



LSR-Adèle



# Modularity for Java and How OSGi Can Help

DECOR

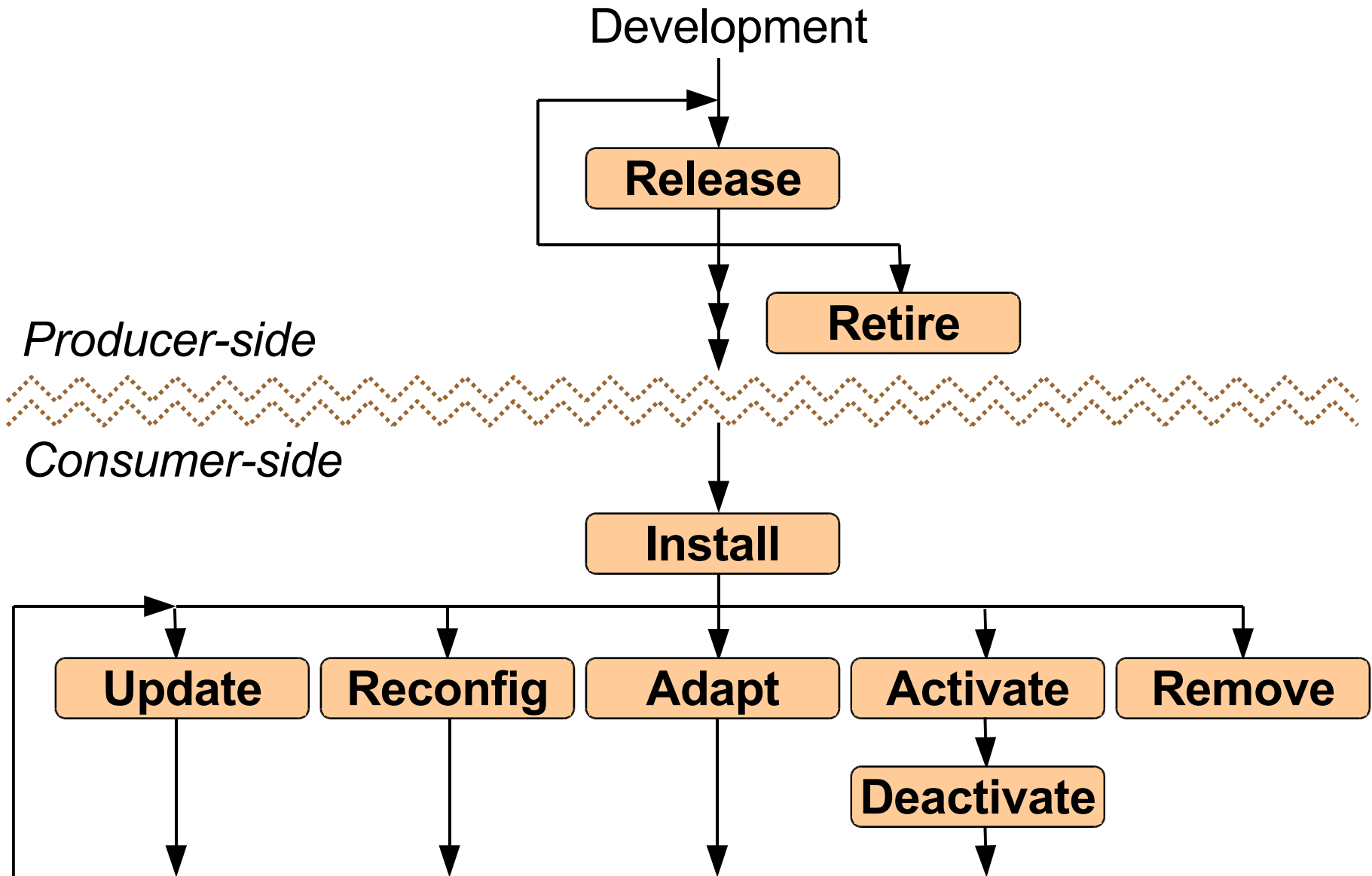
October 28<sup>th</sup>, 2004

**Richard S. Hall**

---



# Software Deployment Life Cycle



# Traditional Focus of Software Deployment

---



- Site (or host)
  - The “nuts and bolts” of deployment
    - Copying, extracting, configuring, localizing, state changes, and registrations, local policies

# Traditional Focus of Software Deployment

---



- Enterprise
  - Higher-level, sophisticated
    - Coordinated, multi-site, multi-domain, transactional, phased in, global policies
- Site (or host)
  - The “nuts and bolts” of deployment
    - Copying, extracting, configuring, localizing, state changes, and registrations, local policies



# A New Focus for Software Deployment

---

- Enterprise
  - Higher-level, sophisticated
    - Coordinated, multi-site, multi-domain, transactional, phased in, global policies
- Site (or host)
  - The “nuts and bolts” of deployment
    - Copying, extracting, configuring, localizing, state changes, and registrations, local policies
- Extensible systems
  - Fine-grained, dynamic reconfiguration
    - Requires similar aspects of site deployment, but tied more closely to execution environment



