OSGi R4 Service Platform: Java Modularity and Beyond

Dr. Richard S. Hall

akquinet fws, Berlin
March 21st, 2007

Agenda

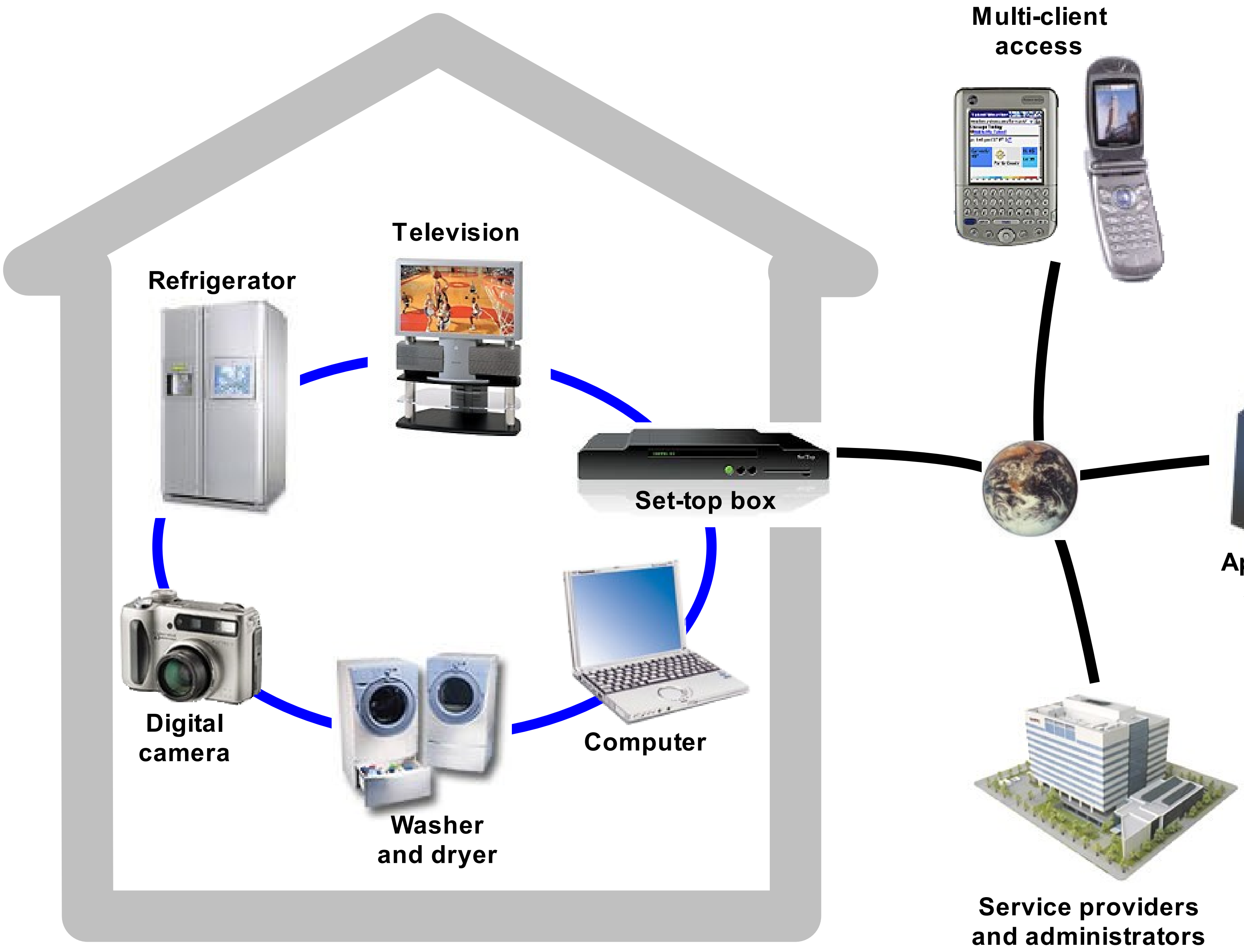
- OSGi R4 Service Platform Overview
- OSGi as a Java Modularity Layer
  - Majority of the presentation
- OSGi as a Service-Oriented Application Framework
- Apache Felix Overview
- Conclusion

OSGi Service Platform Overview

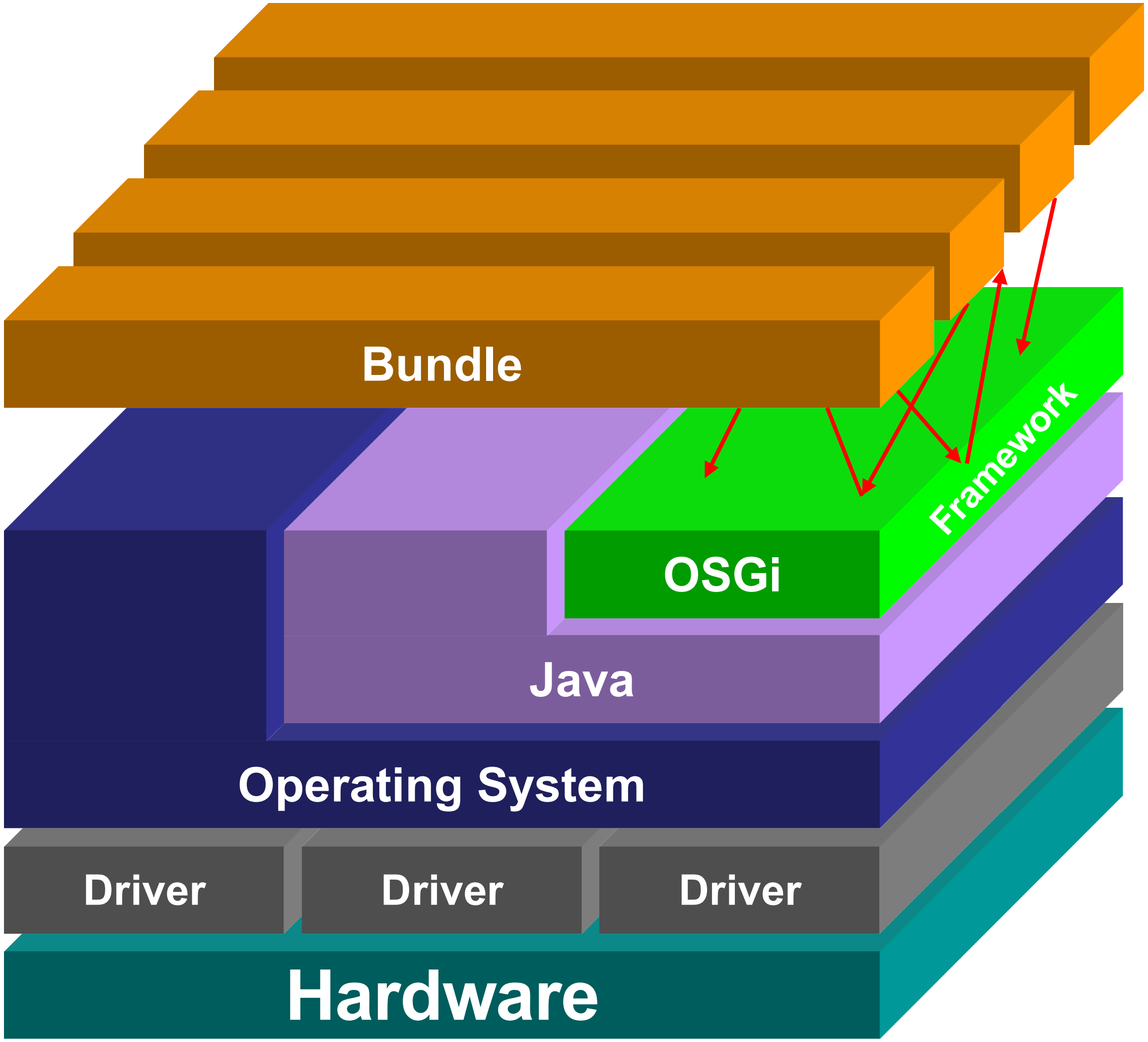
OSGi Alliance

- Formerly known as the Open Services Gateway Initiative
- Industry consortium
- Defines OSGi Service Platform
  - Framework specification for hosting dynamically downloadable services
  - Standard service specifications
- Several expert groups define the specifications
  - Core Platform Expert Group (CPEG) – framework
  - Mobile Expert Group (MEG) – mobile telephony
  - Vehicle Expert Group (VEG) – automobile
  - Enterprise Expert Group (EEG) – enterprise issues

Original Home Services Gateway Vision



OSGi Architectural Overview



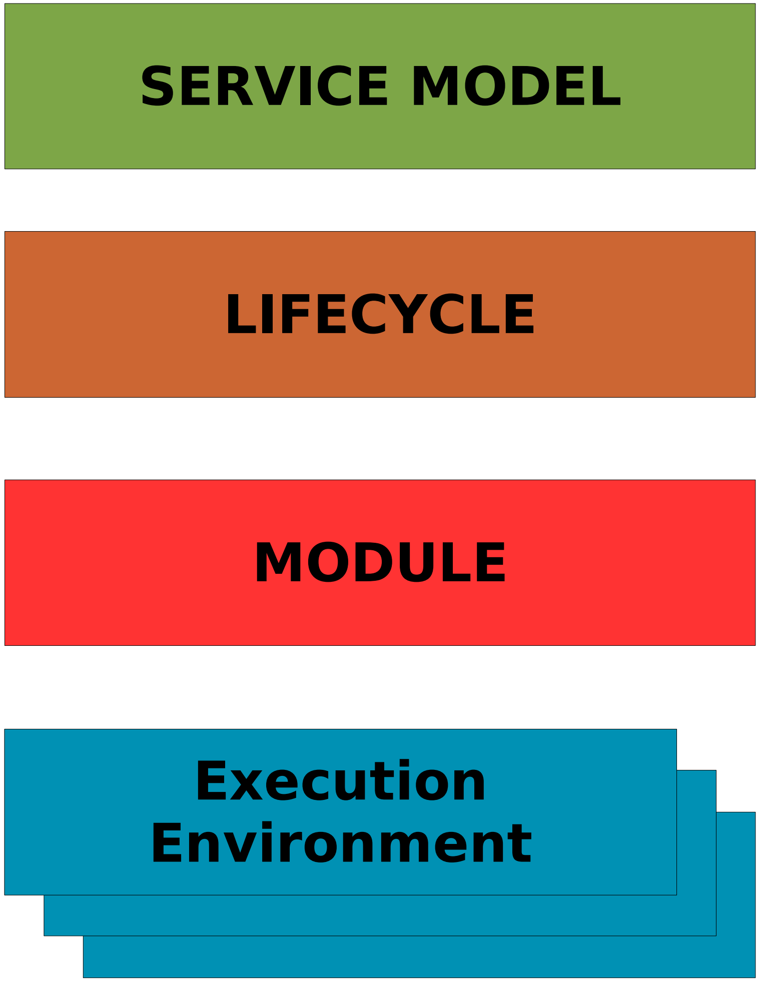
OSGi Framework (1/2)

- Component-oriented framework
  - *Bundles* (i.e., modules/components)
  - Package sharing and version management
  - Life-cycle management and notification
- Service-oriented architecture
  - Publish/find/bind intra-VM service model
- Open remote management architecture
  - No prescribed policy or protocol

OSGi Framework (2/2)

- Runs multiple applications and services
- Single VM instance
- Separate class loader per bundle
  - Class loader graph
  - Independent namespaces
  - Class sharing at the Java package level
- Java Permissions to secure framework
- Explicitly considers dynamic scenarios
  - Run-time install, update, and uninstall of bundles

OSGi Framework Layering



SERVICE MODEL

LIFECYCLE

MODULE

Execution Environment

L3 – Provides a publish/find/bind service model to decouple bundles

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

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

L0 -
•OSGi Minimum Execution Environment
•CDC/Foundation
•JavaSE

OSGi Momentum

- OSGi technology has moved beyond original target domain
- Initial success story was Eclipse RCP (three years ago)
- More recent success stories in enterprise scenarios
  - IBM
  - Spring
  - BEA
  - Oracle
  - JBoss
  - SAP (perhaps?)

OSGi as a Java Modularity Layer

Standard Java Modularity Limitations (1/2)

- 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 if finds, which can lead to shadowing or mixing of versions
  - JAR files can provide sealing

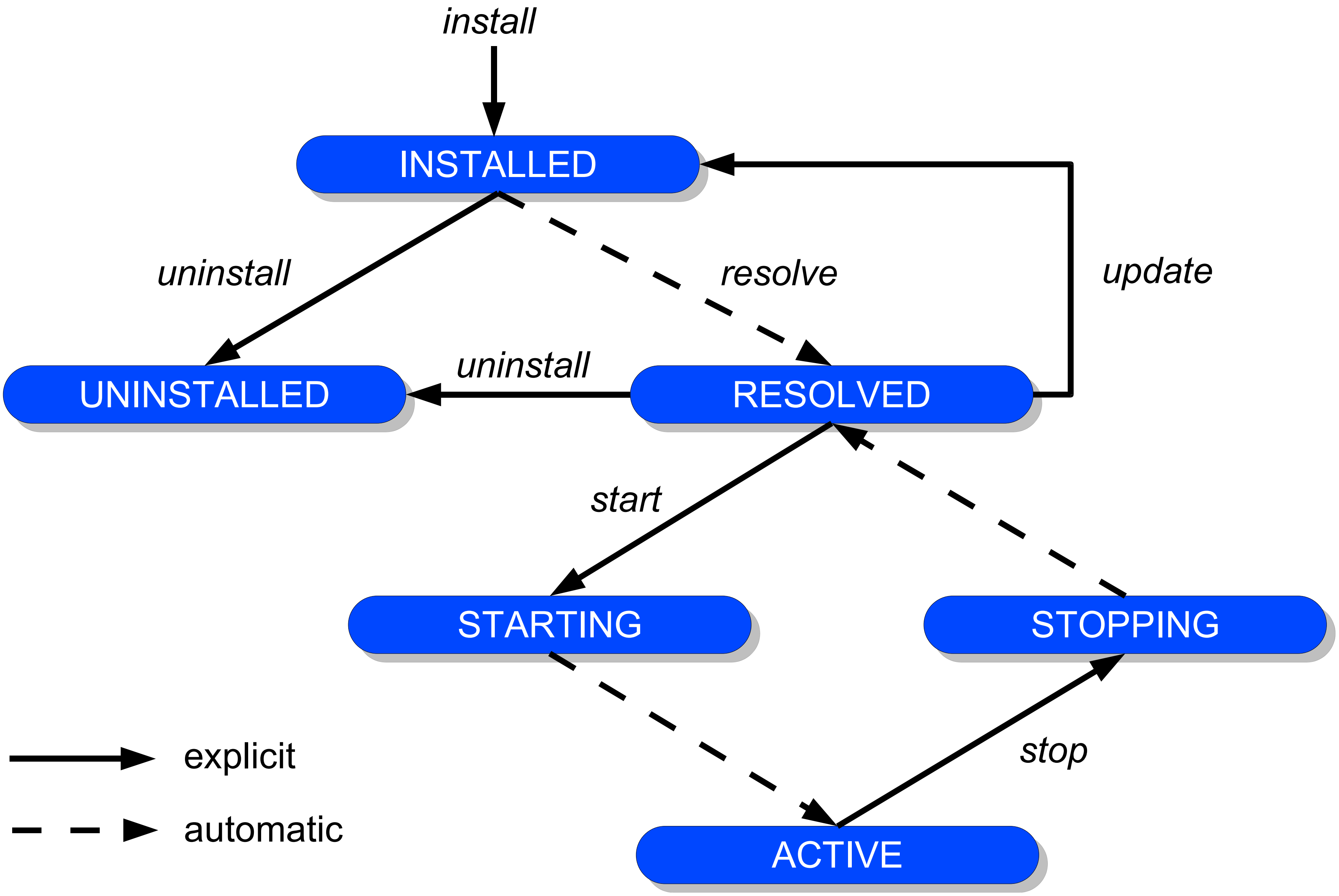
Standard Java Modularity Limitations (2/2)

