RFC 79 - Enhanced Modularity Support

Final

57 Pages

Abstract

The OSGi framework is extended to support a stronger form of modularity and a richer and more flexible means of matching modules to the modules they need to use.

The resultant framework maintains compatibility for current OSGi R3 customers and satisfies the class loading requirements of Eclipse as expressed by RFC 70.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.` ***Outstanding issues and questions are shown in this typeface.***

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
GN	2004-04-13	Initial draft. Glyn Normington, IBM, glyn_normington@uk.ibm.com.
GN	2004-04-14	Comments from BJ Hargrave and Peter Kriens. Improve terminology.
GN	2004-04-20	Modularity layer requirements. Start defining the application domain. Improvements and comments from Peter Kriens.
GN	2004-04-22	Requirement on modularizing existing code.
GN	2004-04-28	Princeton CPEG meeting feedback.
GN	2004-06-11	Add initial proposal to address this RFC's modularity requirements while supporting the requirements of RFC 70. Include some remaining issues from BJ as comments.
GN	2004-07-01	Recast proposal in terms of a modularity layer which covers the continuum of requirements in terms of provider selection, package grouping, package multiplicity, and version constraints.
GN	2004-07-06	Suggestions from Peter Kriens, including package variants, export version ranges, and import filters.
GN	2004-07-14	Comments and edits from BJ Hargrave and Ben Reed.
GN	2004-07-21	Security section plus comments from Tom Watson.

Revision	Date	Comments
GN	2004-07-22	Move partial package export requirement to considered alternatives. De-emphasize layering (mainly implementation issue). Rework re-exports in terms of public imports. Re-order material prior to Chicago CPEG.
GN	2004-07-29	Chicago CPEG meeting feedback. Split packages, partitioned service registry, fragments, and an example of explicit grouping are still outstanding.
GN	2004-08-11	Improve propagation syntax. Name the default grouping. Describe some weaknesses.
GN	2004-09-02	Permission handling complication reported by Ben Reed.
GN	2004-09-10	Support split packages and partial package export. Differentiate between directives and matching attributes and allow matching attributes to be mandatory. Drop support for RFC 70 syntax as this is only applicable to Eclipse and need not be in the R4 standard. Express dependency on change to RFC 70 fragments. Add examples.
GN	2004-10-05	Clear change history. Amsterdam CPEG meeting feedback. Bundle symbolic name mandatory in v2 manifest and may not be specified on export. Clarify semantics when a package is imported or exported more than once. Remove propagated packages from a module's export signature and add Reexport-Package. Add split directive for split packages. Re-introduce DynamicImport-Package. Delete wild carded package names from (static) imports.
GN	2004-10-07	Support system classes and the system bundle. Revise security section. Clarify dynamic import behaviour. Mention multiple grouping constraints.
GN	2004-10-13	Define behaviour of re-export of an optional import. Support bundle version ranges in imports. Move "security considerations" to before "considered alternatives". Note some outstanding questions in bold, italic font.
GN	2004-10-15	Rename propagate to grouping on Import-Package. Add Require-Bundle from RFC 70. Remove split directive from Import-Package.
GN	2004-10-22	Clarify import and export of the same package. Delete definitions of bundle symbolic name and bundle version that were copied from RFC 70 now that we do not intend to rename them. Delete discussion of split packages from Import/Export-Package and make this a feature only of Require-Bundle. Move proposed renamings to considered alternatives. Change reexport directive of Require-Bundle to "visibility". Make the implementation of Require-Bundle optional for frameworks.
GN	2004-10-29	Describe the rationale for Require-Bundle and its optionality. Fix some typo's.
GN	2004-11-04	Make Reexport-Package optional. Document considered alternative for pure bundle dependencies. Process consequent resolutions consistently with initial target resolutions. Simplify export filters. Describe class loading search order.

Revision	Date	Comments
GN	2004-11-15	Add material deleted from RFC 70.
GN	2004-11-18	Improvements from BJ Hargrave including clarified version syntax and making Import-Package take priority over Require-Bundle.
RSH	2004-11-22	Simplify security support.
RSH	2004-12-17	Change system class handling following Boca Raton CPEG meeting.
GN	2004-12-21	Bundle manifest version clarification. Restrict delimiter before version qualifier to be period rather than period or hyphen. Restrict all attributes (including directives) to being specified at most once each per list of parameters. Clarify import and dynamic import semantics. Note that the R4 spec. should use “bundle” instead of “module”.
TJW	2005-01-17	Update the multiple version export considerations section to include more details about how the service registry and service events are filtered according to what version of a package a bundle is wired to.
GN	2005-01-21	Clarify behaviour when a package is export multiple times by a required bundle.
GN	2005-02-23	Use directive for bundle singleton. Replace groupings with dependencies. Delete overly prescriptive description of how to resolve modules.
TJW	2005-02-23	Modify ServiceListener.isAssignableTo checks.
RSH	2005-02-24	Changes to system class handling, especially for R3 compatibility.
TJW	2005-02-25	Tighten up service registry and service events sections.
GN	2005-03-02	Remove ‘uses’ directive from Reexport-Package
RSH/GN	2005-03-09	List manifest install errors
GN	2005-03-10	Minor editorial improvements.
GN	2005-03-30	Change to establishment of wires for DynamicImport-Package as proposed by BJ Hargrave. Limit include/exclude filters to classes and extend wildcarding.
RSH	2005-03-31	Allow for JDK bugs in delegation to parent class loader.
TJW	2005-03-31	Clarify behavior of boot delegation packages to not terminate search.
GN	2005-04-08	Make specification-version a deprecated synonym for version in R4 manifests.
BJH	2005-05-27	No Changes. Final.

1 Introduction

This RFC extends the modularity features of the OSGi framework so that the framework may be applied more broadly including in embedded devices as described by RFP 46 (“OSGi on Embedded Systems”) [5].

Modularity is the capability to construct programs from ‘modules’. A module is an encapsulated unit of code and data that relates to other modules through well-defined and strictly-controlled interfaces. A well-designed module presents an abstract view of itself through its interface and hides its internal data so that its users are not sensitive to the internal design of the module.

Java classes provide a simple kind of module, but Java does not provide a coarser-grained module construct. Java packages enable a limited form of encapsulation and data hiding through package private methods and data members and reduce name clashes by providing a structured name space.

OSGi R3, abbreviated to ‘R3’ from now on, provides modularity in the form of services which communicate via well-defined and strictly-controlled service interfaces. R3 also supports versioning of service interfaces. Versioning is also a requirement for modules, but there is no support in Java to manage versions of classes and packages other than by writing custom class loaders, which is a specialized and error-prone activity.

This RFC provides a module[1] construct which builds on the R3 service import/export mechanism. The resultant modules provide many of the benefits of custom class loaders without requiring the module programmer to code class loaders. Modules express their interface (for use by other modules) by exporting packages. A module may use another module by importing some of its exported packages.

The “modjava” project in IBM Research has been a major point of reference and comparison in the development of this RFC. The results of the project were published in [3].

2 Application Domain

OSGi provides a runtime environment in which Java components known as *bundles* interact. Bundles carry code, resources, and manifest definitions known as *metadata* in JAR files. Some bundles provide OSGi defined standard components; other bundles are application programs. OSGi has a *framework* (an extensible collection of classes and interfaces) which manages bundles and provides the means for bundles to interact with each other through *service* interfaces as well as package sharing. An instance of the runtime environment provided by OSGi is known as a *service platform* and includes an instance of the framework, running inside a single JVM, together with various OSGi standard bundles.

[1] ***When the R4 spec. is written, it would probably be advisable to standardize on the term “bundle”.***

The framework provides a rich set of capabilities. It supports *modularity* by giving each bundle its own namespace of classes (or *class space*) and controlling how bundles share classes according to bundle metadata. It manages the life cycle of bundles as they are installed, started, stopped, updated, and uninstalled. It also manages the publication, or *registration*, of service interfaces in a service registry and determines how bundles find, and bind to, services.

3 Problem Description

The standard Java platform provides limited support for packaging, deploying, and validating Java-based applications and components. The .NET platform explicitly addresses these issues by introducing *assemblies* and the Global Assembly Cache (GAC). Assemblies define how applications and application components are packaged and deployed, while the GAC manages the dependencies among assemblies to ensure that they are validly resolved. In Java, the closest analogy to an assembly is a JAR file, but there is no analogy to the GAC. As a result of this, many Java-based projects, such as jBoss and NetBeans, have resorted to creating custom module-oriented layers with specialized class loaders for packaging, deploying, and validating applications and components.

As an alternative, the R3 framework provides capabilities that are useful in a generic Java module framework. First, it defines a unit of modularization (i.e. a bundle), where the modules export and import packages among each other. For Java developers, this approach is intuitive because normal Java development already employs an importation concept for managing package namespaces. Second, it defines a simple module life cycle. Lastly, it defines a framework to manage the module life cycle to ensure validity. It is clear from these capabilities that the R3 framework provides a significant improvement over standard Java. However, the R3 framework does have some serious limitations as a generic module framework. Specifically, the R3 framework is limited to the following characteristics:

- Only one version of a shared package can exist in the JVM among modules at any given time. While it is possible to load multiple versions of a specific package, this can only be accomplished by using private packages that are embedded in a module and which cannot be shared with other modules.

- Importing modules do not have any influence over which provider is used to resolve their dependency, other than by specifying a package version number.

- The version compatibility policy when declaring an import is hard-coded to “greater than or equal to” a given version number, which is impractical for many real-world project situations.

- Exporting modules have no way to indicate that they are providing a related set of packages that must be used in an all-or-nothing fashion. This may be necessary if there are implementation dependencies among packages. Consequently, an importer might get the packages from multiple providers, which would break the implementation dependencies.

Many of these characteristics are not truly limitations of the OSGi framework; rather, they result from the fact that the framework was not designed to be a generic module framework for Java. Understanding these limitations does, however, provide insight into the degrees of freedom necessary to create a generic module framework for Java, which we shall explore further in the next section.

4 Requirements

R1. Many applications require support for modularity, but need higher level functions which differ from those of the OSGi framework. There is a need to improve the modularity support in the OSGi framework so that it can satisfy more general modularity requirements.

R2. It must be possible to write robust applications with good performance. The framework can assist by enabling static analysis of modules (described in section 4.1) and static optimizations based on the module definition and contents. It should be possible to reason about how a module will behave in the presence of other modules, including what will happen when two modules contain classes with the same name. See the introduction of [3] for background.

R3. R3 bundles must be supported without requiring metadata changes.

R4. RFC 70 class loading requirements must be satisfied with as little impact to RFC 70 metadata as is practical.

R5. Class sharing must support intermediate points between the extremes of R3's interface sharing and RFC 70's sharing of implementations between predetermined sets of modules. This requirement is unpacked in section 4.2.

4.1 Static Analysis Requirements

The framework must support static analysis of modules. This includes anything the developer of a module can infer about the behavior of her code based on knowledge of the imports and exports of the module. A simple case is inferring that some code is unused and deleting it. Another case is reworking the internals of a module to improve performance or other "non-functional" aspects while preserving compatible external behavior. Static analysis can be performed manually or with the help of tools, such as Eclipse, which analyze code structure and assist in refactoring.

Static analysis is typically performed during coding and results in changes to the code, or confirmation of the correctness of assumptions. The results of static analysis are not usually directly represented in the metadata or code of a module except perhaps as assert statements which check assumptions at runtime.

Static analysis may be useful to the maintainers of a module as well as developers who use the module and want to have some confidence that the module will behave correctly in the presence of other modules.

4.2 Class Sharing Requirements

Sharing implementation classes differs from sharing service interfaces in that it does not assume a single global name space of implementation packages for a service platform nor does it assume backward compatibility of later versions of an implementation package. The former enables distinct versions of the same package to be loaded simultaneously in the same service platform, which in R3 was possible only for packages private to bundles. There is however a continuum between the requirements for sharing service interfaces and the requirements expressed in RFC 70.

This continuum of class sharing requirements exhibits four inter-linked degrees of freedom: version constraints, provider selection, package dependencies, and package multiplicity. Version constraints and provider selection are related to importing packages from other modules, while package dependencies and package multiplicity are related to exporting packages to other modules. Flexible provider selection introduces a further requirement to propagate packages.

4.2.1 Version Constraints

Version constraints are a mechanism by which modules importing packages can declare precise or arbitrary version ranges that are acceptable for resolving their package dependencies.

4.2.2 Provider Selection

Provider selection defines a spectrum where an importer can accept anonymous providers, provide hints to direct provider selection, or explicitly declare a provider to resolve its package dependencies.

Interface sharing is anonymous in that the receiver does not care who provides the code as long as it meets the specification contract implied by the package name and version.

RFC 70's implementation sharing mechanism shares code by bundle symbolic name and bundle version. There is typically no specification contract between the receiver and the provider. The receiver ties itself to a specific bundle, typically the bundle that was used to test the receiver. The receiver may specify a range of bundle versions if it will function correctly with more than one version of the providing bundle.