# Extensible Systems

---

- Focus of my research
- Popularized by Java because of its simple dynamic code loading mechanisms, .NET continues this trend
  - Extensible systems provide core functionality that is extended at run time by a *modularity* mechanism
- Software deployment at a lower level
  - i.e., a single process or virtual machine

# Software Deployment in Extensible Systems

---



- Modules must be packaged in an archive for *release* along with meta-data
  - Potentially supported by repositories or discovery services for advertising and dissemination
- Module archives must be *installed*, *reconfigured*, *adapted*, and *removed*
  - May include downloading, extracting, localizing, and versioning
  - Deployment activities occur at run time
- Module *activation* and *deactivation* are central activities
  - Modules are dynamically integrated at run time
  - Existing modules are impacted by continuous deployment activities

# Extensible Systems & Modularity



- 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](http://www.maths.bath.ac.uk/~jap/MATH0015/glossary.html)
  - For my purposes, this must also include the notion of independent packaging
- Extensible systems require some form of modularity mechanism
  - The Java world has many frameworks and systems reinventing this wheel
    - e.g., component frameworks, plugin mechanisms, application servers, etc.





# Importance of Modularity

---

- Not specific to extensible systems, impacts all systems
  - Improves system design
    - Helps developers achieve encapsulation and consistency
  - Brings deployment concepts to the forefront
    - Defines a unit of modularity at a minimum
    - May go as far as to define deployment processes for modules
  - Close relationship to execution environment's code loading mechanisms (e.g., class loading in Java)



# Modularity in Java

---

- Modularity support in Java is primitive
  - Closest analogy is the JAR file
    - Contains Java classes and resources
  - No real connection to deployment
  - No inherent support for dynamically extensible systems
- Lags behind .NET in certain areas
  - Assemblies are treated as a first class concept, as opposed to JAR files
    - Assemblies have explicit versioning rules
    - Assemblies can be shared via GAC
    - Assemblies auto-install is supported



# **Class Loaders Are Not Modules**

---

- Too low-level
  - Class loaders are details of Java execution environment
  - Do not provide proper abstraction
- Complicated to implement
- Difficult to reuse
- Not related to deployment
  - Not possible to package nor to perform software deployment processes on them



# Related Work for Java

---

- Module mechanisms
  - *MJ: A Rational Module System for Java and its Applications* (J. Corwin et al – IBM)
  - *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