- 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 are too fine grained, packages are too simplistic, class loaders are too low level
  - JAR file is best candidates, but still inadequate
  - Modularity is a second-class concept

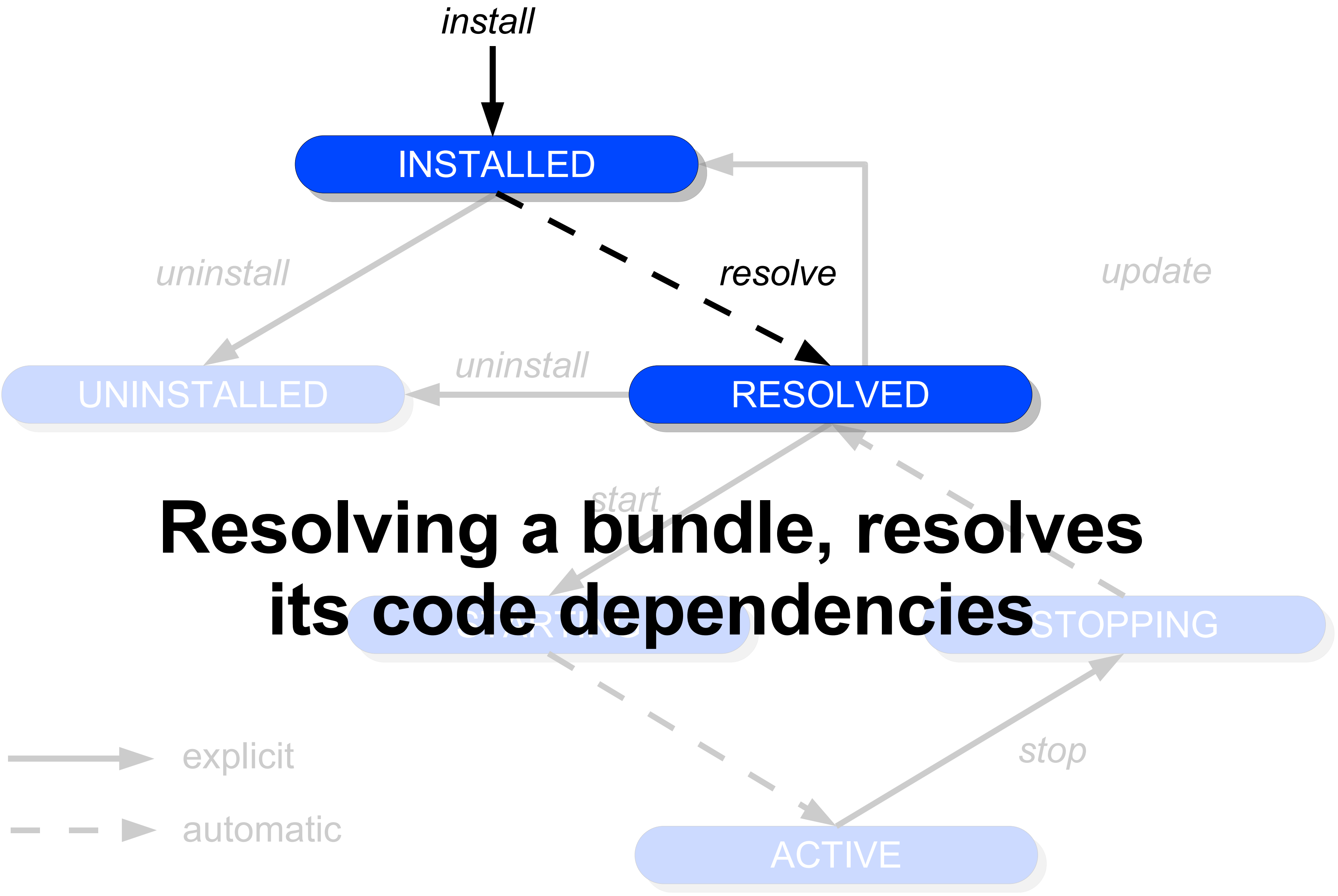
OSGi Framework Modularity Support

- Resolves nearly all deficiencies associated with standard Java support for modularity
  - The OSGi bundle defines an explicit boundary for a module
  - Bundle metadata explicitly declares versioned dependencies on other code
  - Framework automatically manages bundle code dependencies
  - Framework enforces sophisticated consistency rules for class loading within and among bundles

Bundle Life Cycle



Bundle Life Cycle



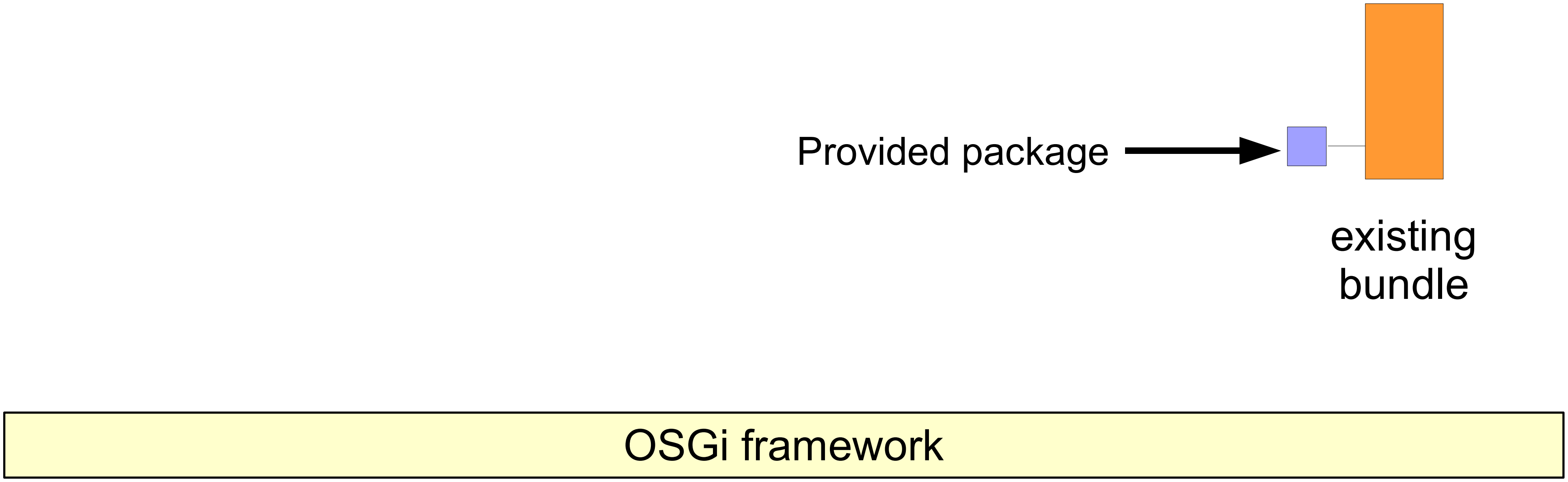
Resolving a bundle, resolves its code dependencies

Bundle Dependency Resolution

- The framework automatically resolves dependencies before a bundle is used
  - Matches bundle’s requirements to providers of those requirements
    - Package imports/exports
    - Explicit bundle dependencies
    - Bundle fragment dependencies
  - Ensures consistency of requirements with respect to versions and other constraints
- If a bundle cannot be successfully resolved, then it cannot be used