Then there are some useful intermediate cases. For example, tightly coupled sharing by bundle symbolic name and bundle version can be too brittle. If there is a need to rework the bundle structure of the provider, e.g. to split a single bundle into multiple bundles, then the receiver is impacted. In such cases, the importer needs to limit the choice of exporter sufficiently for its purposes without determining the provider's bundle structure.

On the other hand, if brittleness is not an issue, an importer can go even further if necessary and limit the choice of exporter not only to a particular bundle symbolic name and bundle version but also, for example, a particular developer. This could prove useful when importing open source code which has been branched and taken in different directions by different developers.

An exporter may need to describe a package more precisely than by package name and package version so that importers can select the correct variant. For example, the contents of packages in javax.microedition.io are not uniquely defined by package name and package version: it is also necessary to know which J2ME profile is supported.

4.2.3 Package Dependencies

A bundle may need to express dependencies between packages such that they are used by a particular importing module in an all-or-nothing fashion. For example, if two Xerces modules are installed, an importer really needs to get all the Xerces packages from one module, not some from each.

Dependencies may occur when a package exported by a bundle uses another package either exported or imported by the bundle. This usage may be externally observable if it occurs in a package signature or it may be hidden and due to an implementation dependency.

4.2.4 Package Multiplicity

Package multiplicity defines whether multiple versions of a package may exist in the service platform at a given time.

There does not seem to be a real need to support singleton packages given that we can stipulate that resolution should prefer resolved modules to (as yet) unresolved modules.

4.2.5 Public Imports

If an exported package P uses classes from an imported package Q in its signatures, then a user of package P must also use the same package Q as P does to avoid class cast exceptions.

For example, suppose module *A* imports package *Q* from module *B* but then exports package *P* which exposes a class Qc from package Q as a parameter type, thus:

```
package P;
class Pc
{
    public void Pm(Q.Qc x);
}
```

Another module *C* importing *P* from *A* must import *Q* from *B* to avoid causing class cast exceptions when using the class *Q.Qc*.

So the requirement is for the module exporting P to be able to influence modules which import P from it to use the same package Q. Essentially, the dependency between P and Q needs to be expressed.

4.2.6 Re-exported Imports

Sometimes a module needs to make some of its imported packages available on its 'export signature' so that these packages are available to its importers, e.g. to those importers which specify the re-exporting module's bundle symbolic name.

This is similar to the function of' "require bundle" with 'reprovide="true"' which was introduced by RFC 70.

4.2.7 Export Filtering

To non-intrusively modularize existing code, it should not be necessary to reorganize the contents of packages. This is problematic when an existing package has some contents which need to be exported from the module and some which should be kept private to the module. This implies a requirement for a module to filter an exported package to avoid exporting all the contents of the package.

4.3 Module Composition Requirements

Sometimes it is necessary for a module to express a dependency on another module without tying down the precise set of packages exported by the other module. This is the case when the name of the module expresses a contract to provide particular functionality. For instance, it should be possible to implement a subsystem such as Eclipse's Standard Widget Toolkit (SWT) and provide its function to users as a single 'SWT' module.

It is even more necessary to avoid tying down the exported packages when a module is adds function to an existing module which can continue to evolve.

Conceptually, this can be thought of as a requirement to compose modules from other modules. Sometimes it is necessary to express a dependency on a module (and also import all its exported packages). Sometimes it is necessary to 'extend' a module to produce a new, functionally richer module.

When a large module is refactored into a composition of smaller modules, package splitting may be necessary.

4.3.1 Package Splitting

Sometimes a large package can be split into several relatively independent pieces which do not need package private access to each other. It may then be appropriate to break a module containing the original package into several modules containing the pieces.

Ideally package names of the pieces would be renamed and all occurrences changed. Additionally, the metadata of modules importing the package would be modified to import the appropriate pieces from the new modules.

However, sometimes this ideal is not practicable and it is not possible to modify the code using the package or the modules importing the package. In this case, it should still be possible to split the module containing the package into multiple modules and then compose the split pieces into a façade module without impacting the importers.

4.4 Bundle Dependencies

As the complexity of the system increases, we need to have additional notions of bundle dependencies and class sharing between bundles. This will allow a simpler expression of the dependencies and class constraints between bundles. In some ways, it can be simpler and less error-prone for one bundle to express its dependencies on another bundle rather than simply on packages alone. This is especially true when a bundle may import many packages with detailed constraints. This requirement also allows for a dependent bundle to be refactored without affecting the bundle which depends upon it.

A bundle can express dependencies on specific packages by importing them and specifying package versions. The import package model allows a bundle to depend on the existence of a specific package version but it does not allow a bundle to specify that it depends on a specific bundle to be present. A bundle must be able to express that it requires another bundle at a particular bundle version.

The dependency on another bundle can be considered an implementation version dependency and the bundle will require the packages explicitly provided by the other bundle. A package explicitly imported will take precedence over a package obtained by requiring a bundle.

A bundle must be able to locate another bundle that it requires. A bundle must be able to specify a symbolic name and version so that it can be uniquely identified in the framework. A Bundle-Name manifest header is defined but its value is a human readable name for the bundle. A developer must have a way to specify a bundle's unique symbolic name and bundle version.

A Bundle-Version manifest header is defined but its value has no defined meaning. For effective management, a defined meaning is necessary. It is necessary to distinguish between versions of a bundle in a management system.

The value of the version should be defined to be the implementation version of the bundle in the form: major.minor.micro.qualifier. This implies an ordering among the versions of a bundle. With a well defined Bundle-Version, one bundle can express its dependencies on other bundles. Such expressions on bundle version should allow for a range of match types. (e.g. exact version, same major, same major and minor, same major and minor greater than or equal to, etc.).

4.4.1 Use Cases

1. An application may be made up of several bundles which work together. Some of the bundles are written by the same developer and the developer has dependencies between his bundles on the packages contained within the bundles. The developer wants to express the dependencies between the packages of his bundles without the need to export the packages into the package namespace with a given specification version. These dependencies are really implementation dependencies. The developer is not attempting to evolve these packages as one might expect for specification version.

   However the developer also uses some well known package which do have specification versions (e.g. JAXP, servlet). So the developer also wants to express his dependencies on these packages at some specification version.

2. A developer wants to use the Xerces parser which has been packaged as a bundle. The Xerces parser consists of many packages which contain the implementation of the parser. These packages do not have specifications detailing their contents and may change between releases of Xerces in non compatible ways. So the Xerces bundle would be defined to provide all the Xerces packages and the bundle would be versioned at the Xerces implementation version. The developer can then specify that his bundle is dependent

on the Xerces bundle as a module using some versioning constraints (e.g. = 4.1.*). When the developer bundle is resolved, its class loader will then have access to all the provided packages in the Xerces bundle.

3. A bundle may deliver content that is not participating in package sharing. For example, a bundle may deliver help content, native executables or other such files that other bundles depend on. The developer can specify that the bundle that delivers the files is required in order for their bundle to be resolved.

4. A bundle may act as a container (for example Configuration Admin, or the Web Container). Such bundles listen for particular services to be registered by other bundles (org.osgi.service.cm.ManagedService or org.osgi.service.webapplication.WebApplicationService) and use these services to plugin into a container. A container bundle usually will not export any packages so there is no way for a bundle to specify a dependency with import package on the container bundle. If a container bundle could be required by another bundle that wishes to plug into it then such dependencies could be expressed by the developer of a bundle. In this use case a container bundle would not share any packages with other bundles but the framework would not resolve any bundle requiring the container bundle because it offers functionality that the other bundles depend on.

5. A developer writes a multi-bundle application including third party bundles. The application goes through extensive testing. Code is shared between the different bundles of the application. To ensure the integrity of the application the developer must be able to express a specific dependency between the bundles of the application, in particular to the third party bundles. The developer does not want the application deployed with any version of the third party bundle except the one tested with.

4.5 Bundle Singletons

There are times when it is appropriate that only a single version of a bundle is resolved in the framework. This may be necessary when the bundle manages a resource for which there can be only one manager. The framework needs to provide a mechanism where a bundle can be declared as a singleton to prevent another version of the bundle from being resolved.

4.5.1 Use Cases

1. A bundle provides the UI framework for the device. There is only one display on the device and correspondingly there should only be one bundle providing the UI framework. The bundle is marked a singleton so the framework can prevent attempts to install or use another version of the bundle.

2. A bundle provides the driver logic to manage a physical network connected to the device (e.g. EHS). There must only be one such bundle installed. The EHS network bundle must declare that it is a singleton bundle for the purpose of EHS network management. If an attempt is made to install another EHS network bundle which also declares that it is the same kind of singleton bundle, then the framework must only allow one of the EHS network bundles to be resolved at any given time.

5 Technical Solution

5.1 Overview

The modularity features of the OSGi framework are extended and made more flexible by extending the corresponding metadata. A bundle manifest version is defined in order to enable bundle manifests written in R3 syntax to be supported.

The framework controls the loading of each module's classes and resources and the use of those classes and resources by other modules. Each module has its own class loader and an internal search path of jar files from which the module's classes and resources are loaded. A module specifies which of its classes are available for sharing with other modules.

A detailed discussion of class-loader based modularity is contained in [3] and [4].

5.1.1 Class Spaces

Assuming that a module is associated with a unique class loader and that module class loaders form a delegation network, then a class space is defined[2] as all classes reachable from a given module class loader. Thus, a class space for a given module contains the classes of the parent class loader, classes in imported packages, and classes on the module's internal class path. A class space is *consistent* in that it never contains two classes with the same fully qualified name, although separate class spaces may contain distinct classes with the same fully qualified name.

Note that a module may load any class from the module's internal class path unless the class is loaded by the parent class loader or the class belongs to an imported package. In particular, it is not necessary for a module to import packages from itself in order to access them.

5.1.2 Metadata

Each module has metadata that defines the packages it imports from other modules, packages it exports to other modules, and packages it imports publicly. To enable consistent class spaces to be constructed automatically when modules are resolved, we use two metadata statements:

- Import-Package allows a module to access a package from another module and optionally propagate the package to other modules. Importing a package hides any corresponding package contained in the importing module.
- Export-Package allows a module to provide a package for others to import. The exporting module contains the package and does not implicitly import the package[3].

Although Import-Package and Export-Package appear to be mutually exclusive, it is valid for a module to specify both for a given package. When the module is resolved, either the import or the export of the package will be ignored by the framework. If a module both imports and exports a given package, it thereby allows the framework to substitute the package from another module for the module's own copy.

[2] This is quite different to the definition of class space which was proposed by RFC 70.
[3] Readers familiar with OSGi R3 may be confused at this point since Export-Package in OSGi R3 implies an import of the same package. The OSGi R3 function is supported as described in section 5.5.4.

5.1.3 Attributes

The OSGi framework defines many concepts that are applicable for a generic module framework for Java, but these concepts need to be further abstracted to support more use cases since module-orientation was not its explicit goal.

Similar to the OSGi framework, any generic Java module framework must define a packaging format for modules, where modules declare imports and exports based on Java packages. For flexibility and extensibility, a generic module framework must allow arbitrary metadata attributes to be attached to module import/export declarations, in contrast to the OSGi framework, which only supports version metadata.

Arbitrary metadata attributes are used for two purposes: to provide directives to the underlying framework on how the import/export declaration is processed and to define matching constraints when resolving exporters to importers. Attributes have simple, single values and are matched using equality comparison, except certain version attributes that are special in the sense that they may specify a required range. This special treatment of version attributes provides for expressive version compatibility capabilities, in contrast to the OSGi framework, which enforces "greater than or equal to" version compatibility.

Arbitrary attributes used for matching enable the full spectrum of provider selection. For example, a module may attach a vendor name and a unique module name to all of its exports. For modules where anonymous providers are sufficient, their import declarations only need to declare package name and version. Another module may include the vendor name in its import declarations if it wants a particular vendor, but does not care which module supplied them. Yet another module may include the module's unique name in its import declarations, ensuring that it gets the packages from a precise module.

Further, an exporter can indicate that certain arbitrary attributes must be specified by a matching import statement. This can be used, for example, to restrict the importers to having certain properties.

If a given attribute is specified more than once (in the same semicolon separated list of parameters), the install of the bundle must fail.

5.1.4 Multiple Version Export Considerations

Allowing multiple modules to export a class with a given name causes some complications for framework implementers. The class name no longer uniquely identifies the exported class. This affects the service registry and permission checking.

5.1.4.1 Service Registry

The service registry must deal with multiple versions of a particular interface being published. It should record the module which registered each interface and use this information to match against a module later attempting to find the interface. A find should be satisfied with the version of the service interface imported by the module attempting to find the interface.

The following methods need to filter ServiceReference objects depending on the version of a package a bundle has access to:

- BundleContext.getServiceReference(String)
- BundleContext.getServiceReferences(String, String)

Before returning the ServiceReference objects, getServiceReferences does the following for each ServiceReference:

1. Get the Bundle object (‘getterBundle’) for the BundleContext

2. For each class or interface name under which the ServiceReference was registered, call ServiceReference.isAssignableTo(getterBundle, classOrInterfaceName).