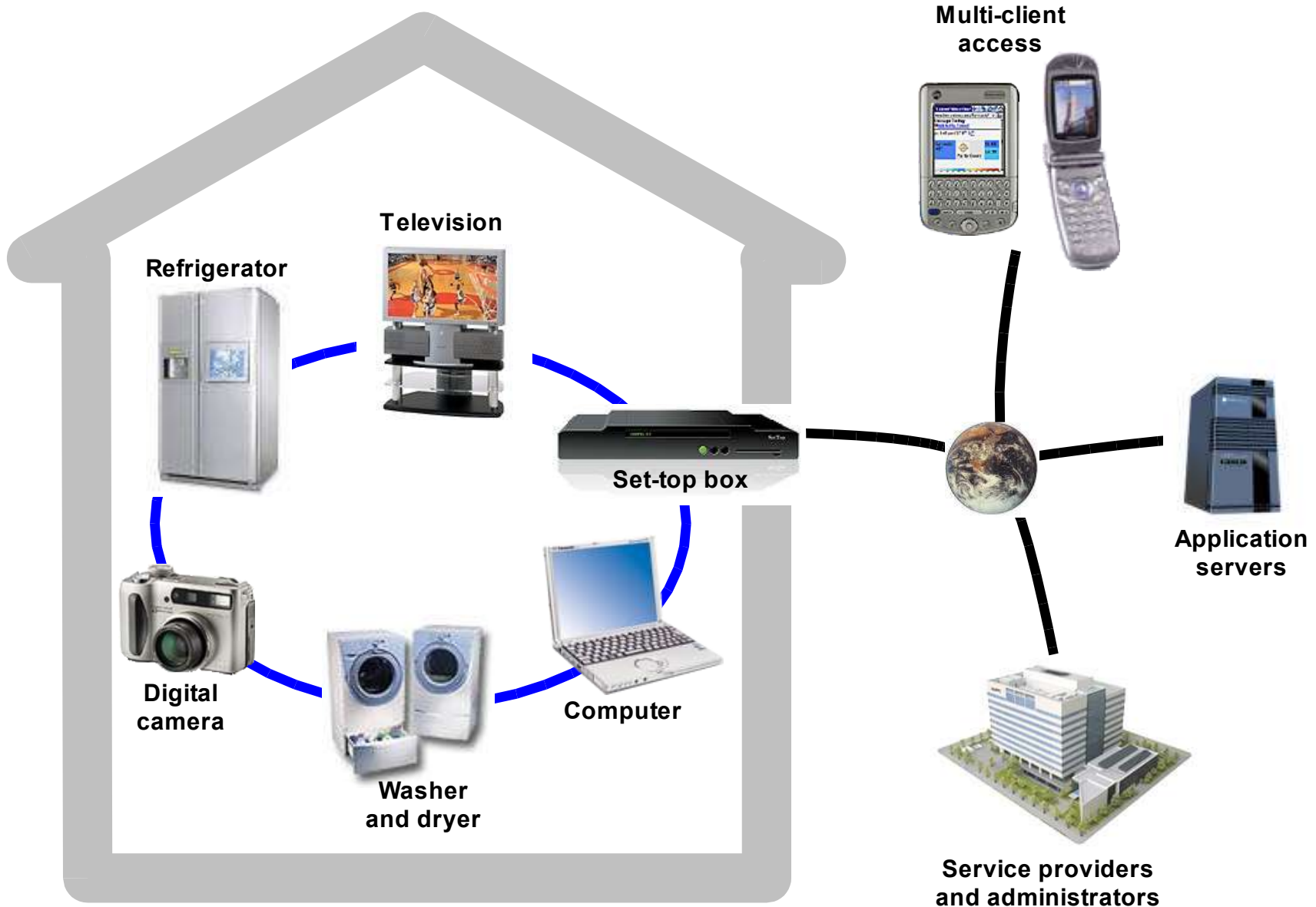
# Modularity Requirements

---

- 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
- Flexible, must support
  - Small to large systems
  - Static to dynamic systems
  - Arbitrary component models
  - Arbitrary interaction patterns



# OSGi Framework





# OSGi and Modularity

---

- Defines a very 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 bundle life cycle that relates directly to deployment processes
- Explicitly considers dynamic scenarios





# Deployment and the Bundle Life Cycle

---

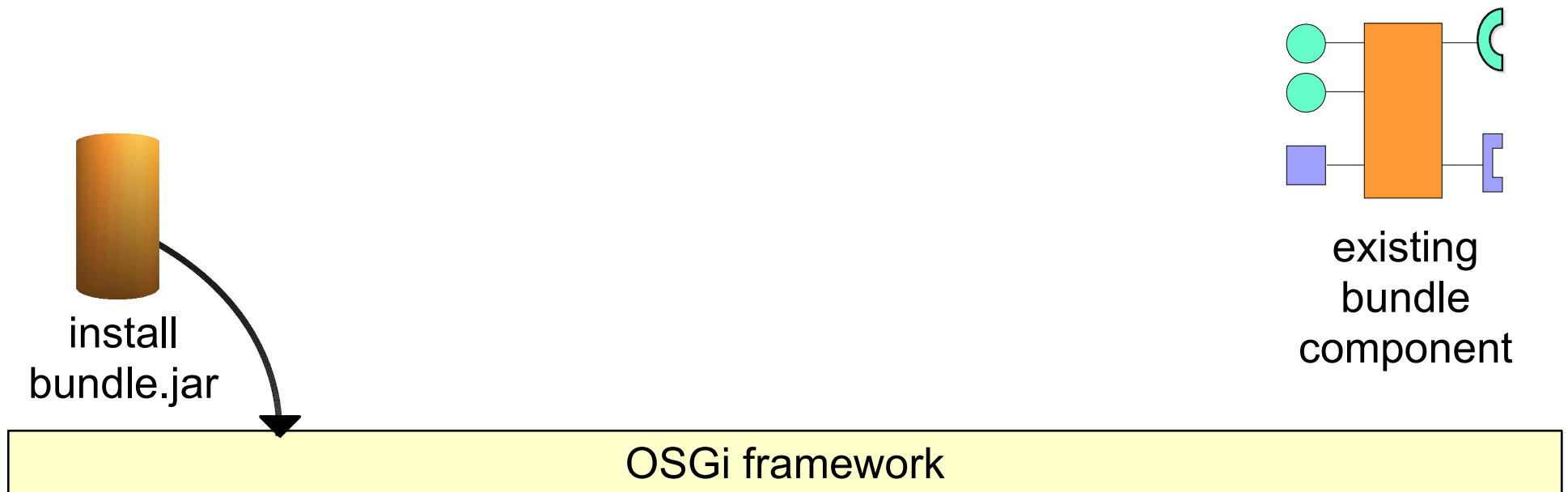


- *install* – retrieve bundle JAR file into framework, generally from a URL
- *resolve* – satisfy all package import dependencies, which enables export packages (implicit)
- *start / stop* – life cycle methods used to create and initialize components contained in bundle
- *update* – retrieve a new bundle JAR file, generally from a URL (deferred)
- *uninstall* – remove a bundle JAR file from framework (deferred)