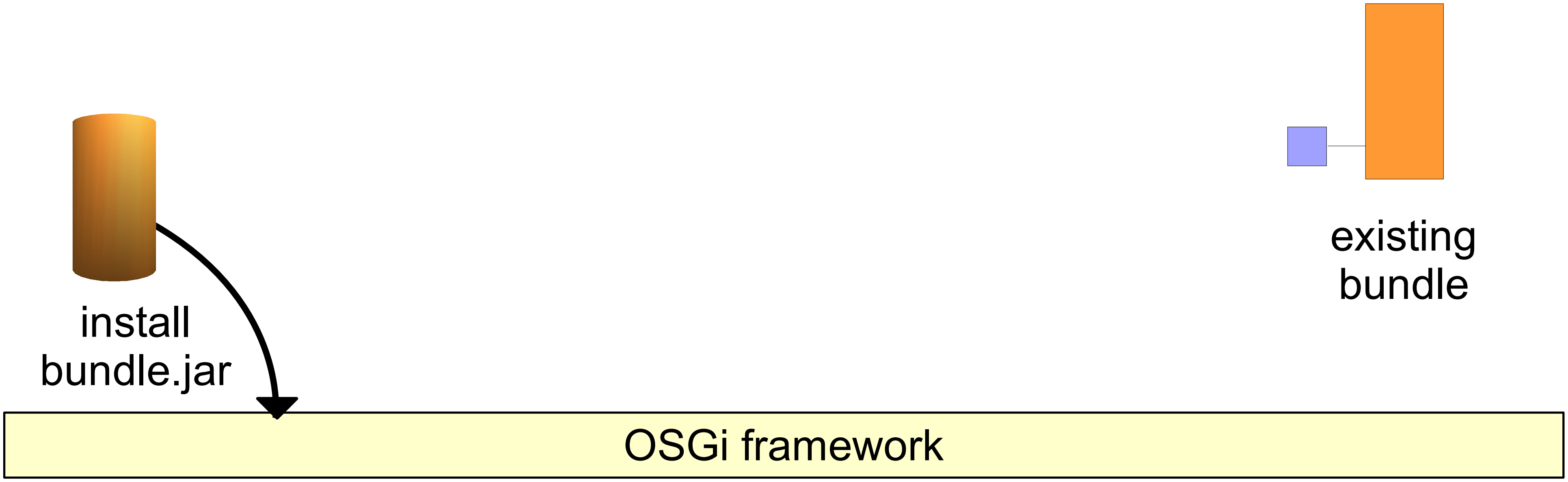
Dependency Resolution Illustration

- A bundle represents a module contained in a JAR file



Dependency Resolution Illustration

- A bundle represents a module contained in a JAR file



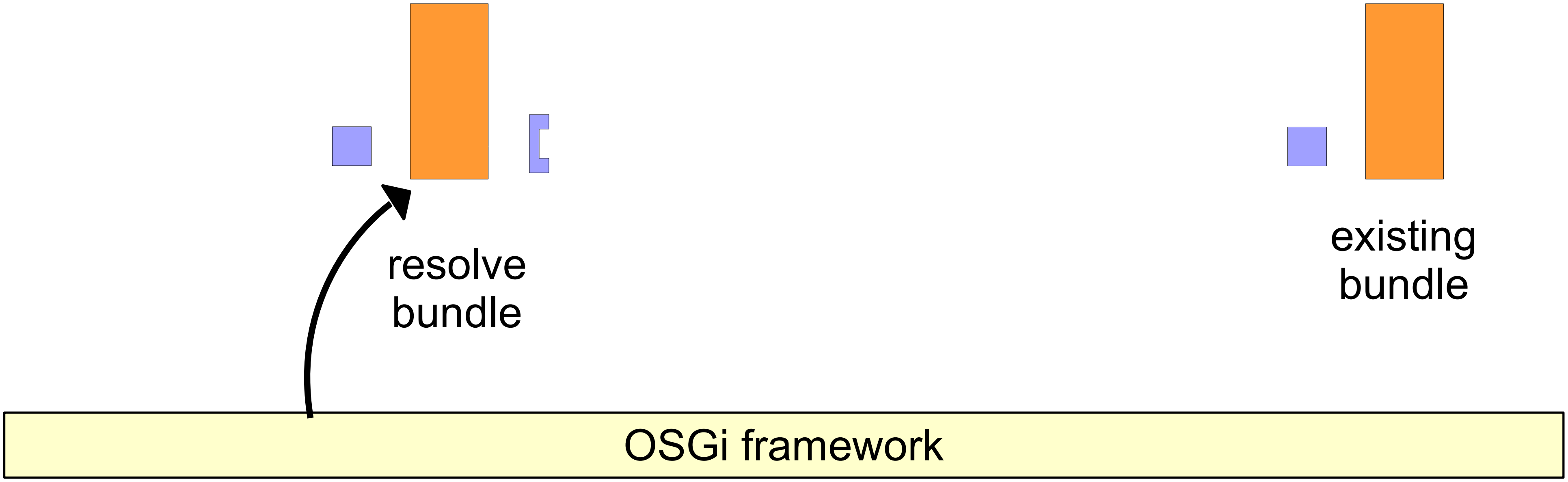
install bundle.jar

existing bundle

OSGi framework

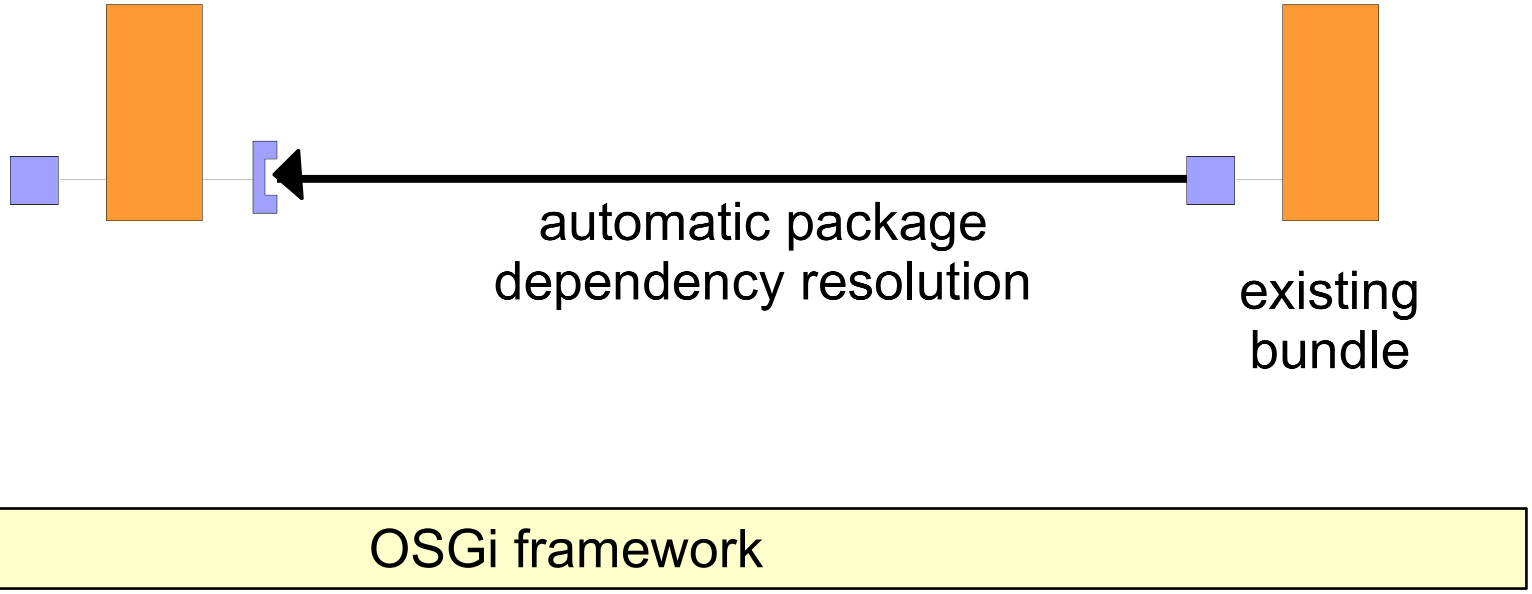
Dependency Resolution Illustration

- A bundle represents a module contained in a JAR file



Dependency Resolution Illustration

- A bundle represents a module contained in a JAR file



OSGi R4 Modularity Details (1/7)

- Multi-version support (i.e., side-by-side versions)
  - Possible to have more than one version of a shared package in memory at the same time
  - Allows multiple applications to run in the same VM or a subcomponents of a single application to depend on different versions of the same libraries
  - Has impacts on the service-oriented aspects of the OSGi framework
    - For a given bundle, the service registry is implicitly partitioned according to the package versions visible to it

OSGi R4 Modularity Details (2/7)

- Explicit code boundaries and dependencies
  - Explicitly expose packages from a bundle (i.e., export)
    - Exporters export precise package versions
  - Explicitly declare dependencies on external packages (i.e., import)
    - Importers may specify an open or closed version range

OSGi R4 Modularity Details (2/7)

- Explicit code boundaries and dependencies
  - Explicitly expose packages from a bundle (i.e., export)
    - Exporters export precise package versions
  - Explicitly declare dependencies on external packages (i.e., import)
    - Importers may specify an open or closed version range

```
Export-Package: bar; version=“1.0.0”
Import-Package: foo; version=“[1.0.0,1.5.0)”
```

OSGi R4 Modularity Details (2/7)

- Explicit code boundaries and dependencies
  - Explicitly expose packages from a bundle (i.e., export)
    - Exporters export precise package versions
  - Explicitly declare dependencies on external packages (i.e., import)
    - Importers may specify an open or closed version range

```
Export-Package: bar; version=“1.0.0”
Import-Package: foo; version=“[1.0.0,1.5.0)”
```

- Support for various sharing policies, e.g,
  - Implementation package with limited backwards compatibility
  - Specification packages with defined backwards compatibility

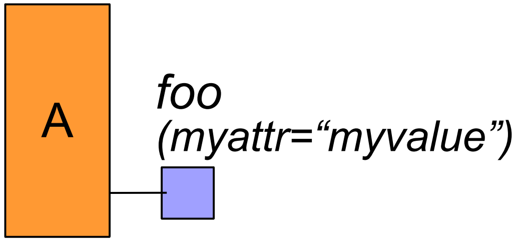
OSGi R4 Modularity Details (3/7)

- Arbitrary export/import attributes for more control
  - 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

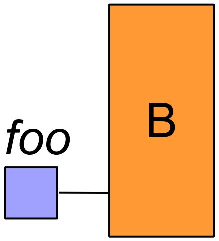
OSGi R4 Modularity Details (3/7)

- Arbitrary export/import attributes for more control
  - 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

```
Export-Package: foo;
 version=“1.0.0”;
 myattr=“myvalue”
```



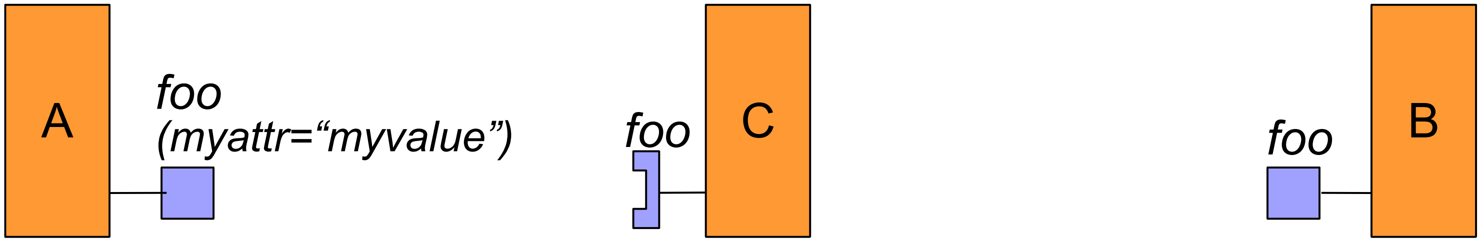
```
Export-Package: foo;
 version=“1.0.0”
```



OSGi R4 Modularity Details (3/7)

- Arbitrary export/import attributes for more control
  - 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

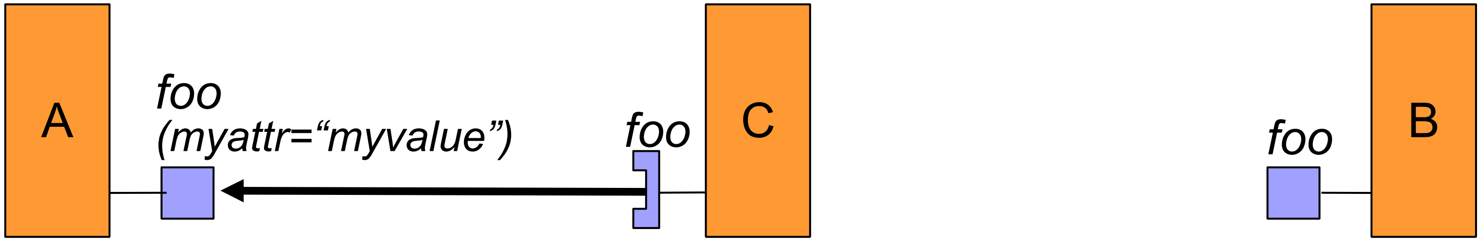
```
Import-Package: foo;
 version=“1.0.0”;
 myattr=“myvalue”
```



OSGi R4 Modularity Details (3/7)

- Arbitrary export/import attributes for more control
  - 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"
```

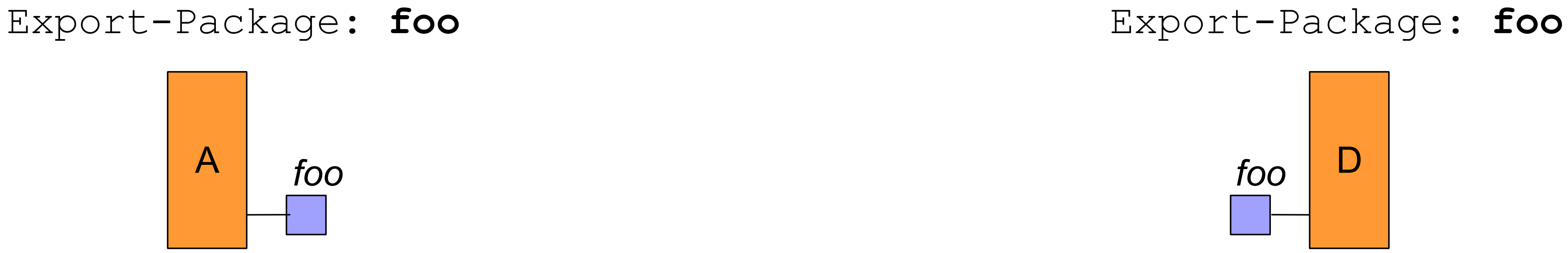


OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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

OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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



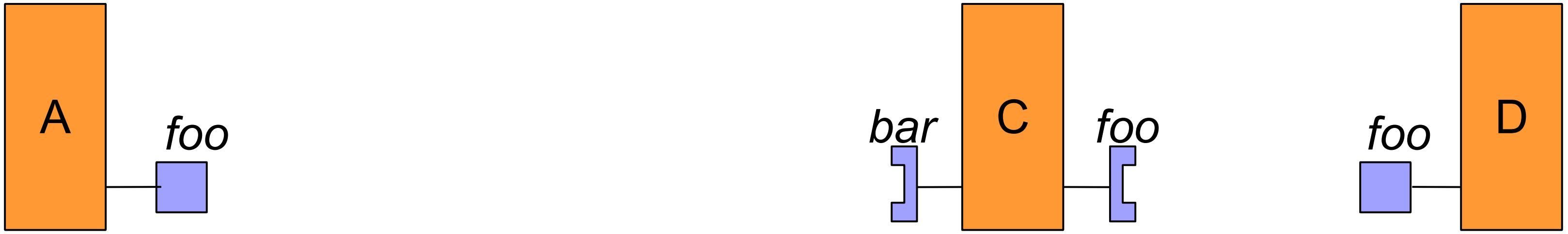
`Export-Package: **foo**`

`Export-Package: **foo**`

OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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

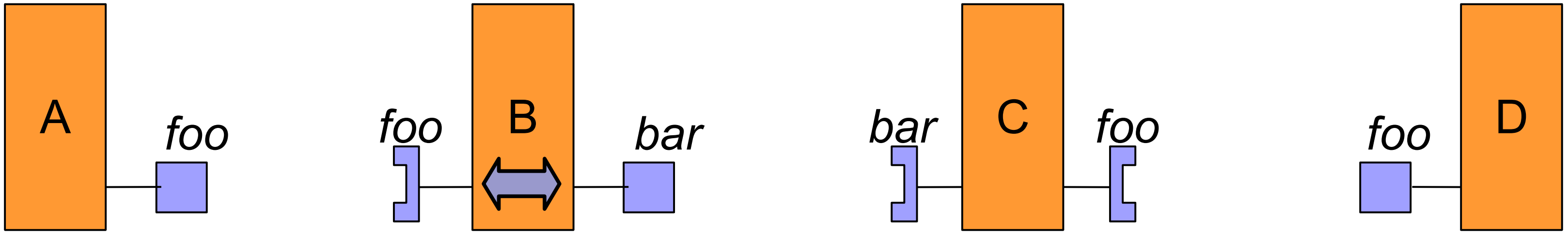
`Import-Package: **foo, bar**`



OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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

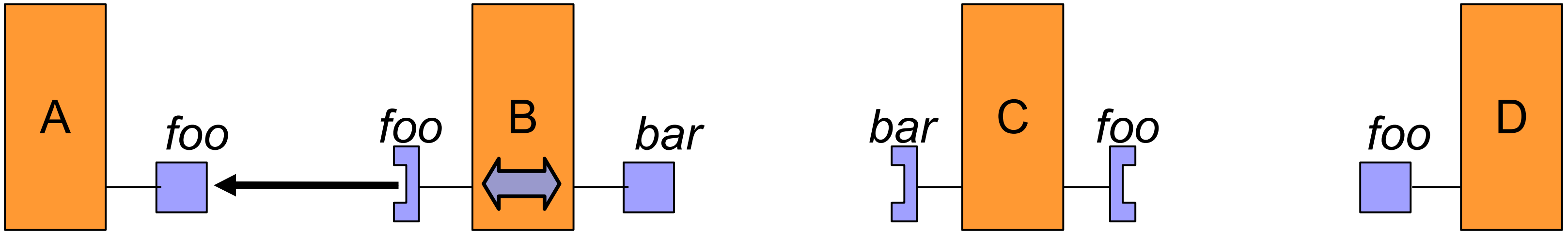
```
Import-Package: foo
Export-Package: bar;
 uses:=“foo”
```



OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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

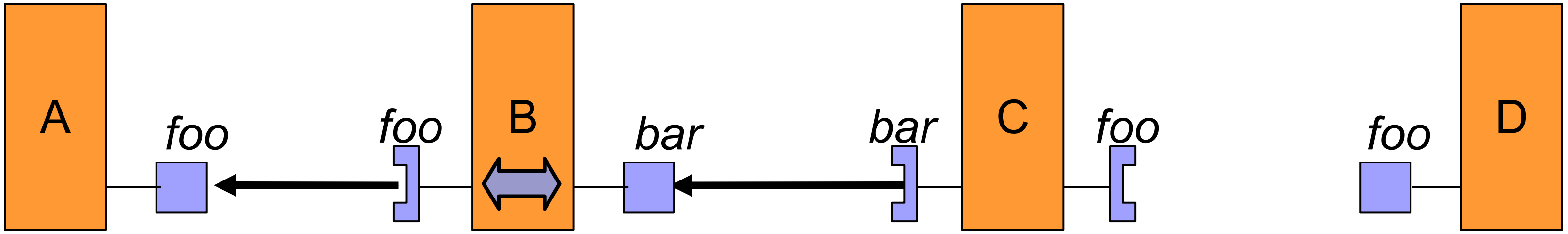
```
Import-Package: foo
Export-Package: bar;
 uses:=“foo”
```



OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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

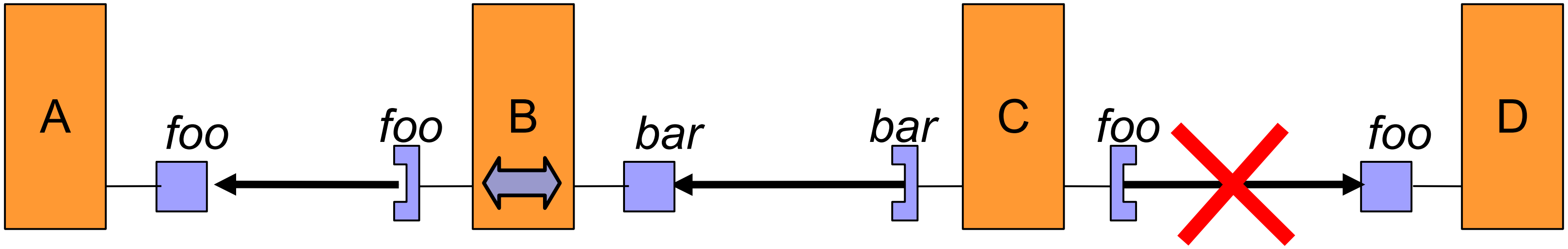
```
Import-Package: foo
Export-Package: bar;
 uses:="foo"
```



OSGi R4 Modularity Details (4/7)

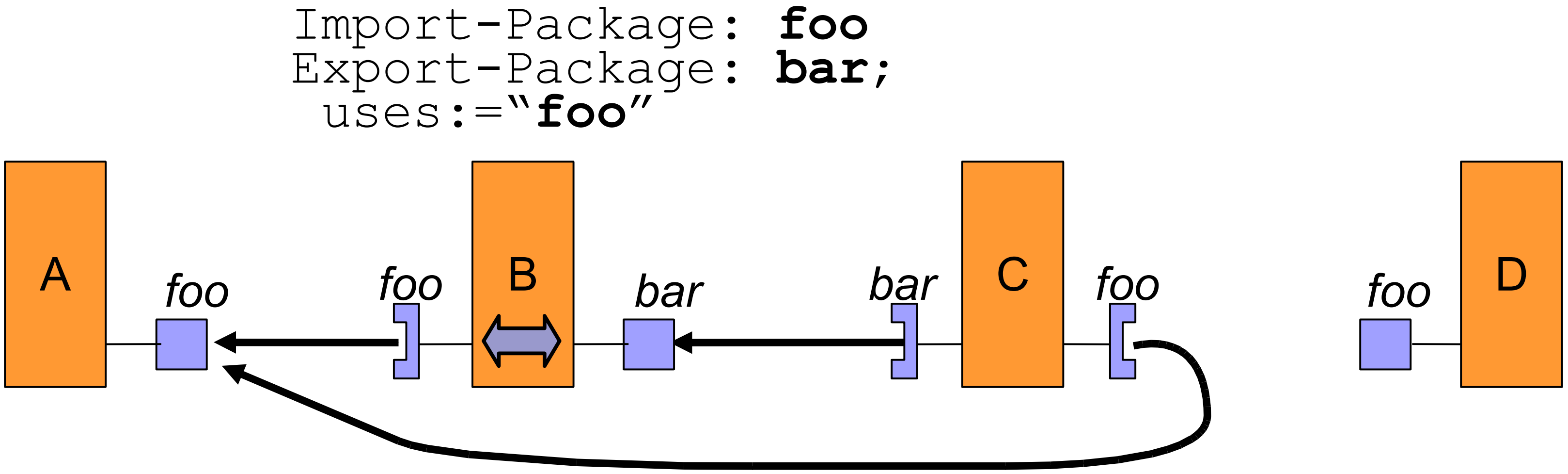
- Sophisticated class space consistency model
  - In addition to dependency resolution
  - Exporters may declare package “uses” dependencies
    - Exported packages express dependencies on imported or other exported packages, which constrain the resolve process
  - The framework must ensure that importers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