3. If any of these calls returns false, exclude the ServiceReference in the return value, otherwise include it.

BundleContext.getServiceReference returns the highest ranked service that BundleContext.getServiceReferences returns. If getServiceReferences returns no service references, then getServiceReference returns null.

ServiceListener.isAssignableTo(Bundle, String) performs the following checks:

1. If the Bundle object (‘registrant bundle’) of the BundleContext which registered the service is equal to the specified bundle, then return true.

2. Get the package name from the specified String class name.

3. For the specified bundle, find the wiring for the package. If no wiring is found, return true[4].

4. For the registrant bundle, find the wiring for the package. If no wiring is found, then find the wiring for the class loader of the service object. If no wiring is found, return false.

5. If the wiring of the specified bundle is equal to the wiring found in the step 4, then return true; otherwise return false.

Some bundles may wish to get all service references available in the framework regardless of what version of a package they are wired to. A new method is added (BundleContext.getAllServiceReferences(String, String)) to allow a bundle to get all ServiceReference objects. getAllServiceReferences returns all the ServiceReference objects without performing any class tests.

5.1.4.2 Service Events

Service events must only be delivered to event listeners which can validly cast the event. Before the framework delivers a ServiceEvent to a ServiceListener it does the following:

1. Get the Bundle object (‘listenerBundle’) for the BundleContext which was used to register the ServiceListener.

2. For each class or interface name (a String) under which the ServiceReference was registered, call ServiceReference.isAssignableTo(listenerBundle, classOrInterfaceName) .

3. If any of these calls returns false, do not deliver the event. Otherwise, deliver the event.

Some bundles may wish to listen to all service events regardless of what version of a package they are wired to. A new type of ServiceListener is added (AllServiceListener) to allow a bundle to listen to all service events. When an AllServiceLisetener is used the framework does not do class checks on the ServiceReference for the ServiceEvent before delivering it to the AllServiceListener.

Similarly, a new type of SeviceTracker is added (AllServiceTracker). The AllServiceTracker allows a bundle to track all services regardless of what version of a package they are wired to. The AllServiceTracker is identical in function to the ServiceTracker API except it will register an AllServiceListener to track ServiceReference objects for a bundle.

[4] This is for cases where the requesting bundle does not import for the package. It is assumed that this bundle will use reflection and there is no need to do further wiring checks.

5.1.4.3 Permission Checking

Since multiple modules may export permission classes with the same class name, the framework must make sure that permission checks are performed using the correct class.

The net result is that the framework needs to look up permissions based on class rather than class name and when it needs to instantiate a permission it needs to use the class of the permission being checked to do the instantiation. This is a complication for framework implementers; bundle programmers are not affected.

Consider the following example:

```
  Bundle A imports com.acme, exports com.pack
  Bundle B imports com.acme, exports com.pack
  Bundle C imports com.acme (does not import com.pack)
```

Bundle A registers a com.acme.FooService that does a check for the com.pack.FooPermission whenever one of its methods is invoked.

Bundle C has FooPermission in its ProtectionDomain.

Now bundle C invokes a method from FooService from A and the FooService does the checkPermission for FooPermission. When the check happens the framework will need to instantiate the FooPermission in bundle C's protection domain to see if it implies the FooPermission being checked. The FooPermission needs to come from A’s com.pack otherwise the permission check will not work. (Note that, after the check, C will have a FooPermission instantiated using a class from a package it doesn't import. This is not a problem – just an observation.)

5.1.5 Exporting System Classes

Classes on the system class path (i.e. accessible via the system class loader) pose a challenge for accurately resolving exports to imports. If the standard Java class loader delegation pattern is used (i.e. always delegating to the parent class loader before trying to find a class locally), then classes on the class path will always be found instead of explicitly resolved ones. Furthermore, it complicates creating framework-independent bundles, since it is possible that one framework might make a package visible through the system class loader whereas another framework might make the same package available via an export; thus, a bundle that does not import the package in question will work on one framework and not the other and likewise if a bundle does import the package and the framework does not explicitly export it. As a result, it is important to define a systematic approach for dealing with classes on the system class path.

Any approach in the module layer for dealing with classes on the system class path must support two important use cases:

1. **Implicitly imported class path packages** – Certain packages on the system class path are required to execute the framework and form integral pieces of the underlying Java runtime (e.g. java.lang). The framework and all executing bundles are required to use the same class definitions for these packages. Since bundles have no choice when it comes to these packages, they should not have to declare explicit import declarations for them.

2. **Explicitly imported class path packages** – Certain packages on the system class path are ancillary packages, either made available by the installed Java runtime installation or the framework deployer, and are not required by the framework. Such packages may be offered to bundles, but it is also possible for bundles to offer alternative implementations of the same packages. Since bundles have a choice when it comes to these packages, they should declare explicit import declarations for them.

In keeping with previous OSGi specifications, the set of implicitly imported class path packages is defined as all java.* packages, since these packages are required by the Java runtime and using multiple versions at the same time is not easily possible. Explicitly, this means that a bundle must not declare imports or exports for java.* packages; doing so is considered an error and any such bundle should fail to install. All other packages accessible via the system class path must be hidden from executing bundles.

Explicitly imported class path packages are a way for the framework to export classes on the system class path to other bundles, effectively “unhiding” them. To achieve this, the system bundle is used to export non-java.* packages on the system class path. Packages exported by the system bundle are treated like a normal export, meaning that they may have version numbers, are used to resolve import package declarations as part of the normal bundle resolution process, and other bundles may provide alternative implementations of the same packages. A system property is introduced to delineate all packages on the class path exported via the system bundle (e.g. org.osgi.framework.systemPackages); this property employs the standard export manifest header syntax:

```
  org.osgi.framework.systemPackages ::= package-description ( ',' package-description )*
```

The value of this property is set either manually or automatically calculated by a framework. The exported packages have a bundle symbolic name value of “system.bundle” and a bundle version value of “0”. Exposing packages from the system class path in this fashion must also take into account any dependencies among the underlying packages. For example, a package dependency would be necessary if a class in an exposed package refers to a class in another package also exposed by the system bundle.

To support both implicit and explicit package imports, the standard class loader delegation pattern for the bundle class loader has to be modified as described below in section 5.1.5.1. In particular, delegation to the parent class loader only occurs if the requested class is from a java.* package.

The modified delegation pattern and the system packages property enable modules to be created in a framework-independent fashion and also enable scenarios where alternative packages can be made available for packages on the system class path.

5.1.5.1 Class Loading Search Order

When an R4 bundle’s class loader attempts to load a class or find a resource, the search must be performed in the following order:

1. If the class or resource is in a java.* package , the request is delegated to the parent class loader; otherwise the search continues with the next step. If the request is delegated to the parent class loader and the class or resource is not found, then the search terminates and the request fails.
2. If the class or resource is a package included in the boot delegation list as described in section 5.1.5.3, the request is delegated to the parent classloader; otherwise the search continues with the next step. If the class or resource is not found, then the search continues with the next step.
3. If the requested class or resource is in a package that is imported using Import-Package or that is dynamically imported as previously established by step 6, then the request is delegated to the exporting bundle’s class loader; otherwise the search continues with the next step. If the request is delegated to an exporting class loader and the class or resource is not found, then the search terminates and the request fails.
4. If the requested class or resource is in a package that is imported from one or more other bundles using Require-Bundle, the request is delegated to the class loaders of the other bundles, in the order in which they are specified in this bundle’s manifest. If the class or resource is not found, then the search continues with the next step.
5. The bundle’s own internal class path is searched. If the class or resource is not found, then the search continues with the next step unless the requested class or resource is in a package that is imported from one or more other bundles using Require-Bundle, in which case the search terminates and the request fails.
6. If the requested class or resource is in a package that is imported using DynamicImport-Package, then a dynamic import of the package is now attempted, otherwise the search terminates and the request fails. If the dynamic import of the package is established, the request is delegated to the exporting bundle’s class loader.

If the request is delegated to an exporting class loader and the class or resource is not found, then the search terminates and the request fails.

When delegating to another bundle class loader, the delegated request enters this algorithm at step 3. The following non-normative flow chart illustrates the search order described above:

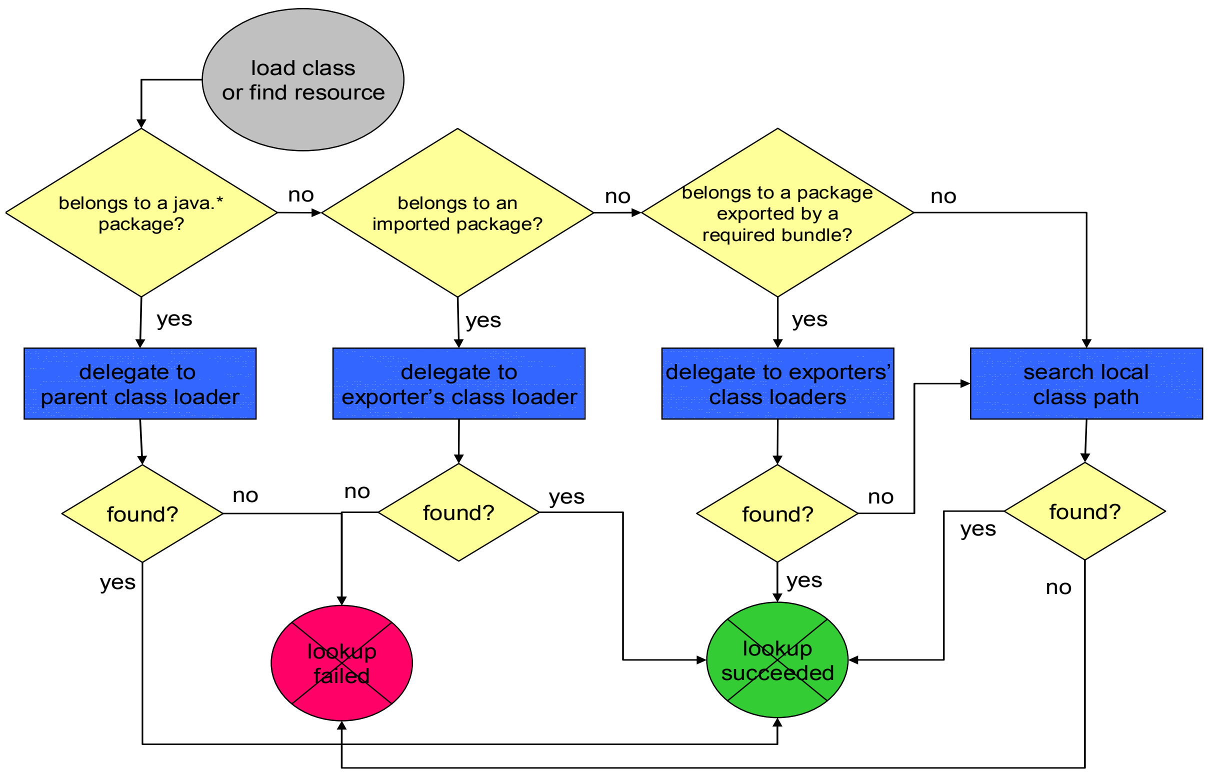

load class or find resource
belongs to a java.* package?
no
belongs to an imported package?
no
belongs to a package exported by a required bundle?
no
yes
yes
yes
delegate to parent class loader
delegate to exporter's class loader
delegate to exporters' class loaders
search local class path
found?
no
no
found?
yes
found?
no
yes
found?
yes
yes
no
lookup failed
lookup succeeded

Figure 1: Class Loading Search Order

5.1.5.2 Class Loading Search Order for R3 Compatibility

Since previous versions of the specification were incomplete and/or contradictory regarding the visibility of packages provided via the system class path, it is possible that legacy bundles exist that violate the rules defined above. To smooth this transition with legacy bundles, a framework must modify its behavior slightly for R3 or lower bundles. For such legacy bundles, the following class/resource search order is acceptable:

1. Delegate the request to the parent class loader and if the class or resource is found, return this result. If the class or resource is in a java.* package and it was not found by the parent class loader, then the request fails. If the class or resource was not found and is not in a java.* package, then the search continues with the next step.

2. If the requested class or resource is in a package imported, either statically or dynamically, by this bundle, then the request is delegated to the exporting bundle's class loader; otherwise the search continues with the next step. If the request is delegated to an exporting bundle's class loader and the class or resource is not found, then the search terminates and the request fails since imported packages are not treated as split.

3. The bundle's own internal class path is searched. If the class or resource is not found, then the search terminates and the request fails.

This algorithm always attempts to load a class or resource from the parent class loader first, before consulting any exporting bundles; this allows legacy bundles to load classes from the system class path without explicitly importing them. This algorithm still does not allow legacy bundles to export java.* packages. When delegating to another bundle class loader, the delegated request enters this algorithm at step 3. This algorithm only applies to R3 or lower legacy bundles; R4 bundles must conform to the strict definition.