# OSGi Component Model

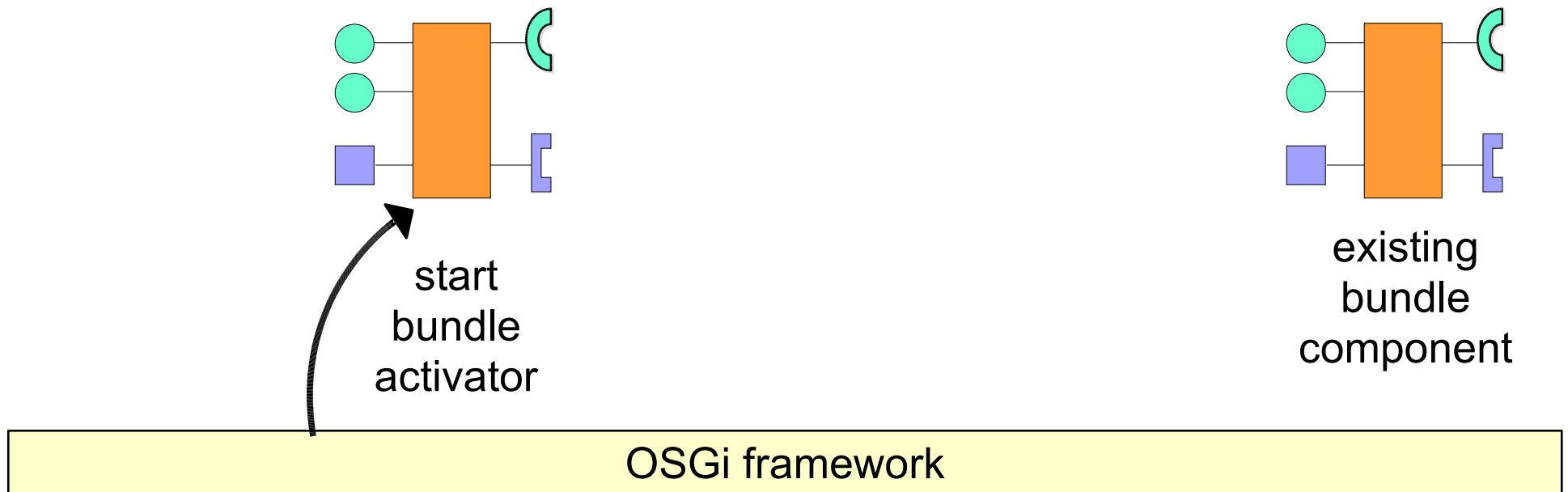
- By default, a single component is delivered in the bundle JAR file





# OSGi Component Model

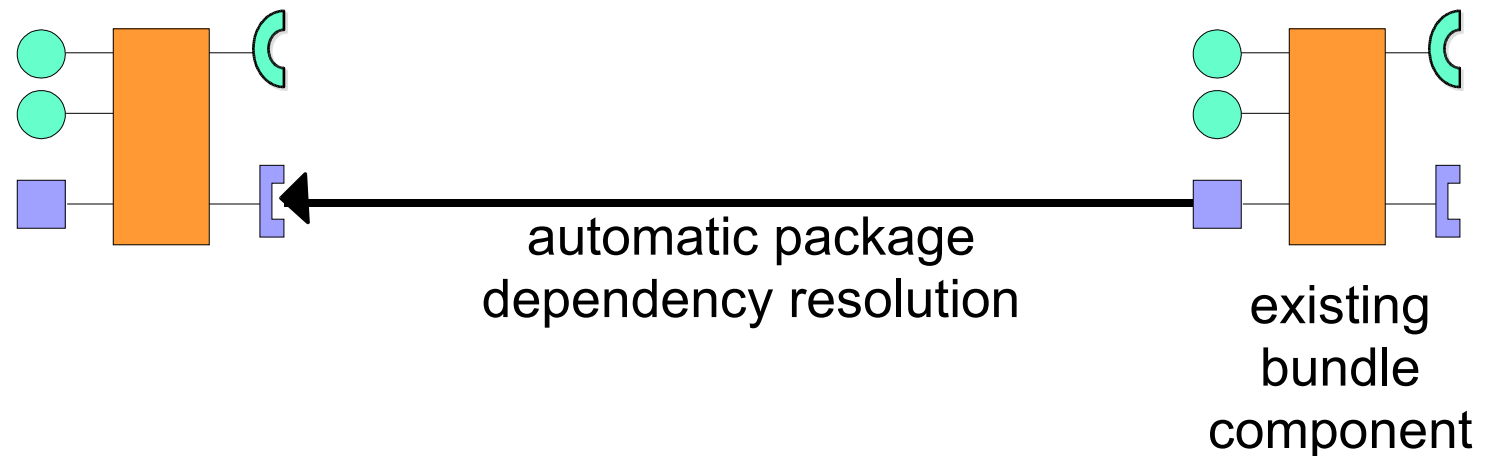
- By default, a single component is delivered in the bundle JAR file