```
Import-Package: foo
Export-Package: bar;
 uses:=“foo”
```



OSGi R4 Modularity Details (4/7)

- Sophisticated class space consistency model
  - In addition to dependency resolution
  - 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”
```

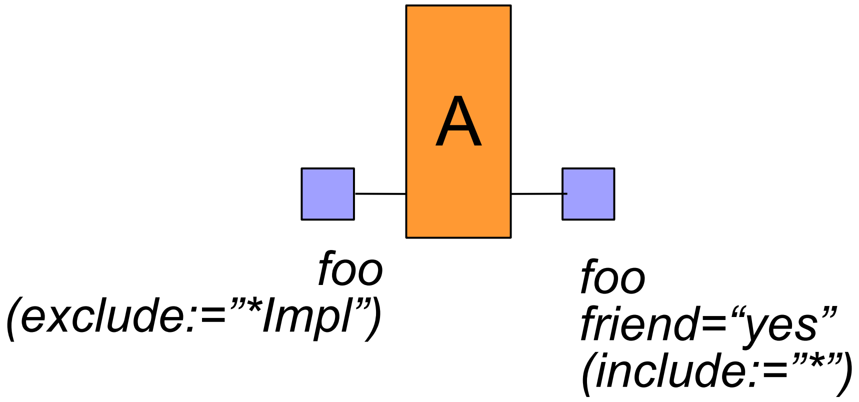
OSGi R4 Modularity Details (5/7)

- Package filtering for fine-grained class visibility
  - Exporters may declare that certain classes are included/excluded from the exported package

OSGi R4 Modularity Details (5/7)

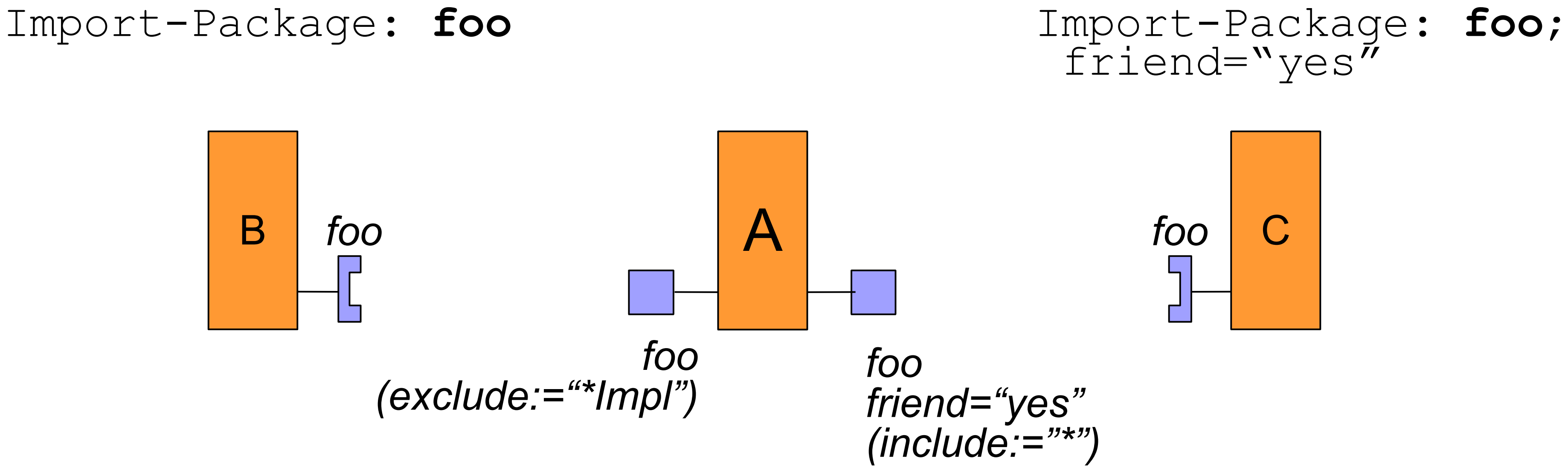
- Package filtering for fine-grained class visibility
  - Exporters may declare that certain classes are included/excluded from the exported package

```
Export-Package: foo;
 exclude:=“*Impl”,
 foo; friend=“yes”;
 mandatory:=“friend”
```



OSGi R4 Modularity Details (5/7)

- Package filtering for fine-grained class visibility
  - Exporters may declare that certain classes are included/excluded from the exported package



```
Import-Package: foo
```

```
Import-Package: foo;
 friend="yes"
```

B

foo

A

foo
(exclude:="*Impl")

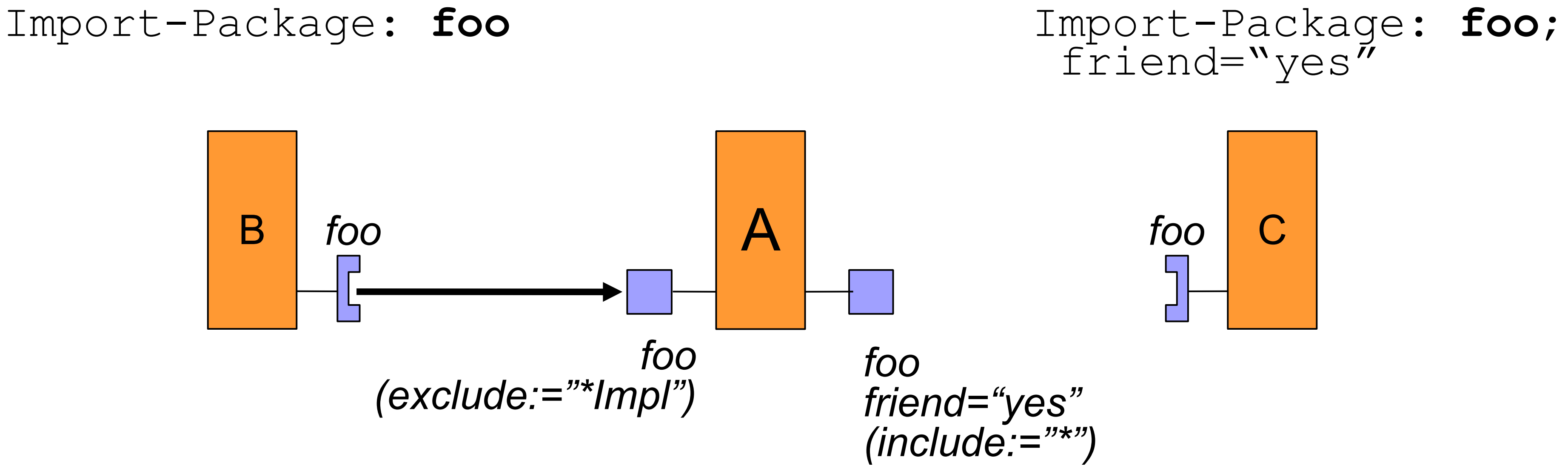
foo
friend="yes"
(include:="*")

foo

C

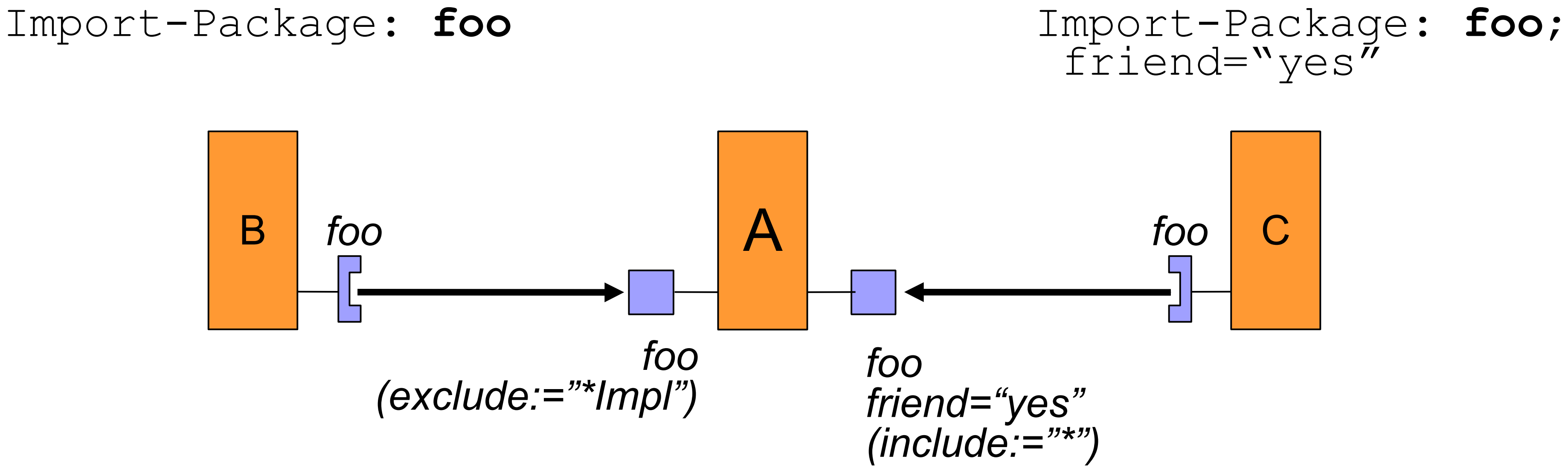
OSGi R4 Modularity Details (5/7)

- Package filtering for fine-grained class visibility
  - Exporters may declare that certain classes are included/excluded from the exported package



OSGi R4 Modularity Details (5/7)

- Package filtering for fine-grained class visibility
  - Exporters may declare that certain classes are included/excluded from the exported package



```
Import-Package: foo
```

```
Import-Package: foo;
 friend="yes"
```

OSGi R4 Modularity Details (6/7)