The following non-normative flow chart illustrates the search order described above:

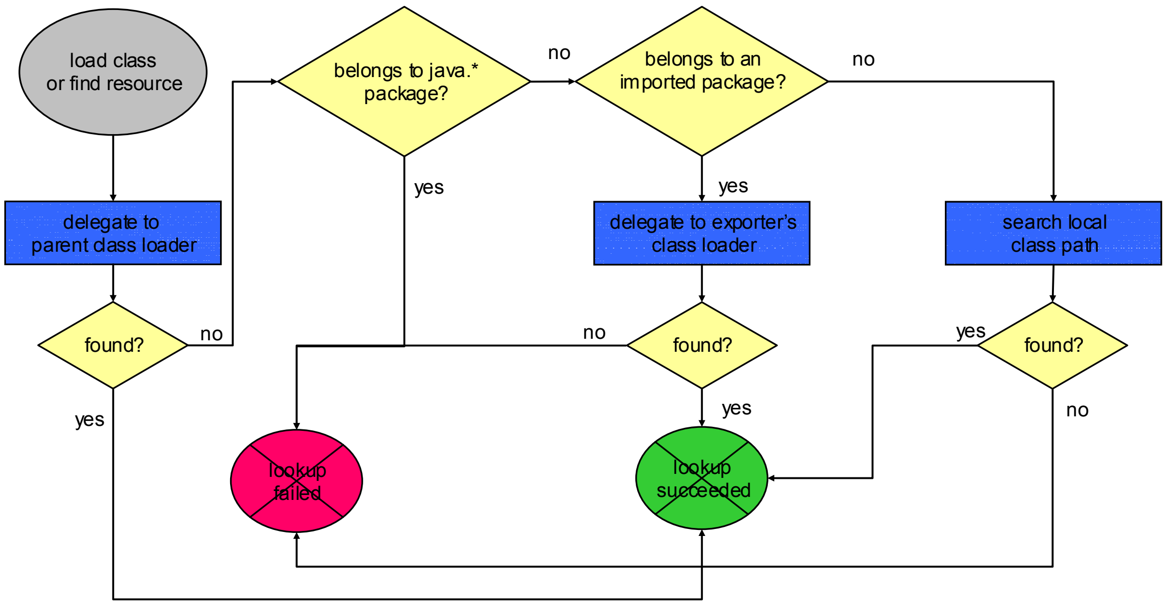

load class or find resource
delegate to parent class loader
found?
no
yes
belongs to java.* package?
no
yes
belongs to an imported package?
no
yes
delegate to exporter's class loader
found?
no
yes
search local class path
found?
yes
no
lookup failed
lookup succeeded

Figure 2: Class Loading Search Order for R3 Compatibility

5.1.5.3 Parent Class Loader Delegation

The class loading search order defined in section 5.1.5.1 explicitly requires that R4 compliant frameworks delegate only java.* packages to the parent class loader. The goal of this requirement is to make bundles more portable among different framework implementations as well as to allow packages on the class path to be hidden or for alternative implementations to be provided. However, certain Java virtual machines appear to make the erroneous assumption that parent class loader delegation always occurs. This implicit assumption of strictly hierarchical class loader delegation can result in NoClassDefFoundErrors if the virtual machine implementation expects to find its own implementation classes from any arbitrary class loader. The end result is that it is likely not to be possible on all JVMs to only delegate java.* classes to the parent class loader.

To deal with this situation, a system property is introduced (org.osgi.framework.bootdelegation) to delineate all package prefixes that should be delegated to the parent class loader; the property value is a comma delimited list. The “java.” prefix is hard coded and any package prefixes specified in the system property are appended to it. As an example, when running on a Sun Microsystems JVM, it may be necessary to specify a value such as:

org.osgi.framework.bootdelegation=sun.,com.sun.

With such a property value, the framework would delegate all java.*, sun.*, and com.sun.* packages to the parent class loader. The intention of this property is not to circumvent the strict class search order defined in section 5.1.5.1; rather, it is purely intended as a work-around for faulty JVM implementations. If an empty string is specified, then all packages will be delegated to the parent class loader, which will result in semantics similar to R3 that has downsides of non-portability of bundles and an inability to hide classes on the boot class path.

5.1.6 Bundles and Modules

Bundles are related to modules as follows: a given bundle archive defines a module; installing a bundle creates a module; updating a bundle may create another module based on the updated bundle archive and so there may be a one-many relationship between the bundle object and the modules created for the bundle. However, a bundle archive will not create more than one module unless it is modified.

When the R4 spec. is written, it would probably be advisable to standardize on the term "bundle".

5.1.7 Module Composition

A Require-Bundle construct is introduced to express a dependency on another module, via its bundle symbolic name, to import all its exported packages, and optionally to re-export these packages. If multiple modules are composed using Require-Bundle, then any packages which are exported from the composed modules and implemented by the composing module are treated as split and are merged.

Since module composition is more likely to be required on larger service platforms, an OSGi framework implementation may opt not to support it.

5.2 Bundle Manifest Version

A bundle manifest may express the version of the syntax in which it is written by specifying a bundle manifest version. Bundles exploiting OSGi R4, or later, syntax must specify a bundle manifest version.

```
  Bundle-ManifestVersion ::= digit+
```

The following value is architected:

- 2 – the bundle manifest version defined by OSGi R4 or, more specifically, by V1.3 of the OSGi Framework Specification.

Bundle manifests written to OSGi R3, or earlier, syntax are taken to have a bundle manifest version of '1', although there is no architected way to express this in such manifests.

Although the bundle manifest version is anticipated to increase monotonically in future releases of OSGi, it is deliberately decoupled from both the OSGi release number and the framework version number. Hence it is not necessary to change the bundle manifest version unless the syntax of the bundle manifest changes or the semantics of the bundle manifest changes incompatibly.

All OSGi implementations should (but are not required to) support bundle manifests with back-level versions but should fail to install bundles with 'future' versions relative to the implemented release of OSGi.

Version 2 bundle manifests must specify the bundle symbolic name. They need not specify the bundle version as this has a default value.

If the syntax of the bundle manifest does not conform to its manifest version, the bundle should fail to install, although unrecognized headers are ignored.

5.3 Bundle Object

A bundle object can be uniquely identified within a framework using the bundle identifier assigned to it by the framework during install time or by the location that was assigned to it during install time. Both of these identifiers on the bundle object allow applications a way to access other bundle objects, but neither provides applications a predictable way of accessing a specific bundle object in advance of bundle deployment.

For example, if bundle A wants to obtain the Bundle object for bundle B it cannot always lookup a bundle using the identifier 15 because bundle B will not always get assigned the identifier 15 by the framework. Similarly, if bundle A wants to obtain the Bundle object for bundle B it cannot always search for a bundle object that has a location of http://com.foo.bundles/bundleB.jar because bundle B could be installed using a different location value.

5.3.1 Bundle Symbolic Name

A new identifier is defined that allows for a predictable, programmatic way of accessing specific bundles. The Bundle-SymbolicName manifest header can be used to identify a bundle. The Bundle-SymbolicName is assigned to a bundle by the developer. The Bundle-SymbolicName manifest header is used to give a symbolic name to a bundle. It does not replace the need for a Bundle-Name manifest header, which provides a human readable name for a bundle. Version 2 bundle manifests must specify the bundle symbolic name.

The Bundle-SymbolicName manifest header must conform to the following syntax:

```
Bundle-SymbolicName ::= symbolic-name (';' directive)*

symbolic-name :: = token('.'token)*

directive ::= name ':=' value

name ::= token

value ::= token | quoted-string
```

The following directives are recognized by the framework for Bundle-SymbolicName

- **singleton**. Indicates that the bundle is a singleton bundle. The string value of "`true`" indicates that the bundle is a singleton bundle. The default value is "`false`".

The Bundle Symbolic Name and Bundle Version allow for a bundle to be uniquely identified in the framework. If a Bundle Symbolic Name is not specified, then the bundle has no Bundle Symbolic Name. If a bundle does not have a Bundle Symbolic Name then the bundle cannot be a fragment host (see RFC 70) nor be required by other bundles (see 5.6). This allows the framework to support legacy bundles that do not take advantage of the new features of the framework.

5.3.1.1 Singleton Bundles

At most one version of a particular singleton bundle may be resolved in the framework. When multiple versions of a singleton bundle with the same symbolic name are installed, the framework implementation must select at most one of the singleton bundles to resolve.

Only bundles specified as singleton bundles are treated as such. In particular, singleton bundles do not affect the resolution of non-singleton bundles with the same symbolic name.

For example, suppose a bundle with the symbolic name s1 has four different versions installed and that they have Bundle-SymbolicName and Bundle-Version headers as follows:

```
Bundle-SymbolicName: s1
Bundle-Version: 1.0

Bundle-SymbolicName: s1
Bundle-Version: 2.0

Bundle-SymbolicName: s1; singleton:=true
Bundle-Version: 3.0

Bundle-SymbolicName: s1; singleton:=true
Bundle-Version: 4.0
```

Then the only effect of singleton bundles is that versions 3.0 and 4.0 may not both enter the resolved state at the same time. So, for example, versions 1.0, 2.0, and 3.0 could enter the resolved state together.

5.3.2 Bundle Lifecycle Changes

A bundle with a bundle symbolic name is uniquely identified (among bundles installed in the same framework) by its bundle symbolic name and bundle version.

5.3.2.1 Installing Bundles

If an attempt is made to install a bundle which has the same bundle symbolic name and bundle version as an already installed bundle, then a BundleException must be thrown and the install must fail.

5.3.2.2 Updating Bundles

If an attempt is made to update a bundle which has the same bundle symbolic name and bundle version as another already installed bundle, then a BundleException must be thrown and the update must fail.

5.4 Versions and Version Ranges

Packages and bundles have version numbers with the following syntax:

```
      version ::= unquoted-version | ('"' unquoted-version '"')
      unquoted-version ::= major('.'minor('.'micro('.'qualifier)?)?)?

      major ::= digit+
      minor ::= digit+
      micro ::= digit+
      qualifier ::= (alpha|digit|'_'|'-')+
```

There must be no whitespace in `version`.

The default version is “0.0.0”. If the minor or micro version components are not specified, they have a default value of “0”. If the qualifier component is not specified, it has a default value of the empty string.

Versions may be compared. A version is considered to be **less than** another version if its major component is less than the other version’s major component, or the major components are equal and its minor component is less than the other version’s minor component, or the major and minor components are equal and its micro component is less than the other version’s micro component, or the major, minor and micro components are equal and its qualifier component is less than the other version’s qualifier component (using String.compareTo[5]).

A version is considered to be **equal to** another version if the major, minor and micro components are equal and the qualifier components are equal (using String.compareTo).

Versions of packages and bundles can be selected by specifying a version range. The syntax of a version range is specified using interval notation [8] or a single version.

```
version-range ::= ('”' interval '”') | atleast
interval ::= ['['|'('] floor ’,’ ceiling [']'|')']
atleast ::= version
floor ::= unquoted-version
ceiling ::= unquoted-version
```

In the following table, for each specified range in the left-hand column, a version x is considered to be a member of the range if the predicate in the right-hand column is true.

Range	**Predicate**
“[floor,ceiling)”	floor ≤ x < ceiling
“[floor,ceiling]”	floor ≤ x ≤ ceiling
“(floor,ceiling)”	floor < x < ceiling
”(floor,ceiling]”	floor < x ≤ ceiling
floor	floor ≤ x

5.4.1 Bundle Version

The Bundle-Version manifest header is used to specify the version of a bundle. When the Bundle-ManifestVersion is 2 or higher, the Bundle-Version manifest header must conform to the following syntax:

```
Bundle-Version ::= version
```

5.5 Extended Import/Export

R3 syntax for importing and exporting packages is replaced with a more expressive syntax which can be used to share interfaces or implementations in various ways. A variety of attributes may be used to match imports to exports including the exporter’s bundle symbolic name and bundle version if the importer needs to be tightly coupled to the exporting bundle.

Support for old syntax is carried forward as described in section 5.5.4.

5.5.1 Exporting Packages

The metadata of a module specifies which implementation packages are exported by the module. If multiple packages need to be exported with identical parameters, the syntax permits a list of packages, separated by semicolons, to be specified before the parameters. The same package may be exported more than once with different parameters.

```
Export-Package ::= package-description ( ',' package-description )*
```

[5] noting that bundle manifests must be encoded in UTF8.

```
package-description ::= package-names ( ';' parameter )*

package-names ::= package-name ( ‘;’ package-name )*

package-name ::= <fully qualified package name>

parameter ::= directive | attribute

attribute ::= name '=' value

[directive, name, and value are defined in section 5.3.1.]
```

A token may not contain white space or delimiters (single quote, double quote, semicolon, comma). A value-string may not contain double quotes.

The following directives are architected by the framework for Export-Package:

- **uses** – a string specifying a list of packages upon which the package(s) being exported depend. The default is that the list is empty which means the package(s) being exported do not depend on any other packages. This directive has the following syntax:

  ```
  uses := ‘”’ package-name ( ‘,’ package-name ) * ‘”’
  ```

  Packages in the list which are neither exported nor imported by the current module are ignored and do not create dependencies.