# OSGi Component Model

- By default, a single component is delivered in the bundle JAR file

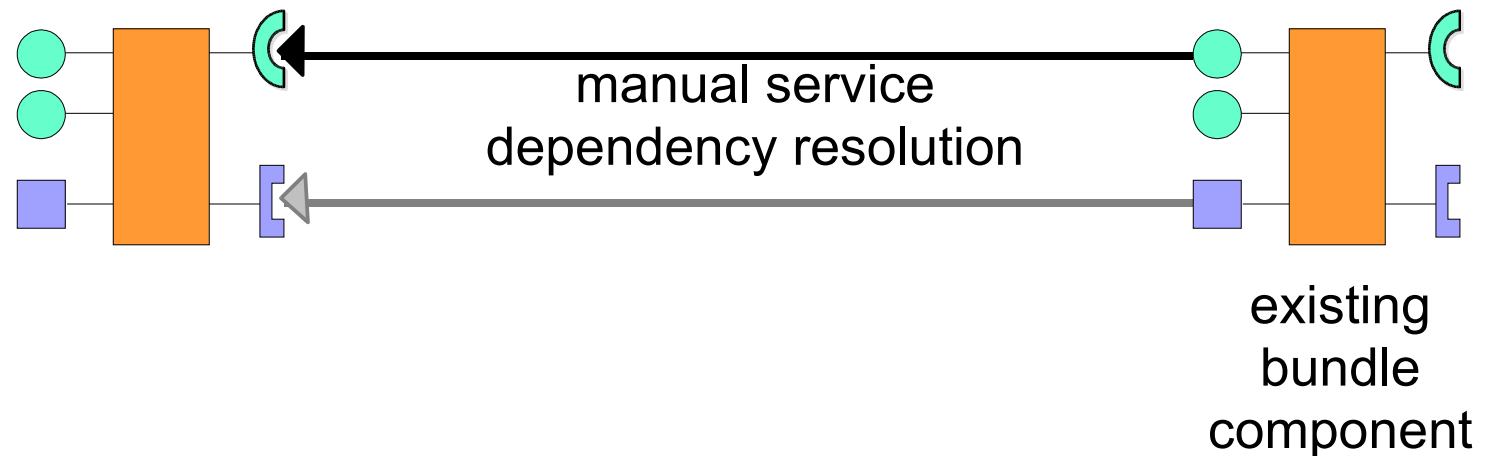


OSGi framework



# OSGi Component Model

- By default, a single component is delivered in the bundle JAR file



OSGi framework



# Benefits of OSGi Modularity

---

- Definitely more advanced than standard Java support for modularity
- In some ways, more advanced than .NET modularity
  - Better support for dynamics
  - More complete support for deployment life cycle
- But...



# OSGi Modularity Issues (1)

---

- 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
- Package provider selection is always anonymous
  - No way to influence selection



## **OSGi Modularity Issues (2)**

---

- Consistency model is simplistic and coarse grained
  - No way to declare dependencies among packages
  - No way to declare dependencies between a module's imports and exports
- Maintains Java's coarse-grained package visibility rules
  - Classes in a package are either completely visible to everyone or hidden





## To Be Fair

---

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



## To Be Clear

---

- The following proposed OSGi framework extensions are purely for discussion purposes
  - They are not endorsed by OSGi
  - The proposals and presented syntax are not currently OSGi compliant, nor may they ever be



# Potential OSGi Extensions (1)

---

- Explicit support for multiple versions of shared packages in memory at the same time
  - This is purely a general change to the prior OSGi philosophy
  - Has deep impact on service aspects as well as modularity
    - Service aspects are ignored here



## Potential OSGi Extensions (2)

---

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



## Potential OSGi Extensions (2)

---

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

*These first two extensions help to enable implementation package sharing*



## Potential OSGi Extensions (3)

---

- Arbitrary export/import attributes
  - Exporters may attach arbitrary attributes to their exports, importers can match against these arbitrary attributes
    - Some attributes may be declared as mandatory
      - Mandatory attributes allow exporters to essentially limit visibility of packages
  - Importers influence package selection using arbitrary attribute matching



## Potential OSGi Extensions (4)

---

- Improved package consistency model
  - Exporters may declare package groups
    - Packages in a group cannot be used a la carte
      - If you use one from the group, then if you use any of the others they must come from the same group
  - Exporters may declare that some imports are propagated through an export
    - Ensures that importers of a module's exports have consistent class definitions



## Potential OSGi Extensions (5)

---

- Improved Java package visibility rules via package filtering
  - Exporters may declare that certain classes are included/excluded from the exported package
  - When combined with mandatory attributes, allows exporters to provide midpoints between public and package private visibility





# Multiple Version Example

---

## **Bundle A**

```
import javax.servlet;  
    version="2.1.0"
```

## **Bundle B**

```
export javax.servlet;  
    version="2.1.0"
```

## **Bundle C**

```
import javax.servlet;  
    version="2.2.0"
```



# Multiple Version Example

---

## **Bundle A**

```
import javax.servlet;  
version="2.1.0"
```



## **Bundle B**

```
export javax.servlet;  
version="2.1.0"
```

Resolving *A* binds it to *B*'s export, like normal.