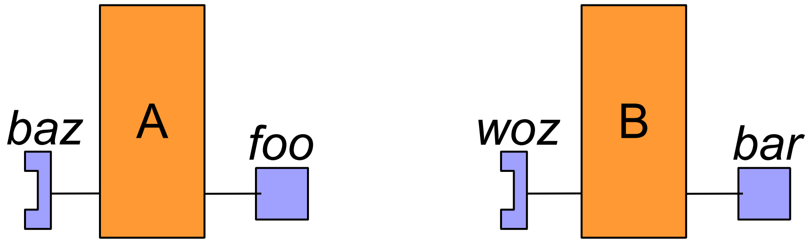
- Bundle fragments
  - Allows bundle content to be extended
  - A special bundle that attaches to a host bundle and uses the same class loader
    - Conceptually becomes part of the host bundle, allowing a logical bundle to be delivered in multiple physical bundles

OSGi R4 Modularity Details (6/7)

- Bundle fragments
  - Allows bundle content to be extended
  - 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

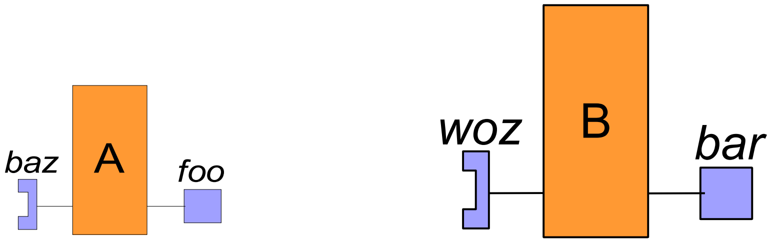
```
Fragment-Host: B
Export-Package: foo
Import-Package: baz
```

```
Bundle-SymbolicName: B
Export-Package: bar
Import-Package: woz
```



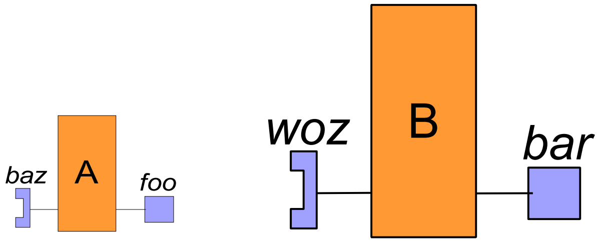
OSGi R4 Modularity Details (6/7)

- Bundle fragments
  - Allows bundle content to be extended
  - 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



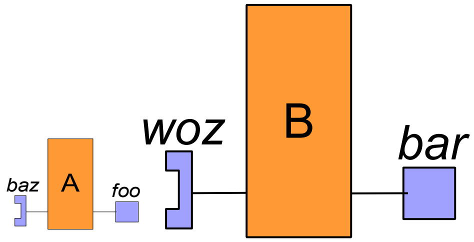
OSGi R4 Modularity Details (6/7)

- Bundle fragments
  - Allows bundle content to be extended
  - 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



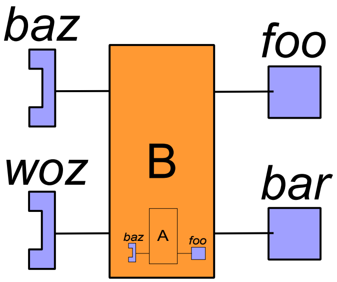
OSGi R4 Modularity Details (6/7)

- Bundle fragments
  - Allows bundle content to be extended
  - 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



OSGi R4 Modularity Details (6/7)

- Bundle fragments
  - Allows bundle content to be extended
  - 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



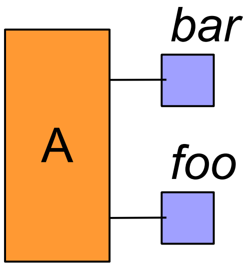
OSGi R4 Modularity Details (7/7)

- Bundle dependencies
  - Allows for tight coupling of bundles when required
  - Import everything that another, specific bundle exports
  - Allows re-exporting and split packages

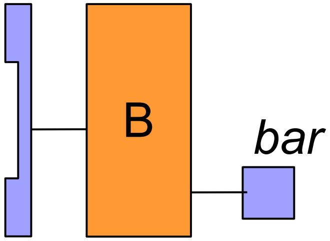
OSGi R4 Modularity Details (7/7)

- Bundle dependencies
  - Allows for tight coupling of bundles when required
  - Import everything that another, specific bundle exports
  - Allows re-exporting and split packages

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



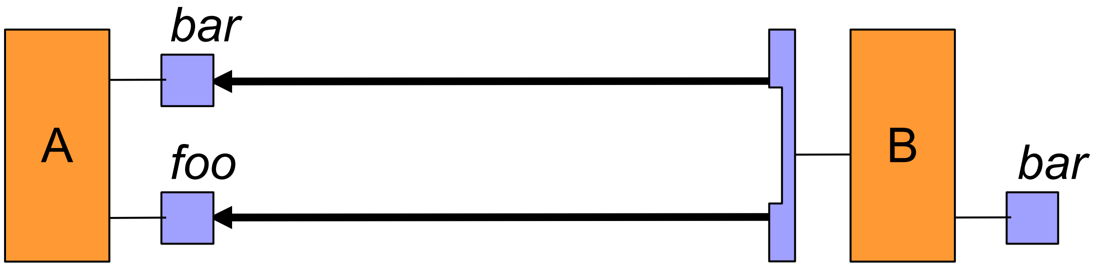
```
Require-Bundle: A
Export-Package: bar
```



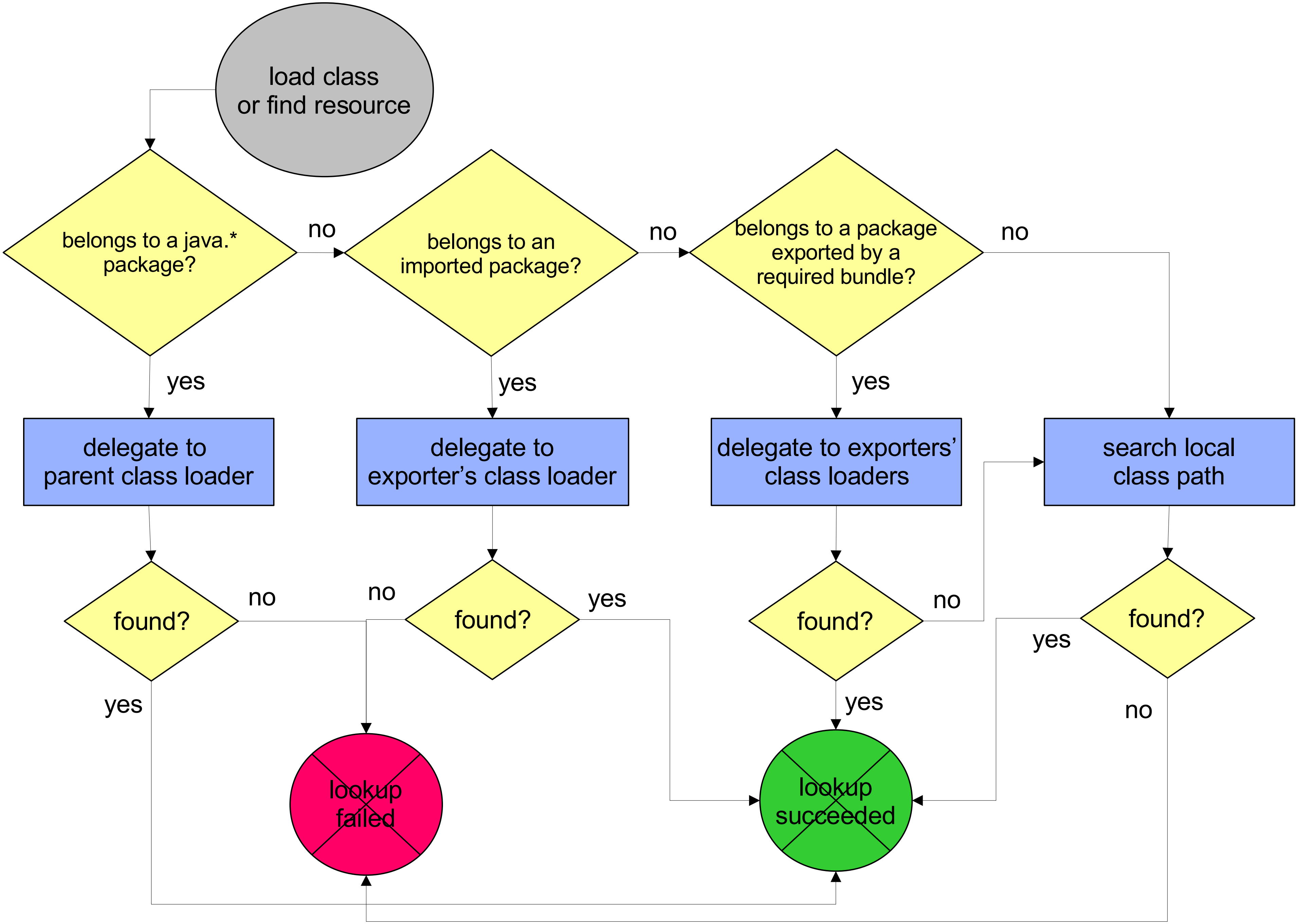
OSGi R4 Modularity Details (7/7)

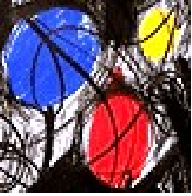
- Bundle dependencies
  - Allows for tight coupling of bundles when required
  - Import everything that another, specific bundle exports
  - Allows re-exporting and split packages

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



OSGi R4 Run-time Class Search Order





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:
 osgi.service.log; version=“[1.0.0,1.1.0)”;
   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-Activato              Activator
Bundle-ClassP               /embedded.jar
Bundle-NativeC
 libfoo.so; osname=Linux; processor=x86,
 foo.dll; osname=Windows 98; processor=x86
Import-Package:
 osgi.service.log; version="[1.0.0,1.1.0)";
   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 essor=x86,
 foo.dll; osname=Wi processor=x86
Import-Package:
 osgi.service.log; version="[1.0.0,1.1.0)";
   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;            x; processor=x86,
 foo.dll; os            s 98; processor=x86
Import-Package:
 osgi.service.log; version="[1.0.0,1.1.0)";
   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"
```

Life cycle entry point

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; osna  =Linux; processor=x86,
 foo.dll;            s 98; processor=x86
Import-Pa
 osgi.service.log; version=“[1.0.0,1.1.0)”;
   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 bundle 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:
 osgi.service.log; ersion="[1.0.0,1.1.0)";
   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"
```

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:                embedded.jar
Bundle-NativeCo
 libfoo.so; osn                     ssor=x86,
 foo.dll; osname=                   processor=x86
Import-Package:
 osgi.service.log; version="[1.0.0,1.1.0)";
   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"
```

Optional dependency on a package version range

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                rocessor=x86,
 foo.dll; osna                   rocessor=x86
Import-Package
 osgi.service.lo                 1.0.0,1.1.0)”;
   resolution:=“                tional
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

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:
 osgi.service.log; ve                 1.1.0)”;
   resolution:=“opti
Export-Package:
 org.foo.service; vers                    ,
   vendor=“org.foo”;  xclude:=“*Impl”,
 org.foo.service.bar; version=1.1;
   uses:=“org.foo.service”
```

Provided package with dependency on exported package

OSGi Modularity Best Practices

- Partition public and non-public classes into separate packages
  - Packages with public classes can be exported
  - Non-public classes are not exported
- Use package imports rather than bundle dependencies
  - Allows substitutability of package providers
- Limit fragment use
- Avoid use of dynamic imports
  - Special type of optional import that is resolved at run time, instead of resolve time
  - Intended for `Class.forName()` or SPI-like use cases

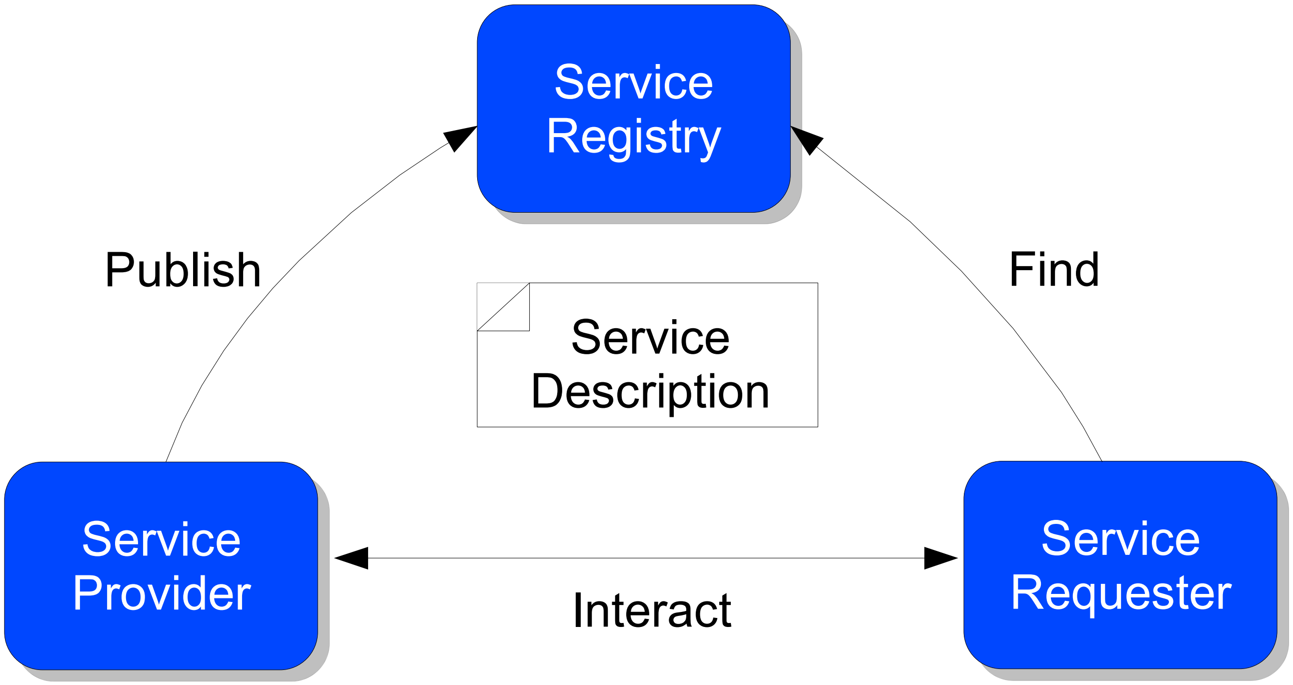
OSGi Modularity Tool Support

- Leveraging OSGi modularity
  - Text editor + **`jar`**
    - Just add metadata to your JAR file's manifest
  - Eclipse
    - Plug-in Development Environment directly supports bundles
  - Bundle packaging tools
    - BND from Peter Kriens
    - Apache Felix maven-bundle-plugin based on BND

