bundle?
  yes
  delegate to exporters' class loaders
  yes
  load class or find resource

no
  search local class path

no
  found?
  yes
  lookup succeeded
  yes
  load class or find resource

no
  lookup failed

no
  lookup failed

no
  lookup failed

no
  lookup failed

yes
  lookup succeeded

no
  lookup succeeded

yes
  lookup succeeded
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.service.bar; version=1.1;
    uses:="org.foo.service"
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"
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.service.bar; version=1.1;
  uses:="org.foo.service"
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"
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"
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:=""

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

Provided package with arbitrary attribute and excluded classes
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"

Provided package with dependency on exported package
Challenges

- Manage the complexity
  - Maintain conceptual integrity
  - Keep the simple cases simple
  - Complexity should only be visible when it is required
  - Avoid bloat to support small devices
Challenges

- Manage the complexity
  - Maintain conceptual integrity
  - Keep the simple cases simple
  - Complexity should only be visible when it is required
  - Avoid bloat to support small devices

*The “good news” is that these changes generally only affect the dependency resolving algorithm*
Conclusions

- Java needs improved modularity support
  - We need to stop re-inventing the wheel
  - Improve application structure
  - Simplify deployment and management especially in technological areas where deployment is inherent
    - e.g., component orientation, extensible systems, and service orientation (to some degree)
- OSGi R1/R2/R3 were all steps in the right direction
- OSGi R4 goes even further in providing sophisticated Java modularity
  - OSGi technology is cited in JSR 277, an initiative by Sun to define a module system for Java, whose expert group includes OSGi members
Questions?