## **Bundle C**

```
import javax.servlet;  
version="2.2.0"
```



# Multiple Version Example

---

## Bundle A

```
import javax.servlet;  
version="2.1.0"
```



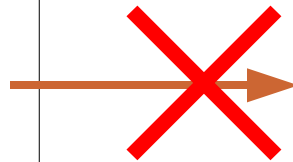
## Bundle B

```
export javax.servlet;  
version="2.1.0"
```

Resolving C is not possible, like normal.

## Bundle C

```
import javax.servlet;  
version="2.2.0"
```





# Multiple Version Example

---

## Bundle A

```
import javax.servlet;  
version="2.1.0"
```



## Bundle B

```
export javax.servlet;  
version="2.1.0"
```

If *D* is installed, then it is possible to resolve *C*.

## Bundle C

```
import javax.servlet;  
version="2.2.0"
```

## Bundle D

```
export javax.servlet;  
version="2.2.0"
```



# Multiple Version Example

---

## **Bundle A**

```
import javax.servlet;  
    version="2.1.0"
```



## **Bundle B**

```
export javax.servlet;  
    version="2.1.0"
```

## **Bundle C**

```
import javax.servlet;  
    version="2.2.0"
```



## **Bundle D**

```
export javax.servlet;  
    version="2.2.0"
```

This is possible due to support for multiple package versions in memory at the same time, but it provides a different visibility semantic than R3.



# Multiple Version Example

---

## **Bundle A**

```
import javax.servlet;  
version="2.1.0"
```



## **Bundle B**

```
export javax.servlet;  
version="2.1.0"
```

What happens if the framework is refreshed?

## **Bundle C**

```
import javax.servlet;  
version="2.2.0"
```



## **Bundle D**

```
export javax.servlet;  
version="2.2.0"
```



# Multiple Version Example

---

## **Bundle A**

```
import javax.servlet;  
version="2.1.0"
```

## **Bundle B**

```
export javax.servlet;  
version="2.1.0"
```

## **Bundle C**

```
import javax.servlet;  
version="2.2.0"
```

## **Bundle D**

```
export javax.servlet;  
version="2.2.0"
```

Every bundle ends up resolved to D, the newest version, just like normal for R3 semantics.

# Version Range and Arbitrary Attribute Example



## **Bundle A**

```
import javax.servlet;  
    version="[2.0.0,2.1.0)"
```

## **Bundle B**

```
export javax.servlet;  
    version="2.1.0"
```

## **Bundle C**

```
import javax.servlet;  
    version="2.2.0";  
    vendor="org.apache"
```

## **Bundle D**

```
export javax.servlet;  
    version="2.2.0"
```

Version ranges and arbitrary attributes influence provider selection

## **Bundle E**

```
export javax.servlet;  
    version="2.2.0";  
    vendor="org.apache"
```



# Version Range and Arbitrary Attribute Example



## **Bundle A**

```
import javax.servlet;  
version="[2.0.0,2.1.0)"
```

## **Bundle B**

```
export javax.servlet;  
version="2.1.0"
```

## **Bundle C**

```
import javax.servlet;  
version="2.2.0";  
vendor="org.apache"
```

## **Bundle D**

```
export javax.servlet;  
version="2.3.0"
```

## **Bundle E**

```
export javax.servlet;  
version="2.2.0";  
vendor="org.apache"
```

Due to version ranges, A can only bind to B. Due to attribute matching, C can only bind to E.