OSGi as a Service-Oriented Application Framework

Service Orientation

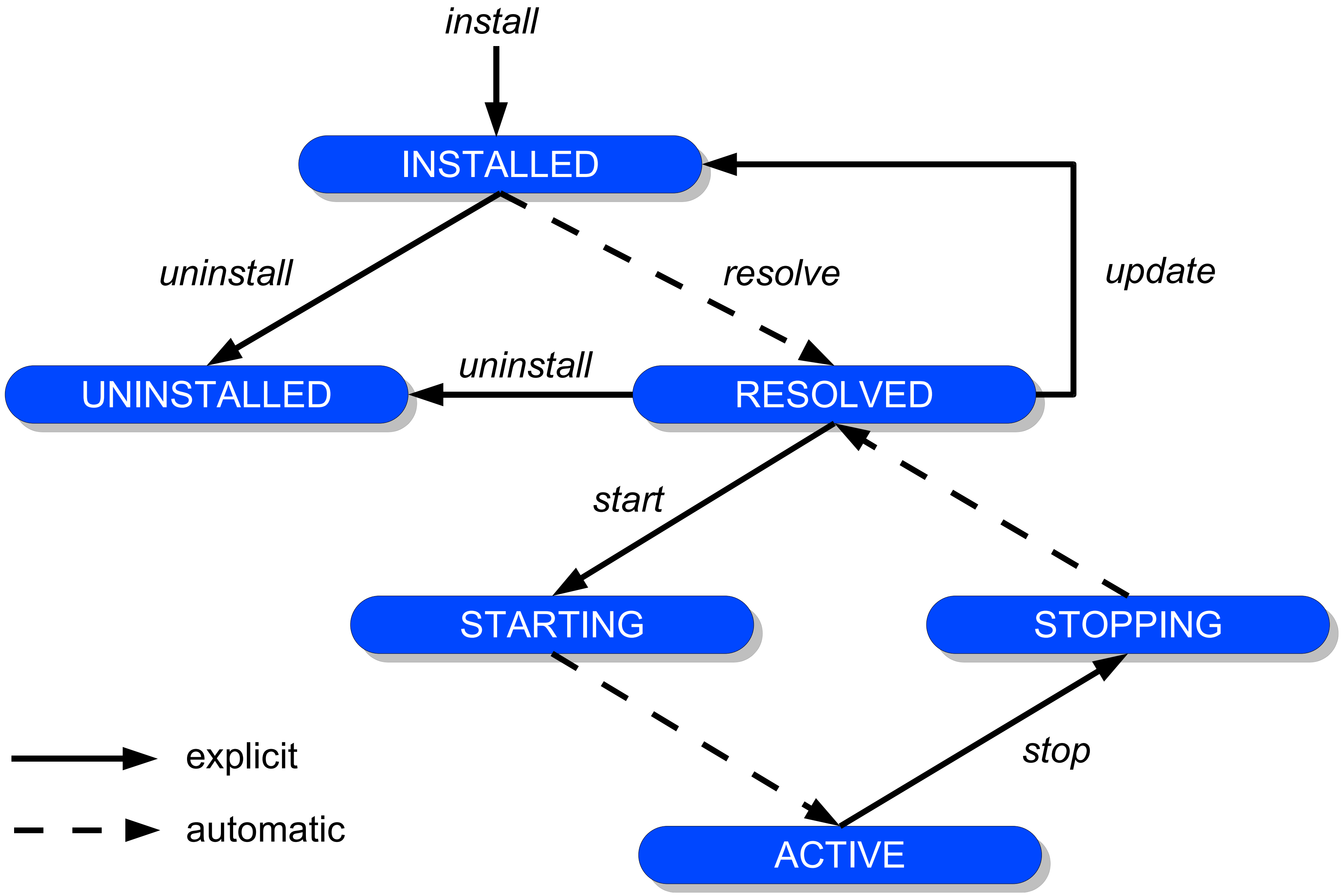
- The OSGi framework promotes a service-oriented interaction pattern among bundles



OSGi Applications

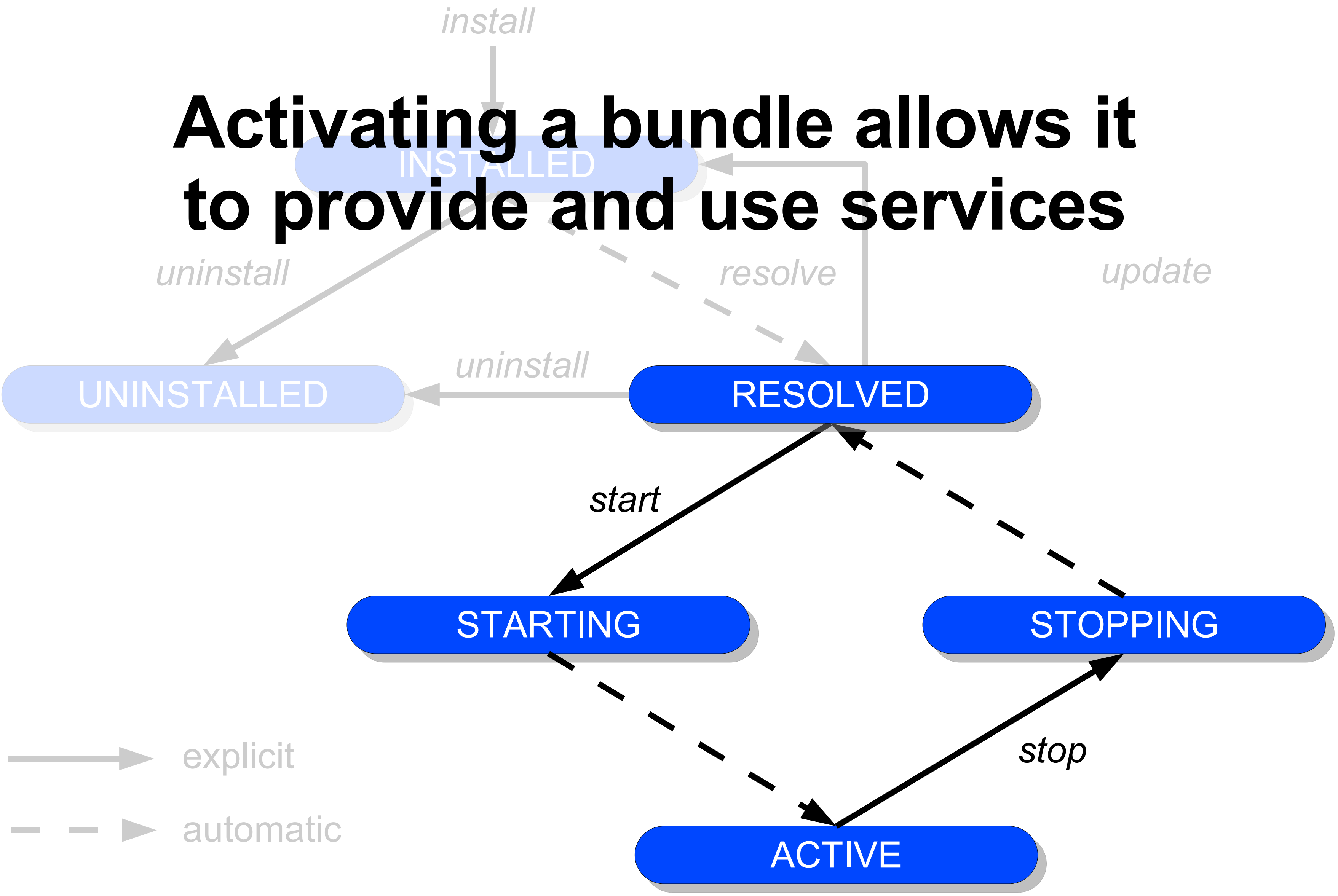
- A collection of bundles that interact via service interfaces
  - Bundles may be independently developed and deployed
  - Bundles and their associated services may appear or disappear at any time
- Resulting application follows a ***Service-Oriented Component Model*** approach
  - Combines ideas from both component and service orientation

Bundle Life Cycle (Revisited)



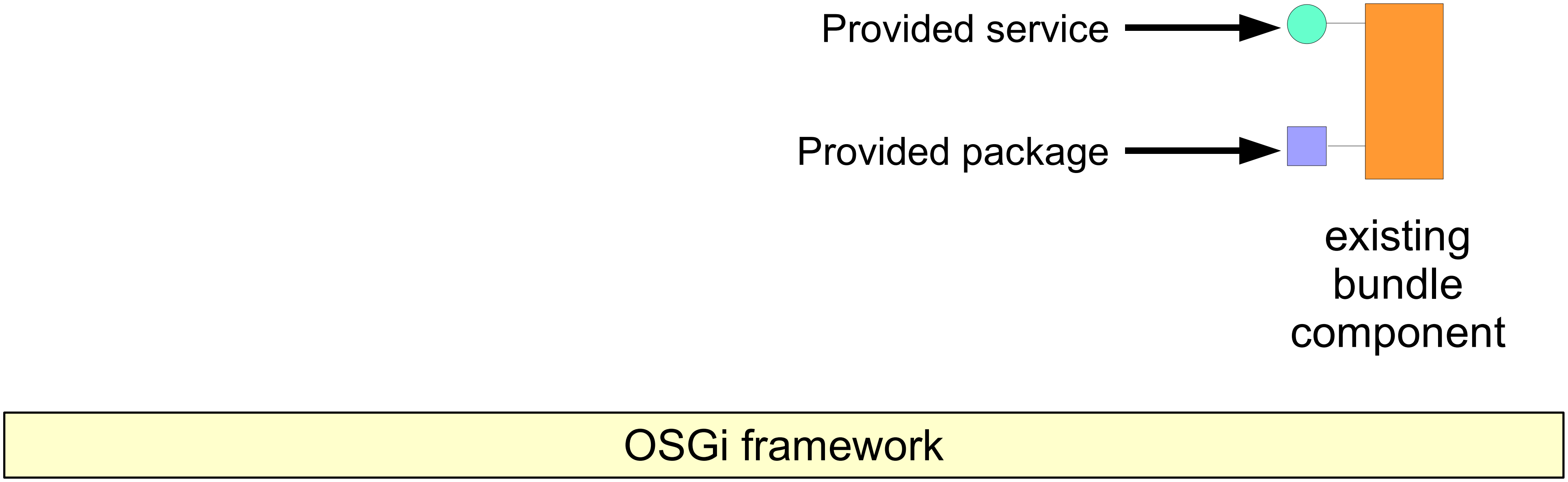
Bundle Life Cycle (Revisited)

Activating a bundle allows it to provide and use services



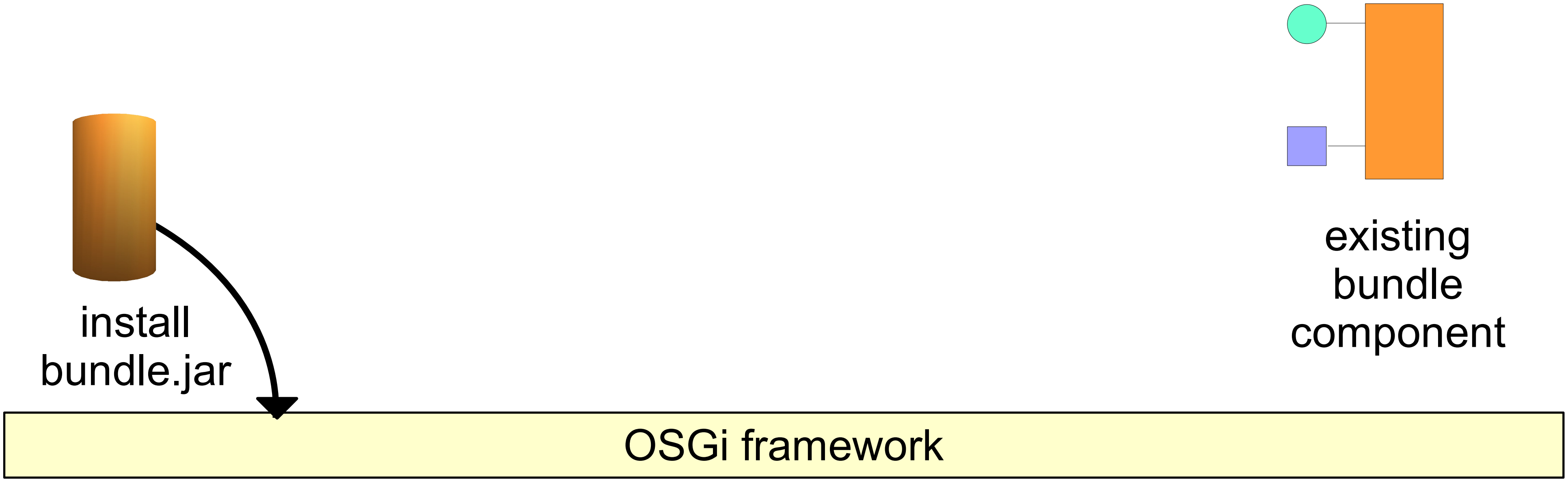
Service Provision Illustration

- Conceptually, a bundle contains a single component which is the bundle activator