  The framework forms the transitive closure of the dependency relationship. For example, if a bundle exports package p, q, and r and declares that “p uses q” and “q uses r”, then the framework will behave as if the bundle also declared that “p uses r”.

  If a module is wired, for a particular package, to a module which declares the package to be dependent on another package (which the exporting module either imports or exports), then the original module and the exporting module must wire to the same root for the dependent package. Another way of putting this is that the class space of a module which imports both a package and a package which the first package depends upon must be consistent with the class space of the module which exports the first package, at least for that package and the dependent package.

- **mandatory** – a string with no default value. This directive has the following syntax:

  ```
  mandatory ::= ‘”’ name ( ‘,’ name )* ‘”’
  ```

  The list specifies names of matching attributes which must be specified by matching Import-Package statements.

- **include/exclude** – a list of classes of the specified packages which must be allowed to be exported in the case of include or prevented from being exported in the case of exclude. This acts as a class load time filter and does not affect module resolution in any way.

  Care must be taken when using filters. For example, a new version of a module which is intended to be backward compatible with an earlier version should not filter out classes which were not filtered out by the earlier version. Also, when modularizing existing code, filtering out classes from an exported package may break users of the package which rely on those classes.

Filtering works as follows. The name of a class being loaded is checked against the include list and the exclude list. If it matches an entry in the include list but does not match any entry in the exclude list, then loading proceeds normally. In all other cases, loading fails and a `ClassNotFound` exception is thrown. Note that the ordering of includes and excludes is not significant.

The default for include is "*" and for exclude that no classes are excluded. If include or exclude are specified, the corresponding default is overridden (rather than being concatenated to the explicitly specified list of contents).

This directive has the following syntax:

```
include ::= content

exclude ::= content

content ::= '"' content-name ( ',' content-name )* '"'

content-name ::= <class-name> | '*' | ( <content-name> '*' ) |
                         (content-name <class-name>)
```

Class names do not include their package and do **not** end with '.class'. Note that the syntax permits zero or more wildcards to occur at arbitrary positions.

Arbitrary *matching* attributes may be specified. The following matching attributes are architected by the framework:

- **version** – the version of the named packages with syntax as defined in section 5.4. The default value is "0.0.0".
- **specification-version** – this is a deprecated synonym for the version attribute with the sole exception that, if both specification-version and version are specified (for the same package(s)), they must have identical values. If both are specified but with different values, the bundle fails to install.
- **bundle-symbolic-name** – the bundle symbolic name of the exporting bundle is implicitly associated with the export statement. The export statement must not specify an explicit bundle symbolic name.
- **bundle-version** – the bundle version of the exporting bundle is implicitly associated with the export statement. The export statement must not specify an explicit bundle version.

The framework will avoid architecting any matching attributes beginning "x-" in future versions, so it is good practice for bundle developers to prefix their arbitrary attributes with "x-".

Package names must be fully qualified and use the period ('.') character as a separator. Modules cannot export the default package.

If a package version is specified and there is any likelihood that implementations may be developed by independent organizations, one or more additional attributes should be used to differentiate between these separate implementations since the version number alone may be insufficient. Bundle symbolic name and bundle version may be used, for example. A less brittle approach may be more appropriate if the bundle structure is likely to evolve in which case an arbitrary attribute like "developer=org.foo" may be used.

5.5.2 Importing Packages

The metadata of a module specifies which implementation packages are imported by the module. Imports are resolved in the order in which they are specified and determine the order in which class load requests are delegated to other modules. If multiple packages need to be imported with identical parameters, the syntax permits a list of packages, separated by semicolons, to be specified before the parameters.

```
  Import-Package ::= package-description ( ',' package-description )*

  [package-description is defined in section 5.5.1.]
```

The following directives are architected by the framework for Import-Package:

- **resolution** – a string taking one of the values "mandatory" or "optional". The default value is "mandatory".
    - o "mandatory" indicates that the import must be resolved when the importing module is resolved. If such an import cannot be resolved and the importing module does not also export the specified packages, the module fails to resolve.
    - o "optional" indicates that the import is optional and the importing module may be resolved without the import being resolved. If the import is not resolved when the module is resolved, the import may not be resolved before the module is re-resolved.

Arbitrary matching attributes may be specified. The following matching attributes, defined in section 5.5.1, are architected by the framework:

- **version** -- a version range to select the (re-)exporter's implementation version. The default value is "0.0.0".
- **specification-version** – this is a deprecated synonym for the version attribute with the sole exception that, if both specification-version and version are specified (for the same package(s)), they must have identical values. If both are specified but with different values, the bundle fails to install.
- **bundle-symbolic-name** – the bundle symbolic name of the (re-)exporting bundle.
- **bundle-version** – a version range to select the bundle version of the (re-)exporting bundle. The default value is "0.0.0".

Package names must be fully qualified and use the period ('.') character as a separator. Modules cannot import the default package.

In order for the Import-Package to be resolved, any arbitrary attributes specified must match the attributes of the corresponding Export-Package statement[6]. However, a match is not prevented if the Export-Package statement includes attributes which are not specified by the corresponding Import-Package statement, unless one or more of those attributes are exported as required (using the "mandatory" directive).

A module which imports a given package more than once is in error and fails to resolve.

If an import matches exports in more than one module, the module with lowest bundle identifier is selected. If an import is resolved to a module which exports the same package more than once, the first export statement which matches the import is selected and only the dependencies specified (via "uses") for that export statement apply to the importer.

Note that specifying a bundle symbolic name can result in brittleness, for example if the exporting bundle eventually needs to be refactored into multiple separate bundles. Other arbitrary matching attributes do not have this disadvantage as they can be specified independently of the exporting bundle. The problem of the brittleness of bundle symbolic name in bundle refactoring can be overcome by writing a façade bundle with the same bundle symbolic name as the original bundle, but this typically requires a framework which has opted to implement Reexport-Package.

[6] This may seem like an expensive match algorithm since there can be an arbitrary number of attributes, but in practice few attributes are likely to be used, so the matching is likely to be cheap. A significant overhead is only incurred if many attributes are used.

5.5.3 Dynamically Importing Packages

The metadata of a module specifies which implementation packages may be dynamically imported by the module.

Dynamic imports are matched to exports (to form package wirings) during class loading and do not affect module resolution. Dynamic imports apply only to packages which are not imported using Import-Package. Wirings of dynamic import, like those of Import-Package, once established take precedence over those of Require-Bundle.

Packages may be named explicitly or by using wild-carded expressions such as "org.foo.*" and "*". Dynamic imports are searched in the order in which they are specified. The order is particularly important when wild-carded package names are used as the order will then determine the order in which matching occurs. If multiple packages need to be dynamically imported with identical parameters, the syntax permits a list of packages, separated by semicolons, to be specified before the parameters.

As required during class loading, the package of the class being loaded is compared against the specified list of (possibly wild-carded) package names. Each matching package name is used in turn to attempt to wire to an export using the same rules as Import-Package. If a wiring attempt is successful, the search terminates and class loading continues. The wiring is not subsequently modified, even if the class cannot be loaded.

```
DynamicImport-Package ::= dynamic-import-package-description ( ','
                            dynamic-import-package-description )*

dynamic-import-package-description ::= wild-card-package-names ( ‘;’ parameter )*

wild-card-package-names ::= wild-card-package-name ( ‘;’ wild-card-package-name )*

wild-card-package-name ::= package-name | ( package-name ‘.*’ ) | ‘*’

[package-name is defined in section 5.5.1.]
```

No directives are architected by the framework for DynamicImport-Package. Hence, dynamic imports may not be propagated.

Arbitrary matching attributes may be specified. The following matching attributes, defined in section 5.5.1, are architected by the framework:

- **version** -- a version range to select the exporter's implementation version. The default value is "0.0.0"[7].
- **specification-version** – this is a deprecated synonym for the version attribute with the sole exception that, if both specification-version and version are specified (for the same package(s)), they must have identical values. If both are specified but with different values, the bundle fails to install.
- **bundle-symbolic-name** – the bundle symbolic name of the (re-)exporting bundle.
- **bundle-version** – a version range to select the bundle version of the (re-)exporting bundle. The default value is "0.0.0".

Package names must use the period ('.') character as a separator and must either be fully qualified or must each consist of a prefix of a fully qualified package name followed by a period and wild card ('*'). Modules cannot import the default package.

[7] We define a version range consisting of a single value "v" to be the range of values greater than or equal to v, thus avoiding the need to represent "infinity" in version ranges. Note that this definition only applies to version ranges and not to version values such as the package version attribute of Export-Package.

In order for a DynamicImport-Package to be resolved to an export statement, any arbitrary attributes specified must match the attributes of the export statement. However, a match is not prevented if the export statement includes attributes which are not specified by the corresponding DynamicImport-Package statement, unless one or more of those attributes are exported as required (using the "mandatory" directive).

The syntax permits a partial package name, such as "org.foo.*" or even "*", to be specified. This should be regarded as short-hand for dynamically importing all packages whose names match the partial package name with the specified attributes.

5.5.4 Re-exporting Packages

Frameworks may support Reexport-Package but are not required to do so. A framework which supports Reexport-Package must also support Require-Bundle (see section 5.6) and BundlePermission (see section 7.2).

The metadata of a module specifies which imported packages are re-exported by the module. If multiple packages need to be re-exported with identical parameters, the syntax permits a list of packages, separated by semicolons, to be specified before the parameters. The same package may be re-exported more than once with different parameters.

```
Reexport-Package ::= package-description ( ',' package-description )*

[package-description is defined in section 5.5.1.]
```

The following directive is architected by the framework for Reexport-Package:

- **mandatory** – a string with no default value. This directive has the following syntax:

  ```
  mandatory ::= '"' name ( ',' name )* '"'
  ```

  The list specifies names of matching attributes which must be specified by matching Import-Package statements.

Arbitrary matching attributes may be specified. The following matching attributes, defined in section 5.5.1, are architected by the framework:

- **version** – the version of the named packages with syntax as defined in section 5.4. The default value is "0.0.0".
- **bundle-symbolic-name** – the bundle symbolic name of the re-exporting bundle is implicitly associated with the re-export statement. The re-export statement must not specify an explicit bundle symbolic name.
- **bundle-version** – the bundle version of the re-exporting bundle is implicitly associated with the re-export statement. The re-export statement must not specify an explicit bundle version.

The framework will avoid architecting any matching attributes beginning "x-" in future versions, so it is good practice for bundle developers to prefix their arbitrary attributes with "x-".

Package names must be fully qualified using the period ('.') character as a separator. Any module which attempts to re-export a package which is not imported by the module fails to resolve.

The re-export statement does not inherit matching attributes or the 'mandatoryness' of matching attributes via the corresponding import of the package. If these are not specified on the re-export statement, they are either unspecified or take default values where defaults are defined.

5.6 Requiring a Bundle

Frameworks may support Require-Bundle but are not required to do so. A framework which supports Require-Bundle must also support Reexport-Package and BundlePermission (see section 7.2).

The Require-Bundle manifest header imports all the exported packages of one or more named bundles. A given package may be exported by more than one of the named bundles in which case it is treated as a split package and the bundles are searched in the order in which they were specified after which the requiring bundle is searched for any additional classes of the package.

When searching for classes in a required bundle, the framework should only search the packages (re-)exported by the required bundle. Any other packages contained in the required bundle must not be searched.

In order to be allowed to require a name bundle, the requiring bundle must have:

BundlePermission[<required bundle symbolic name>, REQUIRE_BUNDLE]

Note that BundlePermission is defined in RFC 73.