# Package Grouping Example

---

## **Bundle A**

```
import javax.servlet;  
    javax.servlet.http;  
version="2.2.0"
```

## **Bundle B**

```
export javax.servlet;  
    version="2.2.0"
```

## **Bundle D**

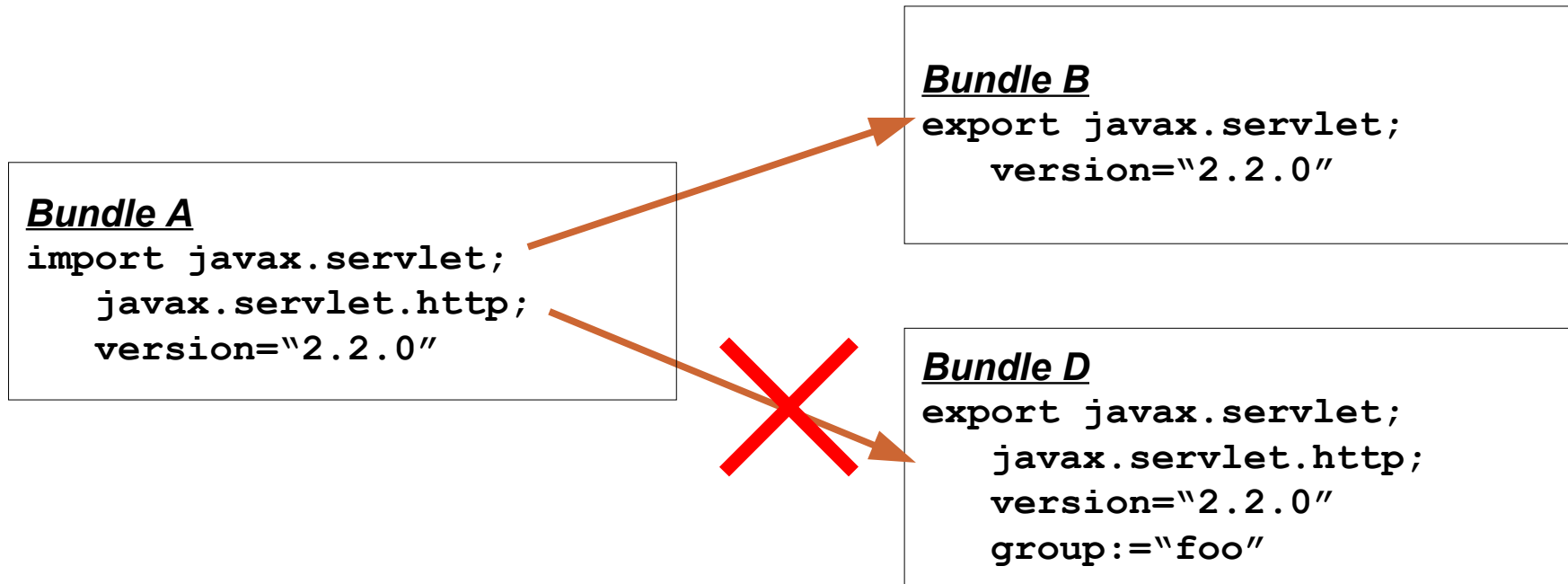
```
export javax.servlet;  
    javax.servlet.http;  
    version="2.2.0"  
    group:="foo"
```

Package grouping is a directive to the dependency resolver to help it maintain consistency when packages cannot be used independently.



# Package Grouping Example

---



If the resolver attempts to resolve *A*'s dependency on `javax.servlet` to *B*, then it will fail when trying to resolve `javax.servlet.http`.



# Package Grouping Example

---

## **Bundle A**

```
import javax.servlet;  
    javax.servlet.http;  
version="2.2.0"
```

## **Bundle B**

```
export javax.servlet;  
    version="2.2.0"
```

## **Bundle D**

```
export javax.servlet;  
    javax.servlet.http;  
    version="2.2.0"  
    group:="foo"
```

In this case, the only option is to resolve A to both exports of D.



# Package Propagation Example

---

## **Bundle A**

```
export javax.servlet;  
    version="[2.2.0,2.2.0]"
```

## **Bundle B**

```
import javax.servlet;  
    version="2.2.0"  
export org.osgi.service.http;  
    version="1.1.0";  
    propagates:="javax.servlet"
```

## **Bundle C**

```
import org.osgi.service.http,  
    javax.servlet
```

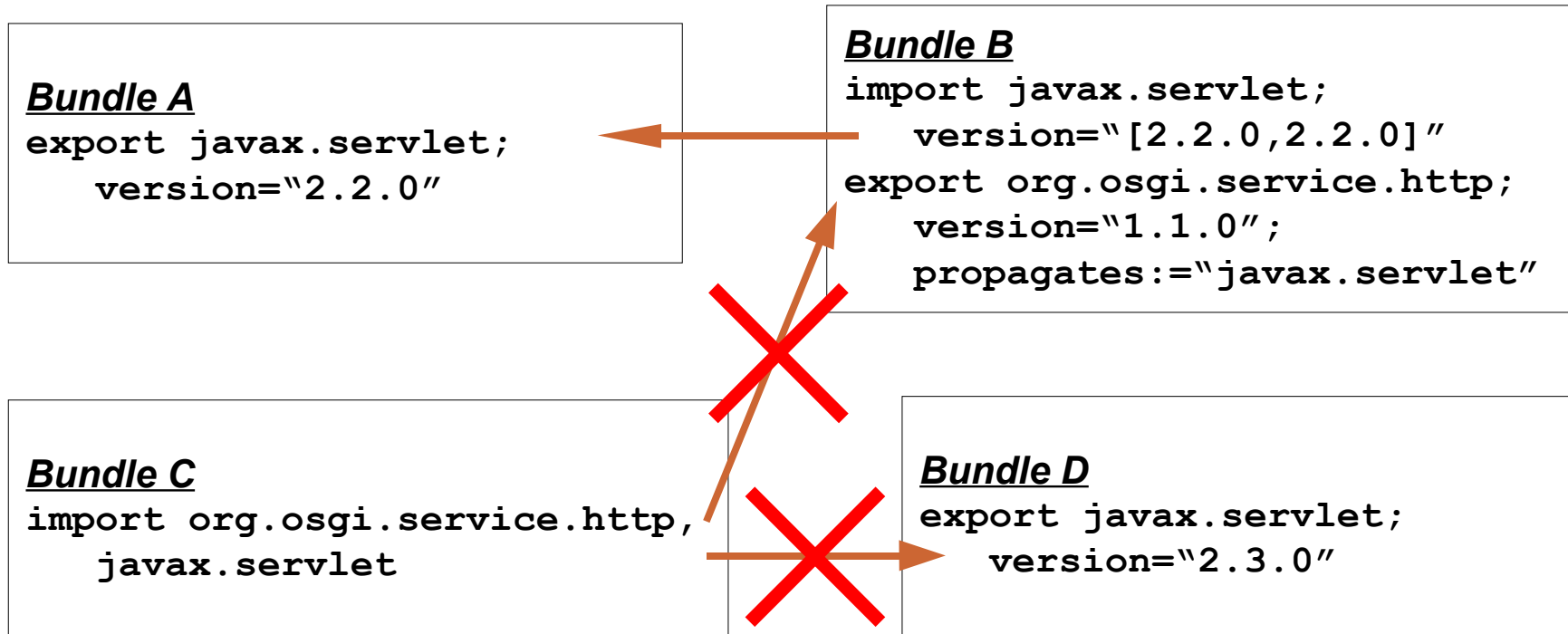
## **Bundle D**

```
export javax.servlet;  
    version="2.3.0"
```