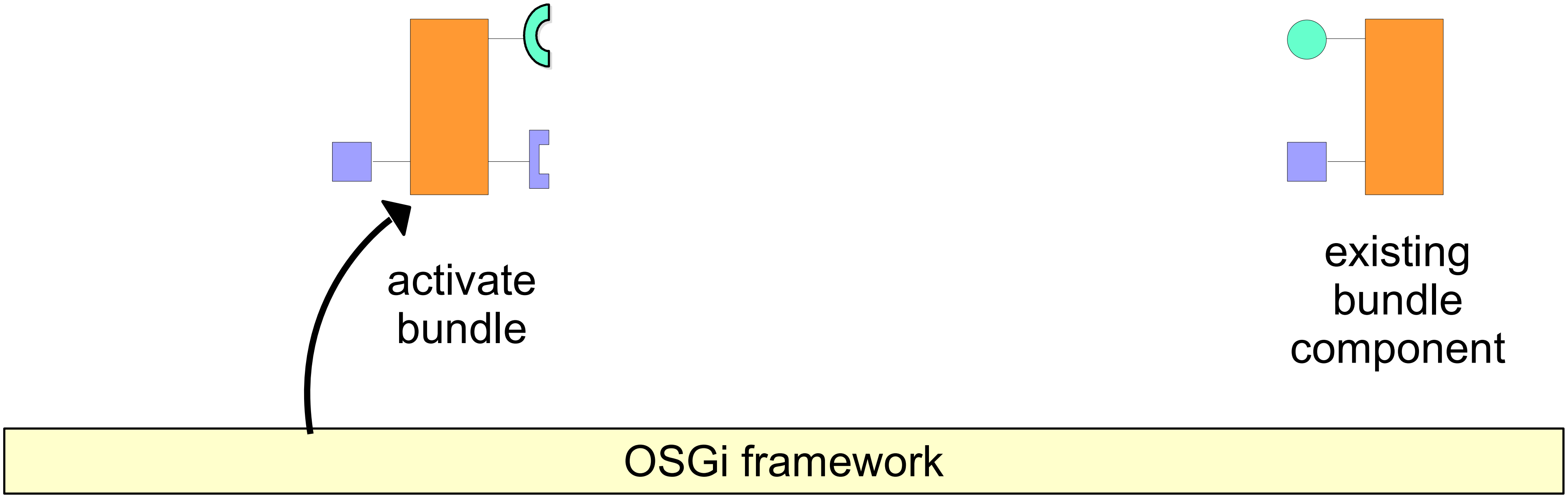
Service Provision Illustration

- Conceptually, a bundle contains a single component which is the bundle activator



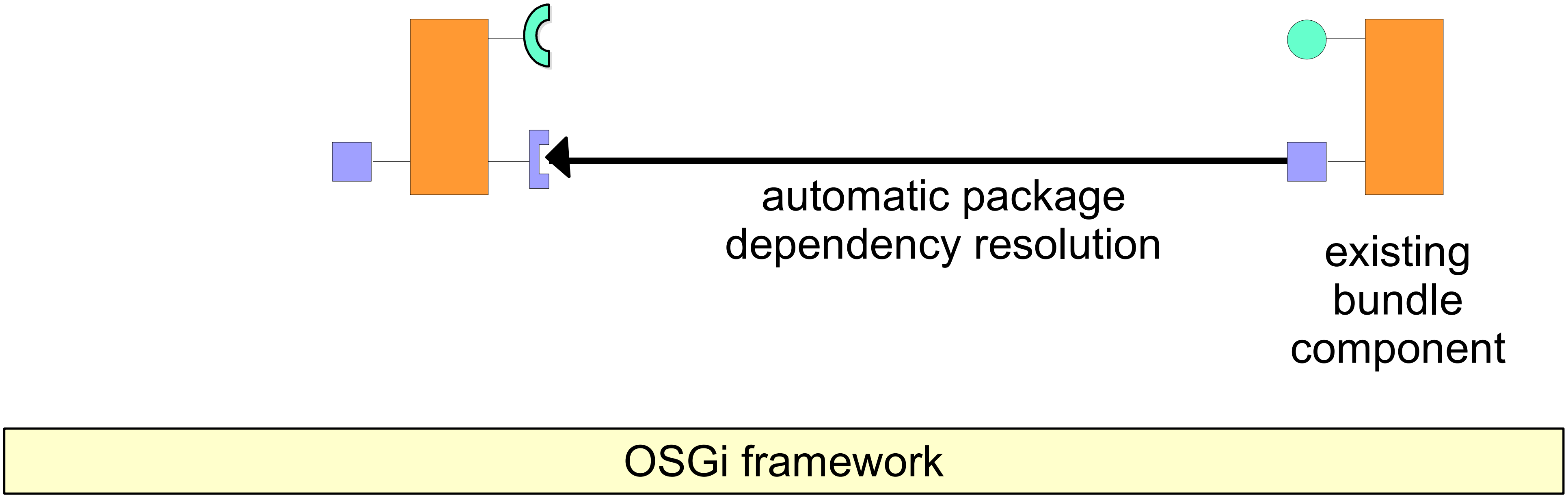
Service Provision Illustration

- Conceptually, a bundle contains a single component which is the bundle activator



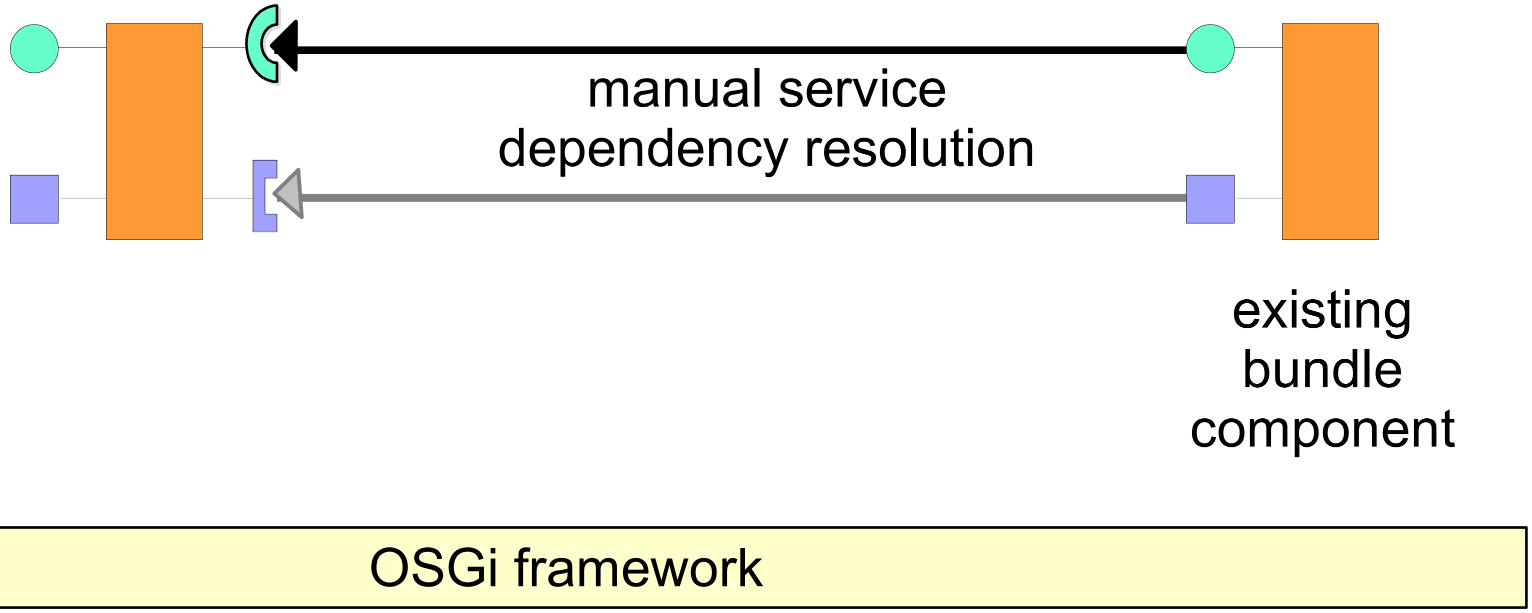
Service Provision Illustration

- Conceptually, a bundle contains a single component which is the bundle activator


automatic package dependency resolution
existing bundle component
OSGi framework

Service Provision Illustration

- Conceptually, a bundle contains a single component which is the bundle activator



Service-Oriented Application Advantages

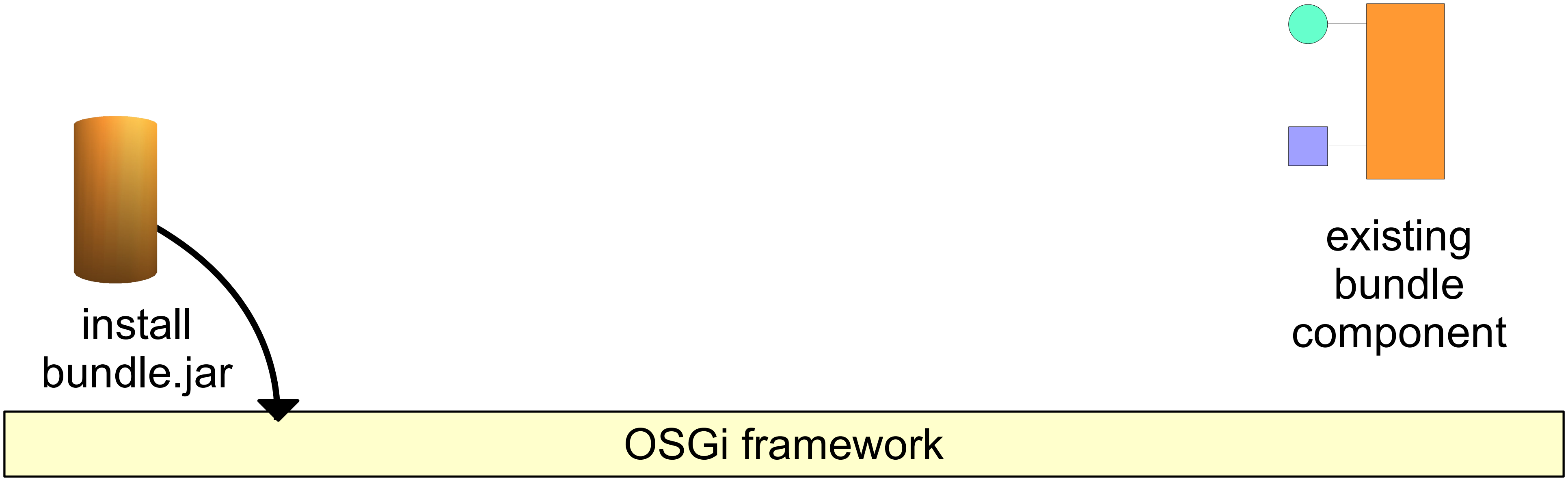
- Lightweight services
  - Direct method invocation
- Structured code
  - Promotes separation of interface from implementation
  - Enables reuse, substitutability, loose coupling, and late binding
- Dynamics
  - Loose coupling and late binding make it possible to support run-time management of modules
- Application's architectural configuration is defined by the set of deployed bundles
  - Just deploy the bundles that you need

Service-Oriented Application Issues

- Complicated
  - Requires a different way of thinking
    - Things you need might not be there or go away at any moment
  - Must manually resolve service dependencies
  - Must track and manage service dynamics
- There is help
  - Service Tracker
    - Still somewhat of a manual approach
    - Old-fashioned approach
  - Declarative Services (DS), Spring-OSGi, iPOJO
    - Sophisticated service-oriented component frameworks
    - Automated dependency injection
    - More modern, POJO-oriented approaches

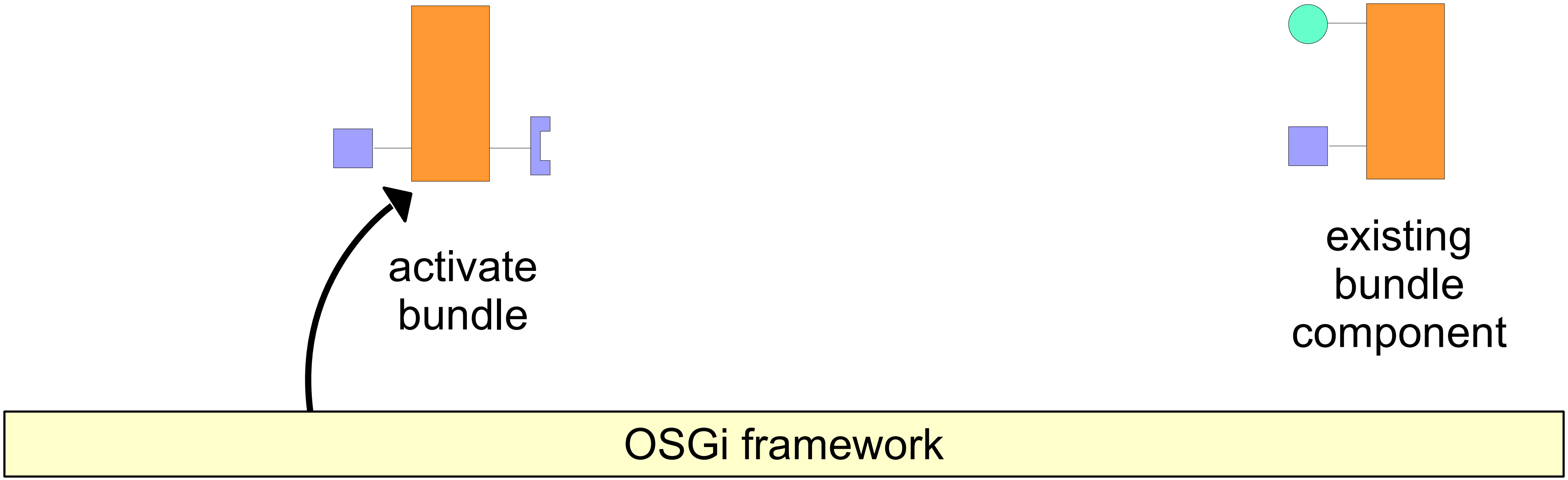
Service-Oriented Application Illustration

- Bundles are deployment units for component types that can be automatically instantiated, resolved, and managed



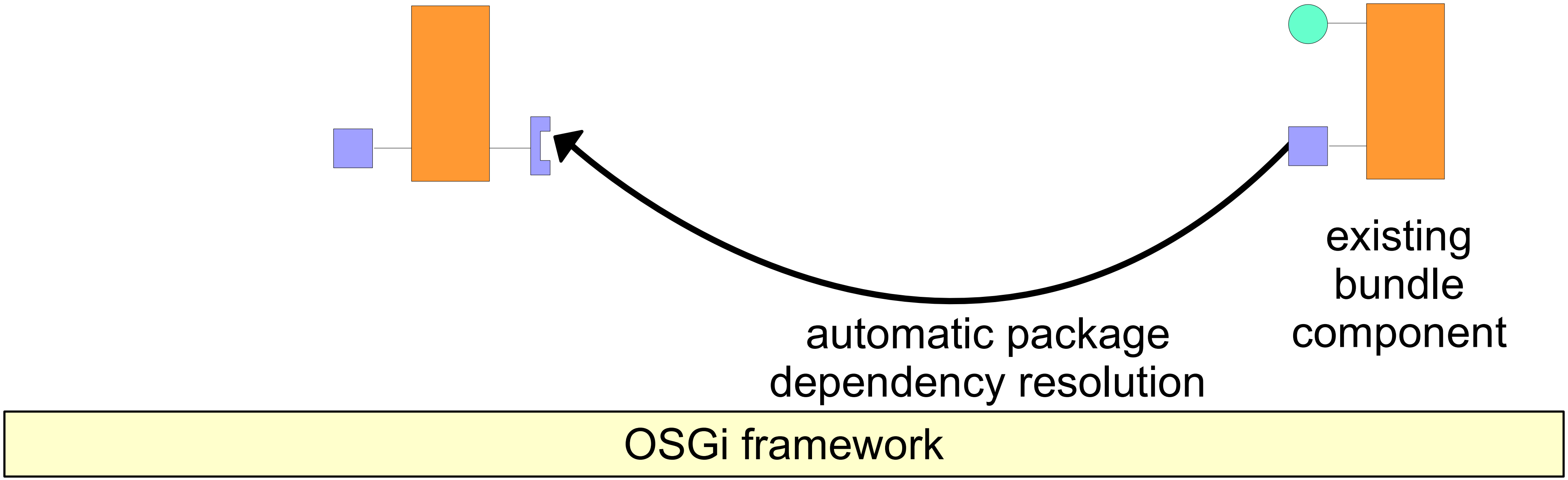
Service-Oriented Application Illustration

- Bundles are deployment units for component types that can be automatically instantiated, resolved, and managed



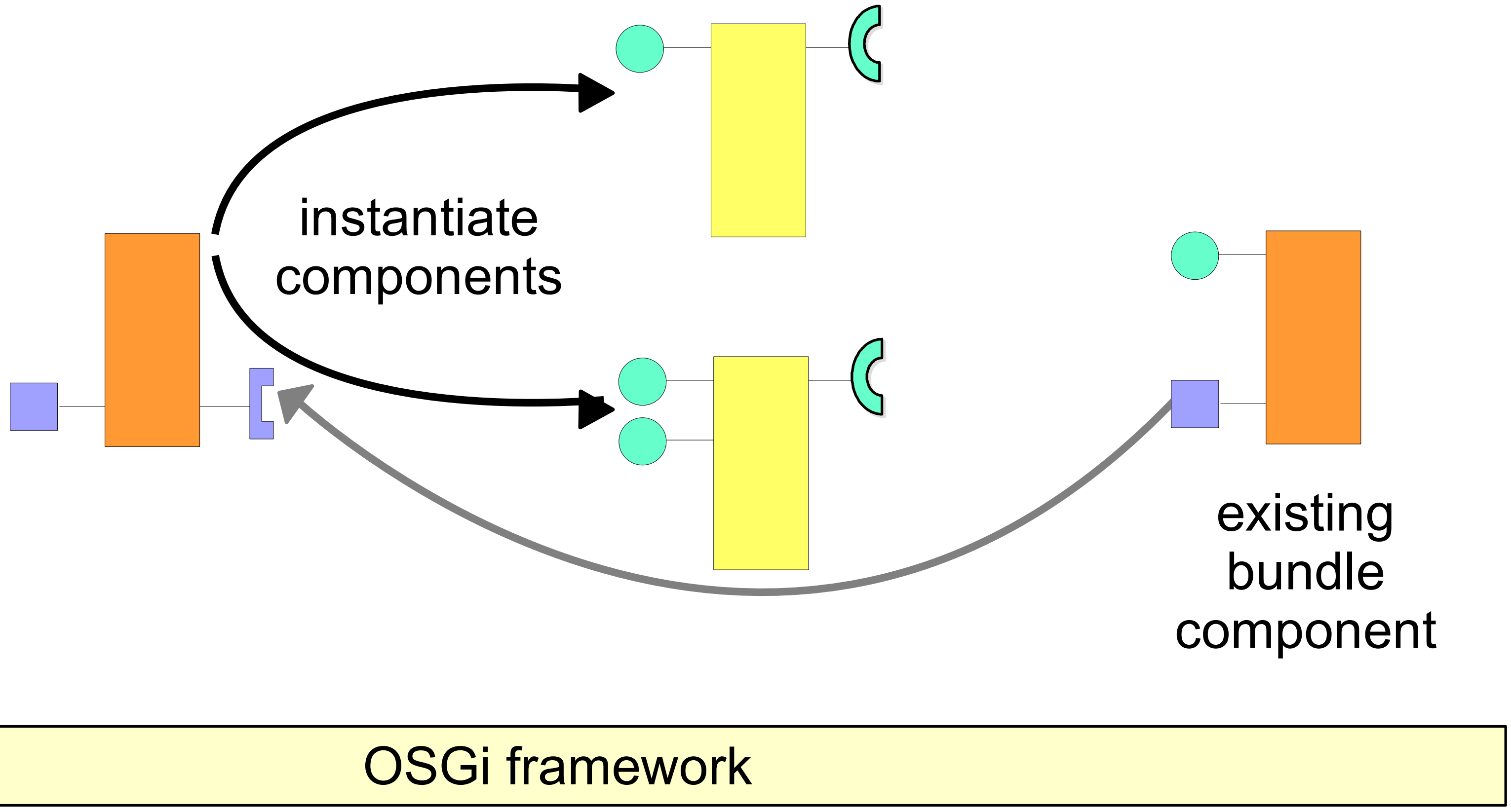
Service-Oriented Application Illustration

- Bundles are deployment units for component types that can be automatically instantiated, resolved, and managed



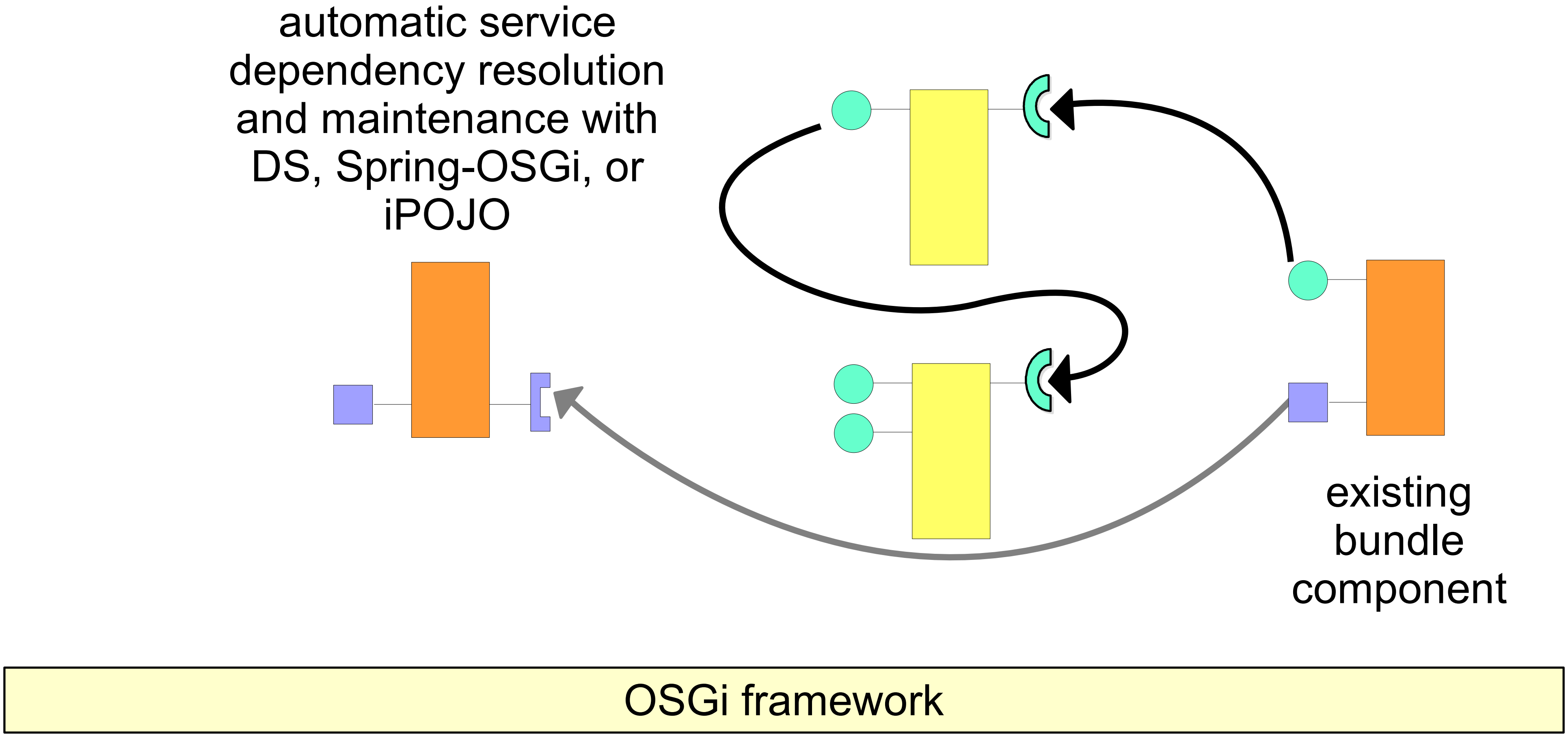
Service-Oriented Application Illustration

- Bundles are deployment units for component types that can be automatically instantiated, resolved, and managed



Service-Oriented Application Illustration

- Bundles are deployment units for component types that can be automatically instantiated, resolved, and managed


automatic service dependency resolution and maintenance with DS, Spring-OSGi, or iPOJO
existing bundle component
OSGi framework

Declarative Services Example (1/2)

- Declarative Services provides a minimally intrusive way to
  - Define components that provide and use services
  - Automate dependency resolution and maintenance

```
package foo.impl;
public class HelloImpl implements foo.HelloService {
    LogService log;
    protected void setLog(LogService l) {
        log = l;
    }
    protected void unsetLog(LogService l) {
        log = null;
    }
    public void sayHello(String s) {
        log.log(LogService.LOG_INFO, “Hello ” + s);
    }
}
```

Declarative Services Example (2/2)

- Declarative Services component metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="example.hello">
   <implementation class="foo.impl.HelloImpl"/>
   <service>
      <provide interface="foo.HelloService"/>
   </service>
   <reference name="LOG"
    interface="org.osgi.service.log.LogService"
    bind="setLog"
    unbind="unsetLog"
   />
</component>
```

iPOJO Example (1/2)

- iPOJO provides an extensible POJO-based way to
  - Define components that provide and use services
  - Automate dependency resolution and maintenance
  - Define composite components with sub-service visibility scoping

```
package foo.impl;
public class HelloImpl implements foo.HelloService {
    LogService log;
    public void sayHello(String s) {
        log.log(LogService.LOG_INFO, “Hello ” + s);
    }
}
```

iPOJO Example (2/2)

- iPOJO component metadata

```xml
<?xml version="1.0" encoding="UTF-8"?>
<component className="foo.impl.HelloImpl">
   <provides/>
   <dependency field="log"/>
</component>
<instance component="foo.impl.HelloImpl"
 name="example.hello"/>
```

OSGi Application Development Approach

- Modules vs. “Modules + Services”
  - It is possible to use only the modularity aspects of the OSGi framework and not use services as a way of structuring your application
  - May be necessary if another component model is already in use or application interaction is structured differently
- “On Top” vs. Embedded
  - An application can be a set of collaborating bundles that can be deployed on any framework or an application can embed an instance of the framework to create an extensibility/plugin mechanism, which will often tie the application to a specific framework implementation

Apache Felix Overview

Apache Felix (1/4)

- Currently in the Apache Incubator
  - Graduation to top-level project anticipated this month
- Apache licensed open source implementation of OSGi R4
  - Framework (in progress, stable and functional)
    - Version 0.8.0 currently available
  - Services (in progress, stable and functional)
    - Package Admin, Start Level, URL Handlers, Declarative Services, UPnP Device, HTTP Service, Configuration Admin, Preferences, User Admin, Wire Admin, Event Admin, Meta Type, and Log
    - OSGi Bundle Repository (OBR), Dependency Manager, Service Binder, Shell (TUI and GUI), **iPOJO**, Mangen

Apache Felix (2/4)

- Felix community is growing strong
  - 20 committers
  - Code granted and contributed from several organizations and communities
    - Grenoble University, ObjectWeb, CNR-ISTI, Ascert, Luminis, Apache Directory, INSA, DIT UPM, Day Management AG
    - Several community member contributions
  - Apache projects interested in Felix and/or OSGi
    - Directory, Cocoon, JAMES, Jackrabbit, Harmony, Derby

Apache Felix (3/4)

- Felix bundle developer support
  - Apache Maven2 bundle plugin
    - Merges OSGi bundle manifest with Maven2 POM file
    - Automatically generates metadata, such as Bundle-ClassPath, Import-Package, and Export-Package
      - Greatly simplifies bundle development by eliminating error-prone manual header creation process
    - Automatically creates final bundle JAR file
      - Also supports embed required packages, instead of importing them
- Felix Commons
  - Effort to bundle-ize common open source libraries
    - Recently started
  - Includes 13 bundles, such as antlr, cglib, commons-collections, etc.
  - All community donated wrappers

Apache Felix (4/4)

- Roadmap
  - Incubator graduation hopefully this month
  - Version 1.0.0 release shortly after graduation
    - To include major portions of R4 specification functionality
      - Largely only missing support for fragments
    - Also focusing on security aspects

Conclusions

Conclusions

- Java needs improved modularity support
  - The OSGi R4 framework provides it now
- Importance and relevance is growing
  - Industry support in mobile and enterprise scenarios
- Several related JCP JSRs
  - JSR-291 introduces the OSGi framework into JCP
    - Will result in OSGi R4.1
  - JSR-294 to introduce VM modularity support in Java 7
    - Super packages and separate compilation
  - JSR-277 to introduce somewhat overlapping JAR file-based modularity in Java 7
    - Overlaps in packaging and deployment
    - Differs in dynamics/life cycle, support for existing JREs

Questions?