Package propagation is a directive to the dependency resolver to help it maintain consistency when module imports are visible via its exports (i.e., public versus private imports).



# Package Propagation Example



It is not possible for C to be resolved to the newest version of `javax.servlet` if it gets `org.osgi.service.http` from C, because it propagates `javax.servlet` to importers.



# Package Propagation Example

## Bundle A

```
export javax.servlet;  
    version="2.2.0"
```

## Bundle B

```
import javax.servlet;  
    version="[2.2.0,2.2.0]"  
export org.osgi.service.http;  
    version="1.1.0";  
propagates:="javax.servlet"
```

## Bundle C

```
import org.osgi.service.http,  
    javax.servlet
```

## Bundle D

```
export javax.servlet;  
    version="2.3.0"
```

The only option is to resolve C to the same version of `javax.servlet` that is used by *B*.



# Package Propagation Example

## Bundle A

```
export javax.servlet;  
    version="2.2.0"
```

## Bundle B

```
import javax.servlet;  
    version="[2.2.0,2.2.0]"  
export org.osgi.service.http;  
    version="1.1.0";  
    propagates:="javax.servlet"
```

## Bundle C

```
import org.osgi.service.http,  
    javax.servlet;  
    version= "2.3.0"
```

## Bundle D

```
export javax.servlet;  
    version="2.3.0"
```

Of course, given certain sets of constraints, such as if *C* requires a version of `javax.servlet` that is different than the one used by *B*, then it will not be possible to resolve *C*.





# Package Propagation Example

## Bundle A

```
export javax.servlet;  
version="2.2.0"
```

## Bundle B

```
import javax.servlet;  
version="[2.2.0,2.2.0]"  
export org.osgi.service.http;  
version="1.1.0";  
propagates:="javax.servlet"
```

## Bundle C

```
import org.osgi.service.http,  
javax.servlet
```

## Bundle D

```
export javax.servlet;  
version="2.3.0"
```

In practice, though, importers of packages that propagate other packages, should not specify constraints on the propagated packages so they automatically resolve to the appropriate package.



# Package Filtering Example

---

## **Bundle A**

```
import org.foo;  
    version="1.1.0"
```

## **Bundle C**

```
import org.foo;  
    attribute="value"
```

## **Bundle B**

```
export org.foo; version="1.1.0";  
    exclude:="org.foo.Private",  
    org.foo; version="1.1.0";  
    attribute="value";  
    mandatory:="attribute"
```

Package filtering is a directive to the underlying module layer to limit class visibility beyond what is possible with standard Java constructs. Combined with mandatory attributes, it is possible to have a “friend” concept.



# Package Filtering Example

---

## **Bundle A**

```
import org.foo;  
    version="1.1.0"
```

## **Bundle A**

```
import org.foo;  
    attribute="value"
```

## **Bundle B**

```
export org.foo; version="1.1.0";  
    exclude:="org.foo.Private",  
    org.foo; version="1.1.0";  
    attribute="value";  
    mandatory:="attribute"
```

In order to get visibility to all classes in the package, a “friend” must specify the mandatory attribute. This is not completely strict, security must be used if guarantees are required.



# Why the Complexity?

---

- Sharing of implementation packages leads to complex possibilities
  - Dependencies are more precise and rigid, unlike specification dependencies
  - Results in the need to allow multiple package versions in memory
- A generic modularity mechanism must have sophisticated constructs
  - Necessary to support complex and/or legacy systems
- It is unavoidable in extensible systems
  - Support for these issues must be addressed, either ad hoc or systematically



# Challenges

---

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

*The “good news” so far, is that these proposed changes generally only affect the dependency resolving algorithm.*



# Conclusions

---

- Extensible systems are very popular and highlight the need for modularity mechanisms
- Java lacks good modularity mechanisms, lags behind .NET
- Nearly all applications could benefit from improved modularity support in Java
- The OSGi framework provides a starting point for Java modularity, but does not go far enough
- It is possible to extend OSGi to support sophisticated modularity constructs