The Require-Bundle manifest header must conform to the following syntax:

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

[symbolic-name is defined in section 5.3.1 and parameter in section 5.5.1.]
```

The following directives are architected by the framework for Require-Bundle:

- **visibility** - a string value taking one of the values “private” or “reexport”. The default value is “private”.
  - “private” indicates that any packages that are exported by the required bundle are not made visible on the export signature of the requiring bundle.
  - “reexport” indicates that any packages that are exported by the required bundle are re-exported by the requiring bundle. Any arbitrary matching attributes with which they were exported by the required bundle are deleted.
- **resolution** – a string taking one of the values “mandatory” or “optional”. The default value is “mandatory”.
  - “mandatory” indicates that the required bundle must be resolved if the requiring module is resolved.
  - “optional” indicates that the requiring module may be resolved without the required module being resolved. If the required bundle is not resolved when the requiring bundle is resolved, the requiring bundle is not affected if the required bundle may subsequently be resolved until such time as the requiring bundle is re-resolved.

The following matching attribute is architected by the framework:

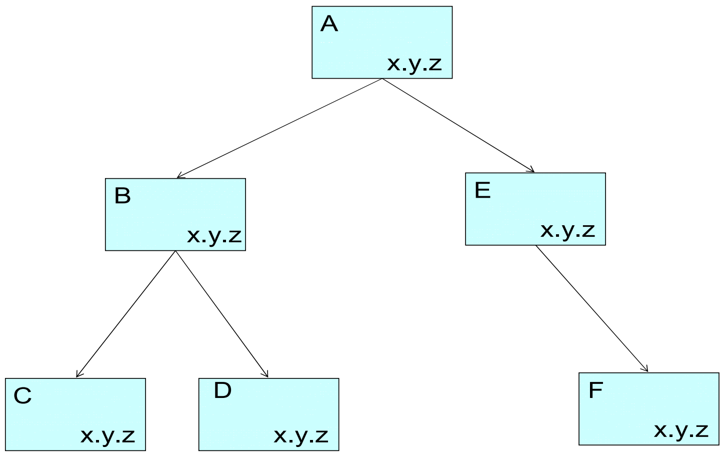
- **bundle-version** - The value of this parameter is a `version-range` to select the bundle version of the required bundle. The default value is “0.0.0”.

A bundle may both import packages (via Import-Package) and require one or more bundles (via Require-Bundle), but if a package is imported via Import-Package it is not also imported via Require-Bundle: Import-Package takes priority over Require-Bundle and packages which are exported by a required bundle and imported via Import-Package are not treated as split.

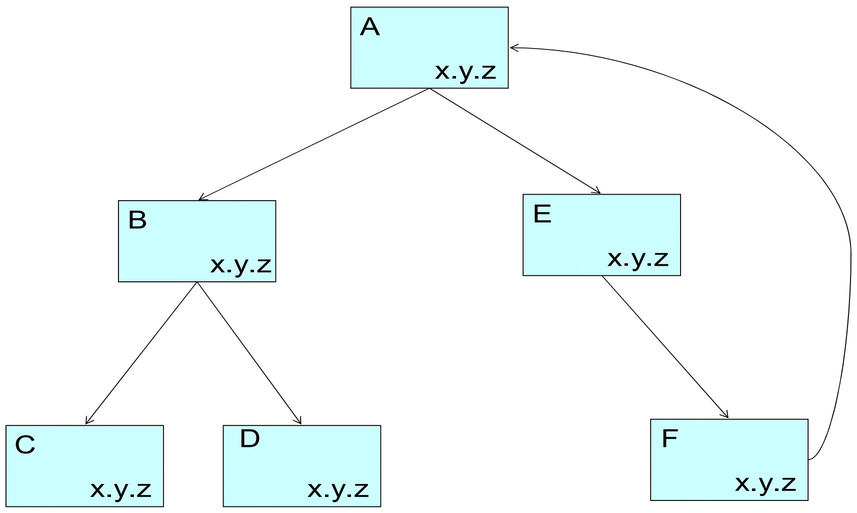
If the required bundle exports the same package more than once, the first export statement is used and only the dependencies specified (via “uses”) for that export statement apply to the requiring bundle.

5.6.1 Require Bundle Cycles

Multiple bundles may export the same package. Bundles which export the same package that are involved in a require bundle cycle can lead to lookup cycles when searching for classes and resources from the package. Consider the following diagram:



This diagram illustrates six bundles A, B, C, D, E, and F. Each of the bundles export the package x.y.z. In this example, A requires B and E, B requires C and D, and E requires F. When the classloader for A is requested to load a class or resource from package x.y.z then the required bundle search order is the following: C, D, B, F, E, A. This is a depth first search (DFS) order. The depth first search order can introduce endless search cycles if the dependency tree has a cycle in it. Using the above diagram, a cycle can be introduced if class space F requires A like the following:



When the classloader for bundle A is requested to load a class or resource from package x.y.z then the bundle search order would be the following: C, D, B, F, A, C, D, B, F, A C, D, B, F, A … . Since a dependency cycle was introduced each time bundle F is reached the search will recurse back to A and start over. The framework must prevent such dependency cycles from causing endless recursive lookups. To avoid endless looping, the framework must mark each bundle upon first visiting it and not explore the required bundles of a previously visited bundle. Using the visited pattern on the dependency graph above will result in the following bundle search order: C, D, B, F, A, E.

5.7 Support for OSGi R3 Bundles

OSGi R4 implementations should support R3 bundles unmodified, including their bundle manifest syntax as described in section 5.2. If such bundles are modified to use new features of the bundle manifest version 2 syntax, they must be migrated as described below since the old syntax must not be mixed with the new.

The following table lists replacements for R3 specific syntax.

OSGi R3 Syntax	Equivalent Bundle Manifest Version 2 Syntax
`Import-Package: <p>; specification-version=”<v>”`	`Import-Package: <p>; version=”<v>”`
`DynamicImport-Package: <p>`	`DynamicImport-Package: <p>`

`Export-Package: <p>; specification-version=”<v>”`	`Export-Package: <p>; version=”<v>”`
	`Import-Package: <p>; version=”<v>”`

`Export-Package: <p>`	`Export-Package: <p>`
	`Import-Package: <p>`

`Export-Package: <p>; specification-version=”<v2>”`
`Import-Package: <p>; specification-version=”<v1>”`
`(where v1 < v2)`
	`Export-Package: <p>; version=”<v2>”`
	`Import-Package: <p>; version=”<v1>”`

A bundle manifest which mixes R3 specific syntax with bundle manifest version 2 syntax is in error and causes the containing bundle to fail to install.

Note: Import-Package with specification version omitted needs no replacement since the default version value of “0.0.0” gives the same semantics as R3.

Note: the specification-version attribute is a deprecated synonym for the version attribute in bundle manifest version 2 headers.

5.8 Installing Modules

The following syntactic and semantic errors cause a R4 bundle to fail to install:

1. Bundle-ManifestVersion value greater than 2.
2. Missing Bundle-SymbolicName.
3. Duplicate attribute or duplicate directive.
4. Mutiple imports of a given package.
5. Export or import of java.*.
6. Export or import of "." (i.e. the default package).

7. Export-Package with a mandatory attribute that is not defined.

8. Installing a bundle that has the same symbolic name and version of an installed bundle.

9. Any syntactic error (e.g. improperly formatted version or bundle symbolic name, unrecognized directive, unrecognized directive value, etc.).

10. specification-version and version specified together (for the same package(s)) but with different values on manifest headers that treat them as synonyms. For example:

    Import-Package p;specification-version=1;version=2

    would fail to install, but:

    Import-Package p;specification-version=1, q;version=2

    would not be an error.

Note that if Bundle-ManifestVersion is not specified, then the bundle manifest version defaults to 1 and certain R4 syntax, such as a new manifest header, is ignored rather than causing an error.

5.9 Resolving Modules

Resolving a set of modules is a complex process. Rather than prescribing a particular algorithm, we describe requirements that any resolution of a set of modules must satisfy.

5.9.1 Class Sharing in Terms of Wiring Requirements

Each package induces a “package wiring” relation between resolved modules. We define some terms to describe this relation. A *root* (of the package) is a module which exports but which does not import the package, a *re-exporter* (of the package) is a module which imports the package and then re-exports it. A *leaf* (of the package) is a module which imports but which does not re-export the package.

A module may only act as a re-exporter of a package if it imports and re-exports the package. A module which both imports and exports a given package will result in one of two wirings:

- The module is chosen as a root of the package and its import statement ignored, or

- The module is chosen as a leaf of the package and its export statement is ignored.

Note that the first of these cases will only happen if the import statement could be satisfied by the export statement. If the import cannot be satisfied by the export, then either the export statement is ignored or the module fails to resolve.

If we consider the transitive closure of a package wiring relation, each leaf must be related to precisely one root, as must each re-exporter. This ensures class space consistency. For example, the package wiring below shows leaves wired to roots via re-exporters. The example also shows that a package wiring may have multiple roots provided no leaf or re-exporter is wired to more than one root.

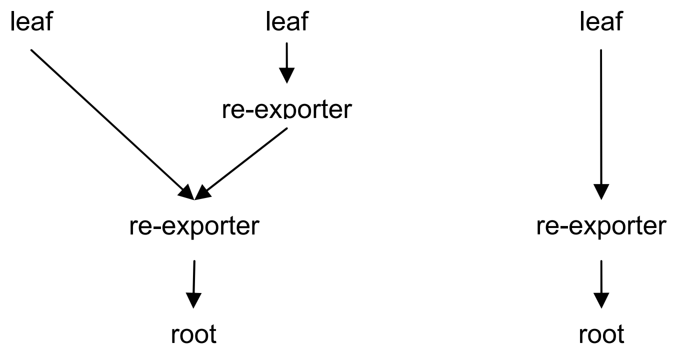


Figure 3: Leaves, re-exporters, and roots of a package wiring

We define a canonical wiring to be a wiring in which there are no re-exporters. Any wiring may be transformed into a canonical wiring (the "canonical form" of the wiring) by rewiring each leaf or re-exporter to its related (by the transitive closure of the wiring relation) root. This process converts all re-exporters into leaves. An example may help:

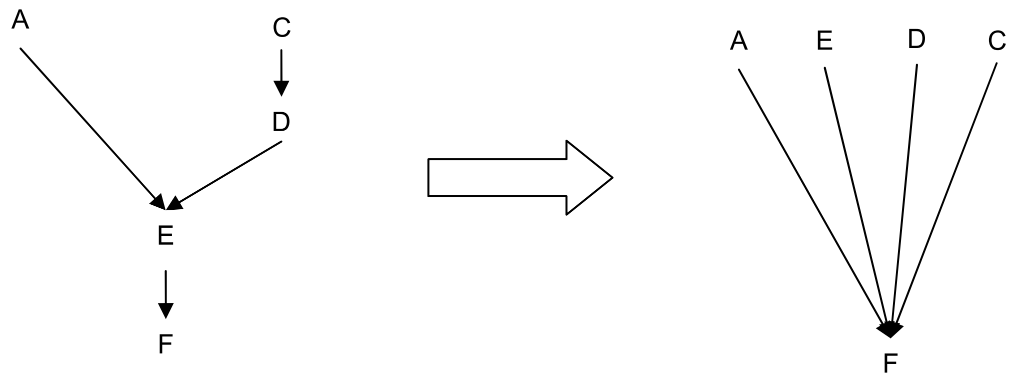


Figure 4: A package wiring and its canonical form

In general, we define two package wirings of a given set of modules to be "equivalent" if both wirings have the same canonical form.

We can now express the class sharing requirements in terms of requirements on wiring relations:

- Version constraints require that each pair of modules related by a package wiring must match with respect to the importer's version range.
- Provider selection requires that each pair of modules related by a package wiring must have matching attributes: the importer's attribute values must match the values of the corresponding attributes in the exporter. Also, any attributes specified as mandatory by the exporter must be supplied by the importer.
- Package dependencies constrain package wirings from a given importer for packages which depend upon each other. If a module is wired, for a particular package, to a module which declares the package to be dependent on another package (which the exporting module either imports or exports), then the original module and the exporting module must wire to the same root for the dependent package.

Figure 5 shows an example. The left hand side shows wirings that are constrained to satisfy this requirement while the right hand shows how the requirement need not apply to other parts of the same wirings (for modules which do not import from the module specifying the dependency).

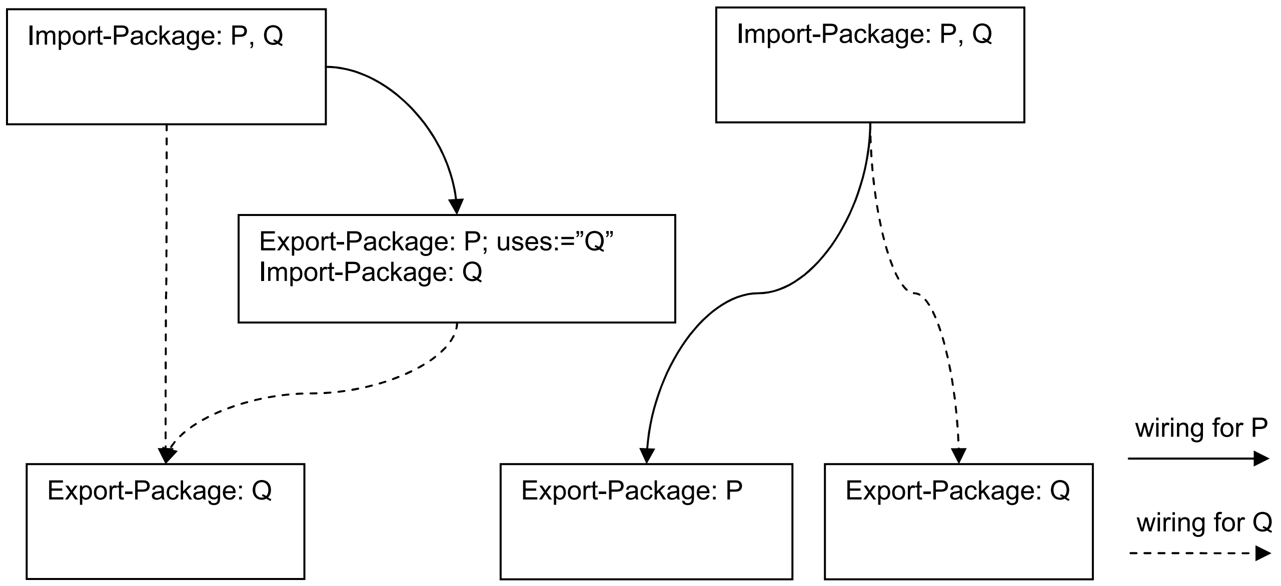


Figure 5: Wirings satisfying the public import requirement

A wiring which satisfies all the above requirements is said to be “valid”.

5.9.2 Resolving Modules (Post-condition form)

Any set of resolved modules has a wiring which is valid. The wiring for a given package must include all imports, exports and re-exports of the package except where a module both imports and exports the package in which case the wiring includes either the import or the export but not both.

As imports are resolved, if there is a free choice between an export from a resolved module and an export from an unresolved module, the export from the resolve module is chosen. If there is a free choice between two exports with different versions, the higher version is chosen unless one of the exports belongs to a resolved module and the other doesn’t, in which case the resolved export is chosen.

Specifying the behavior of a resolver in terms of post-conditions opens up the possibility of spec-compliant resolvers that fail to resolve some simple cases. This can be avoided by the inclusion of a suitable set of tests in the TCK. For example, the TCK should check that a compliant framework is capable of at least one level of back-tracking in the resolver.

5.10 Requirements on Fragments in RFC 70

Fragments as defined in RFC 70 provide a way of dividing a bundle into separately installable pieces: a ‘host’ bundle and one or more fragment bundles which are combined with the host at install time. A fragment is a bundle which identifies the host or hosts with which it is to be combined. Fragments enable a bundle to leave itself open for augmentation after its development is complete.

Since fragments define classes and resources in the host bundle's class loader, the bundle developer's rule of thumb should be to use a fragment if and only if it is necessary to extend (and share the package private parts of) a package defined by the host bundle. A fragment may also ‘shadow’ (i.e. override) some of the hosts classes or resources of a host.

Fragments need to be defined so as to enable static analysis of a host bundle without a priori knowledge of the fragments which will be installed along with the host bundle. **So RFC 70 must be changed to enable a bundle to prevent fragments from attaching to it (even when security is switched off).**

To enable optimizations such as those performed by a Just In Time compiler, the rules for when fragments are resolved also need to change. **RFC 70 must also be changed to restrict fragments so that they may only attach to a host while the host is being resolved and not subsequently.**

With this latter change in place, fragments can be seen to operate at a lower level than the modularity features in this RFC. Essentially, fragments perform restricted kinds of editing operations on their host bundles. This is analogous to the facility to override a bundle’s metadata that we anticipate supporting in the future.

6 Examples

The purpose of the following examples is two-fold:

1. Illustrate typical use cases using the RFC 79 approach.
2. Illustrate more complex aspects of RFC 79 import/export consistency management.

The examples are only intended to be informative, not normative. Note that bundle manifest version 2 syntax is used for R3 examples for ease of comparison with non-R3 examples. Also the resolver behavior described is illustrative and non-normative.

Due to space constraints, the figures in the examples merely hint at the correct RFC 79 syntax. The term ‘module’, as elsewhere in this document, should be interpreted as ‘bundle’ or ‘bundle symbolic name’ depending on the context.

6.1 OSGi R3 Example

The typical OSGi R3 use case is achieved by using an export declaration for each exported package and also a corresponding import declaration for each exported package that accepts an open ended version range. By providing both an export and an import declaration for the same package, the module is giving the framework the choice of using the module as a provider or a consumer of the package. The following meta-data depicts a simple use case:

Meta-data for module *A*:

```
Export-Package: javax.servlet; version=”2.1.0”
Import-Package: javax.servlet; version=“2.1.0”
```

Meta-data for module *B*:

```
Export-Package: javax.servlet; version=“2.2.0”
```

```
Import-Package: javax.servlet; version="2.2.0"
```

Meta-data for module *C*:

```
Import-Package: javax.servlet; version="2.1.0"
```

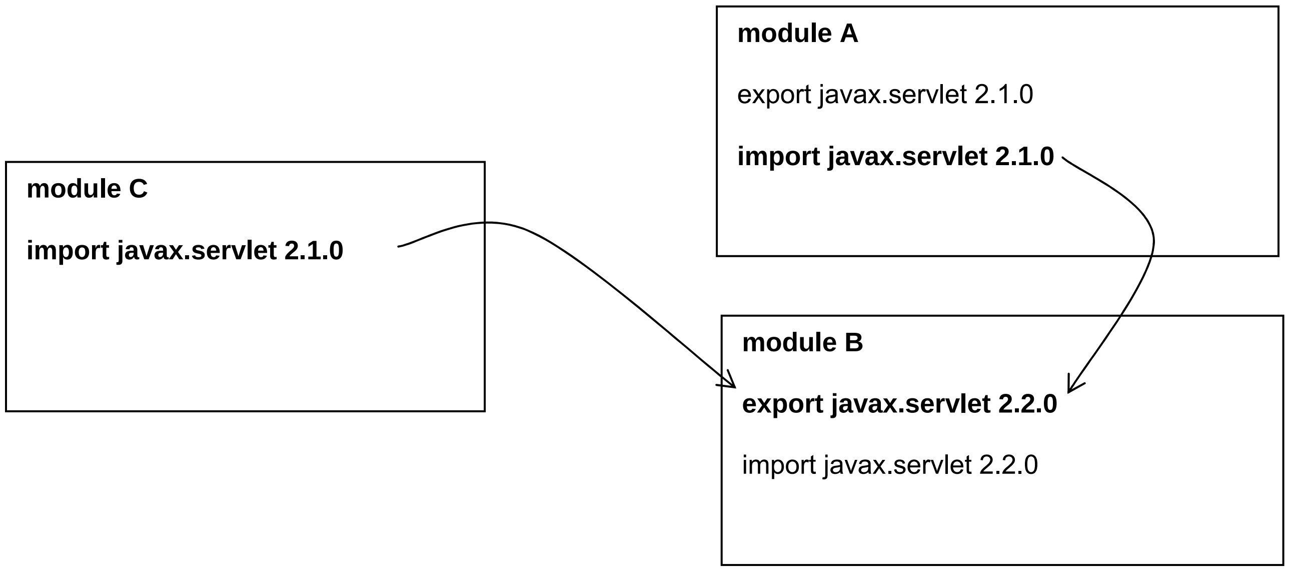

module A
export javax.servlet 2.1.0
import javax.servlet 2.1.0
module C
import javax.servlet 2.1.0
module B
export javax.servlet 2.2.0
import javax.servlet 2.2.0

Figure 6: OSGI R3 Example

Assume for the above example meta-data that the framework has not resolved any modules yet. To resolve module *A*, the framework finds all providers of javax.servlet that match the given import constraint. It finds modules *A* and *B* and selects *B* as its primary candidate because it has the greatest version. The framework determines that *B* does not propagate any packages, so there are no potential conflicts with *A*'s imports. Consequently, the framework attempts to resolve module *B* passing in the javax.servlet package target.

In resolving *B*, the framework ignores the import of javax.servlet. With module *B* resolved, the resolution of module *A* can finish by binding its import declaration to module *B*.

To resolve module *C*, the framework finds all matching providers of javax.servlet. In this case, it finds only module *B*, which was previously resolved. Upon not finding any dependencies ("uses") in module *B*, the framework is free to bind module *C*'s import to it. Any ordering of resolution for these modules leads to the same result, which is expected in the normal OSGi use case.

6.2 Extended OSGi Example with Implementation Packages

The OSGi framework was never intended to share implementation packages, only specification packages. Despite this limitation, users of the OSGi framework regularly abused its specification package sharing mechanism to also share implementation packages along with their specification packages. This led to mixed results since implementation packages were not likely to be eternally backwards compatible as is required of specification packages. RFC 79 explicitly extends OSGi R3 with the ability to share implementation packages. This is supported through the ability to specify restricted version ranges for importers. The following meta-data depicts a simple use case:

Meta-data for module *A*:

```
Export-Package: org.foo.impl; version="1.0.0",
                javax.servlet; version="2.1.0"
Import-Package: javax.servlet; version="2.1.0"
```

Meta-data for module *B*:

```
Export-Package: org.foo.impl; version="1.1.0",
                javax.servlet; version="2.2.0"
Import-Package: javax.servlet; version="2.2.0"
```

Meta-data for module *C*:

```
Import-Package: javax.servlet; version="2.1.0",
                org.foo.impl; version="[1.1.0, 1.1.0]"
```

Meta-data for module *D*:

```
Import-Package: javax.servlet; version="2.1.0",
                org.foo.impl; version="[1.0.0, 1.0.0]"
```

Meta-data for module *E*:

```
Import-Package: org.foo.impl; version="[1.0.0,1.1.0]"
```

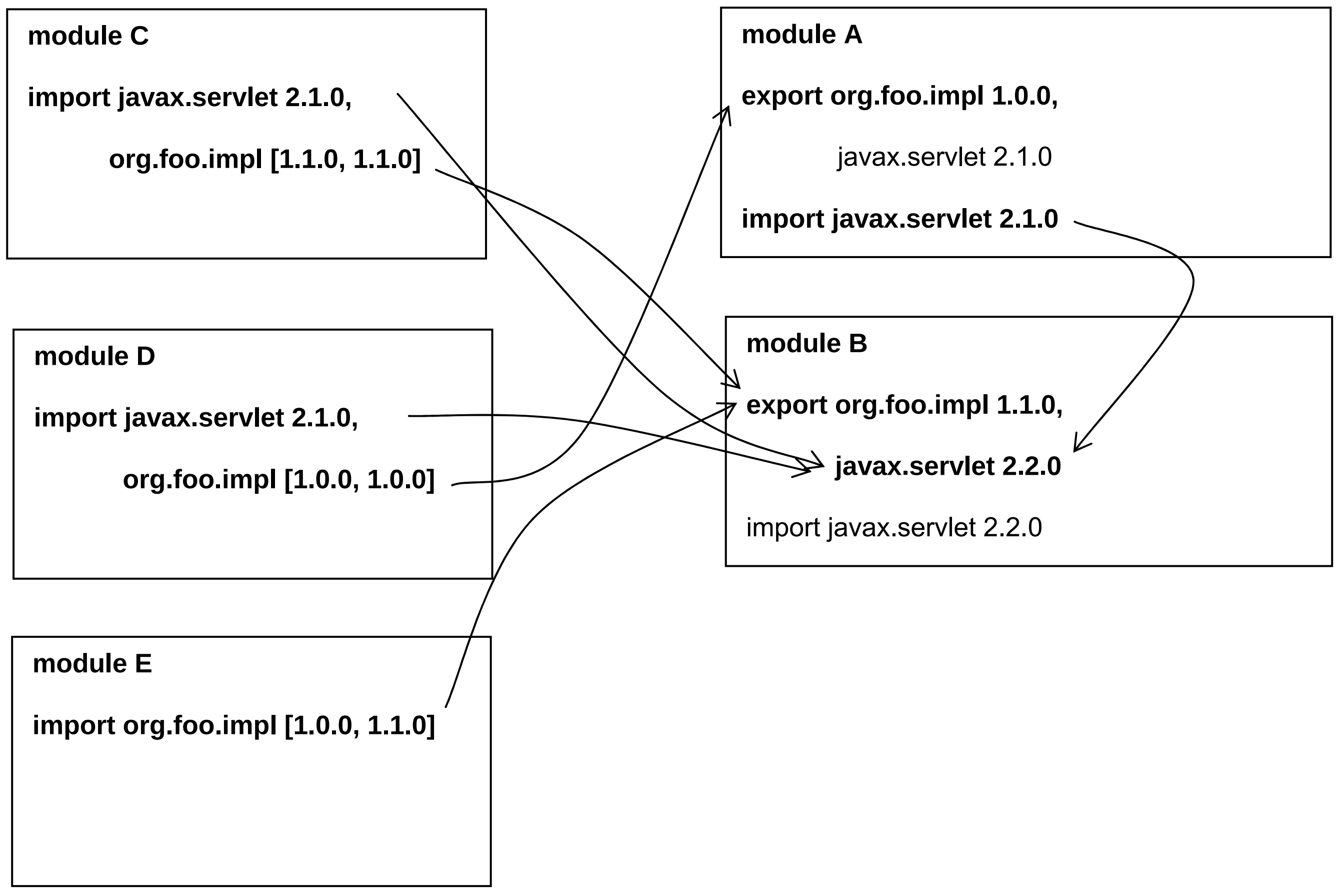

module C
import javax.servlet 2.1.0,
org.foo.impl [1.1.0, 1.1.0]
module A
export org.foo.impl 1.0.0,
javax.servlet 2.1.0
import javax.servlet 2.1.0
module D
import javax.servlet 2.1.0,
org.foo.impl [1.0.0, 1.0.0]
module B
export org.foo.impl 1.1.0,
javax.servlet 2.2.0
import javax.servlet 2.2.0
module E
import org.foo.impl [1.0.0, 1.1.0]

Figure 7: Extended OSGi Example with Implementation Packages

Assume for the above example meta-data that modules *A* and *B* have been resolved in the same fashion as before, resulting in *A* importing and *B* exporting javax.servlet. Resolving module C follows the same steps as the previous example for its javax.servlet import declaration, which will ultimately be bound to module *B*. The org.foo.impl import of module C will be resolved by first finding all possible providers. For this import, only module *B* is a candidate. Since module *B* is already resolved and has no exports which are imported by C and which clash with C's import constraints, module C's resolution can conclude by binding its final import to it.

Resolving module *D* is nearly identical to module *C*, except that module *D*'s import of org.foo.impl is bound to module *A*. Finally, module *E*'s single import has a choice to resolve to either module *A* or *B*, since they both match its constraint. In this case, the framework chooses the greatest version. Any ordering of resolution for these modules will lead to the same result.

6.3 OSGi R3 Example Extended for Bundle Symbolic Name

As described in the previous example, RFC 79 supports implementation package sharing using import version ranges. RFC 70 describes another shared implementation package use case, the ability to import implementation packages from an explicit provider. This is supported in RFC 79 via arbitrary attributes that can be attached to export and import declarations. The following meta-data depicts a simple use case:

Meta-data for module *A*:

```
Bundle-SymbolicName: myfoo

Export-Package: org.foo.impl; version="1.0.0",
                javax.servlet; version="2.1.0"

Import-Package: javax.servlet; version="2.1.0"
```

Meta-data for module *B*:

```
Export-Package: org.foo.impl; version="1.0.0",

                javax.servlet; version="2.2.0"
Import-Package: javax.servlet; version="2.2.0"
```

Meta-data for module *C*:

```
Import-Package: javax.servlet; version="2.1.0",

                org.foo.impl; version="[1.0.0, 1.0.0]"; bundle-symbolic-name="myfoo"
```

Meta-data for module *D*:

```
Import-Package: javax.servlet; version="2.1.0",

                org.foo.impl; version="[1.0.0, 1.0.0]"
```

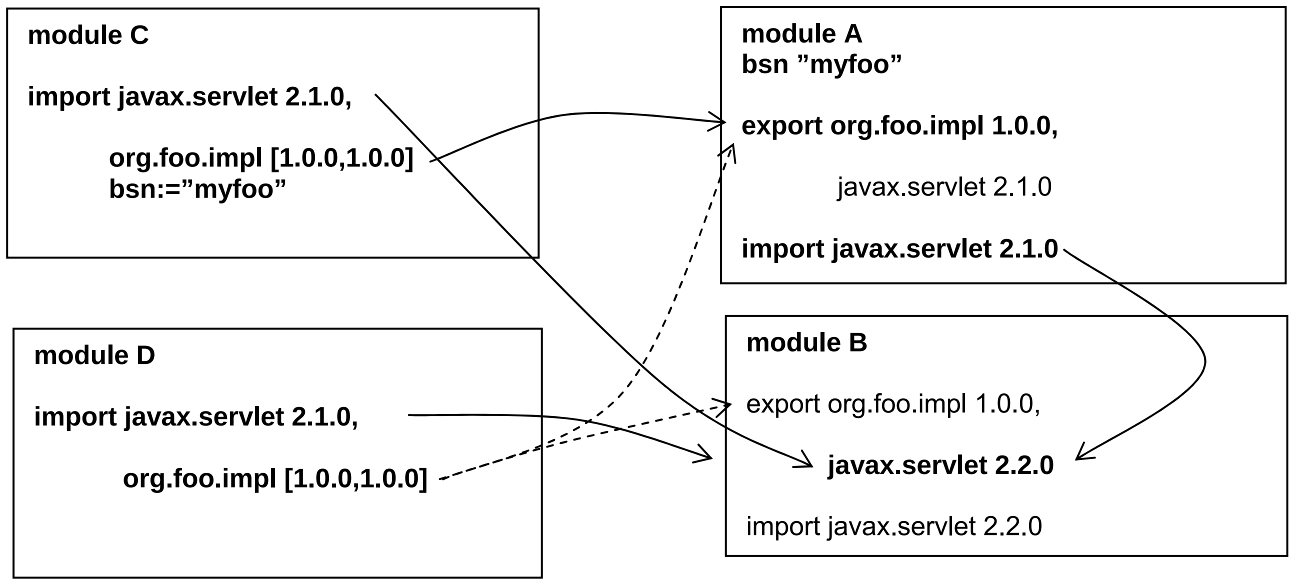

module C
import javax.servlet 2.1.0,
org.foo.impl [1.0.0,1.0.0]
bsn:="myfoo"
module A
bsn "myfoo"
export org.foo.impl 1.0.0,
javax.servlet 2.1.0
import javax.servlet 2.1.0
module D
import javax.servlet 2.1.0,
org.foo.impl [1.0.0,1.0.0]
module B
export org.foo.impl 1.0.0,
javax.servlet 2.2.0
import javax.servlet 2.2.0

Figure 8: OSGi R3 Example Extended for Bundle Symbolic Name

Assume for the above example meta-data that modules *A* and *B* have been resolved in the same fashion as before, resulting in *A* importing and *B* exporting javax.servlet. Resolving module *C* first results in the javax.servlet import being bound to the selected global package of module *B*. For module *C*'s org.foo.impl import, the only potential candidate is module *A*, even though module *B* matches its version constraint. There is no problem binding *C*'s import to *A*'s export of org.foo.impl. Resolving module *D* first results in the javax.servlet import being bound to the selected global package of module *B*. For module *D*'s org.foo.impl import, the framework has a choice of either module *A* or *B*, which is non-deterministic choice. Any ordering of resolution for these modules will lead to the same result.

6.4 Implementation Package Grouping

Grouping is useful when a collection of packages in a bundle must be used in conjunction with each other and may not be mixed with packages from elsewhere. For example, an ApacheCommons module could specify some exports in a single grouping of the form:

```
Export-Package: org.apache.commons.logging; org.apache.commons.io;
                version=”2.0.0”; grouping:=”commons”;
                implementation-name="Commons”
```

Then a module *A* needing to use specific versions of all the commons packages could specify for example:

```
Import-Package: org.apache.commons.logging; org.apache.commons.io;
                version=”2.0.0”;
                implementation-name="Commons"
```

On the other hand, a module *B* requiring a specific version of the I/O package could specify for example:

```
Import-Package: org.apache.commons.logging;
                    version=”2.0.0”;
                    implementation-name="Commons",
                org.apache.commons.io;
                    version=”[2.0.0,2.0.0]”;
                    implementation-name="Commons”
```

For the purpose of this example, suppose there is another module (ApacheCommons’) which contains exports of the form:

```
Export-Package: org.apache.commons.logging; org.apache.commons.io;
                version=”2.1.0”; grouping:=”commons”
                implementation-name="Commons"
```

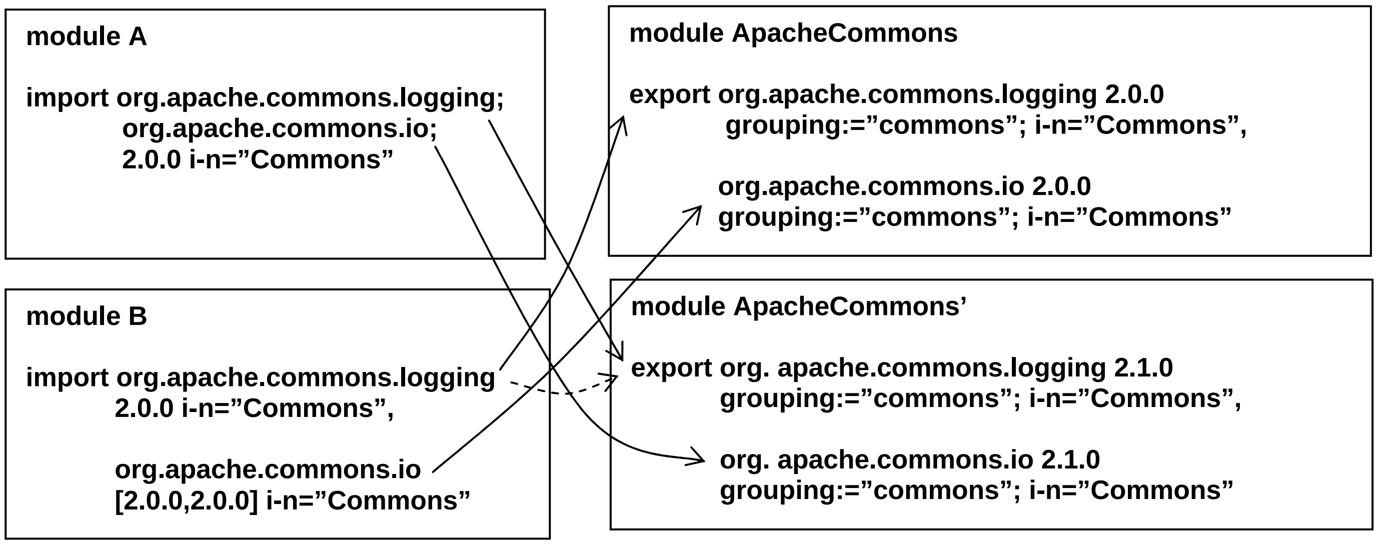

module A
import org.apache.commons.logging;
org.apache.commons.io;
2.0.0 i-n="Commons"
module B
import org.apache.commons.logging
2.0.0 i-n="Commons",
org.apache.commons.io
[2.0.0,2.0.0] i-n="Commons"
module ApacheCommons
export org.apache.commons.logging 2.0.0
grouping:="commons"; i-n="Commons",
org.apache.commons.io 2.0.0
grouping:="commons"; i-n="Commons"
module ApacheCommons'
export org. apache.commons.logging 2.1.0
grouping:="commons"; i-n="Commons",
org. apache.commons.io 2.1.0
grouping:="commons"; i-n="Commons"

Figure 9: Implementation Package Grouping

Assume that the framework has not resolved any modules yet. To resolve module *A*, the framework finds all providers that match the import constraint. Both ApacheCommons and ApacheCommons' are candidates, so the framework selects the higher version.

To resolve module *B*, the framework first of all tries to resolve the logging import. Both ApacheCommons and ApacheCommons' are candidates, so the framework selects the higher version. Next, the framework tries to resolve the I/O import. This time only ApacheCommons is a candidate, but the grouping of logging and I/O means that resolution cannot succeed with module *B* importing logging from ApacheCommons' and I/O from ApacheCommons. The framework then has to backtrack and select the other candidate, ApacheCommons, for logging. It then finds that the only candidate for I/O satisfies the grouping of logging and I/O and so both are successfully resolved to ApacheCommons.

6.5 Mixed OSGi Example with Specific Version Constraint

No longer supported. See section 8.16.

6.6 Implementation Sharing with Consistency Checking

To better support implementation sharing, RFC 79 extends module meta-data over OSGi R3 to include the concept of publicly importing a package to improve consistency checking using reachability analysis. The dependency metadata capture situations where a module's imports are visible via the module's exports. Such a situation effectively propagates the module's imports to any other modules that wish to use its exports. As an example, consider a [semi-] real-world example of XML parsers.

Assume that an XML parser is packaged into three separate modules: one for SAX, one for DOM, and one for JAXP. Now consider the fact that the package javax.xml.parsers contains the classes DocumentBuilder and SAXParser. DocumentBuilder has a method, called newDocument(), that return an instance of type org.w3c.dom.Document. SAXParser has a method, called getParser(), that returns an instance of type org.xml.sax.Parser. From this information, it is clear that any importer of javax.xml.parsers is also required to import org.w3c.dom and org.xml.sax because these packages are visible from classes in JAXP's exported package. To denote this situation, the JAXP module does not use import declarations for the two external packages, instead it uses dependency declarations. This lets the framework know that JAXP is importing packages that are visible via its exports. The following meta-data depicts a similar use case:

Module SAX:

```
Export-Package: org.xml.sax; version=“1.3.0”
```

Module DOM A:

```
Export-Package: org.w3c.dom; version=“2.1.0”
```

Module DOM B:

```
Export-Package: org.w3c.dom; version=“2.2.0”
```

Module JAXP:

```
Export-Package: javax.xml.parsers; version=“1.1.0”; uses:=”org.w3c.dom,org.xml.sax”
Import-Package: org.w3c.dom; version=“[2.1.0, 2.1.0]”,
                org.xml.sax; version=“[1.3.0, 1.3.0]”
```

Module Client A:

```
Import-Package: org.xml.sax; version=“[1.3.0, 1.3.0]”,
                org.w3c.dom; version=“[2.1.0, 2.1.0]”,
                javax.xml.parsers; version=“[1.1.0, 1.1.0]”
```

Module Client B:

```
Import-Package: org.xml.sax; version=“[1.3.0, 1.3.0]”,
                org.w3c.dom; version=“[2.2.0, 2.2.0]”,
                javax.xml.parsers; version=“[1.1.0, 1.1.0]”
```

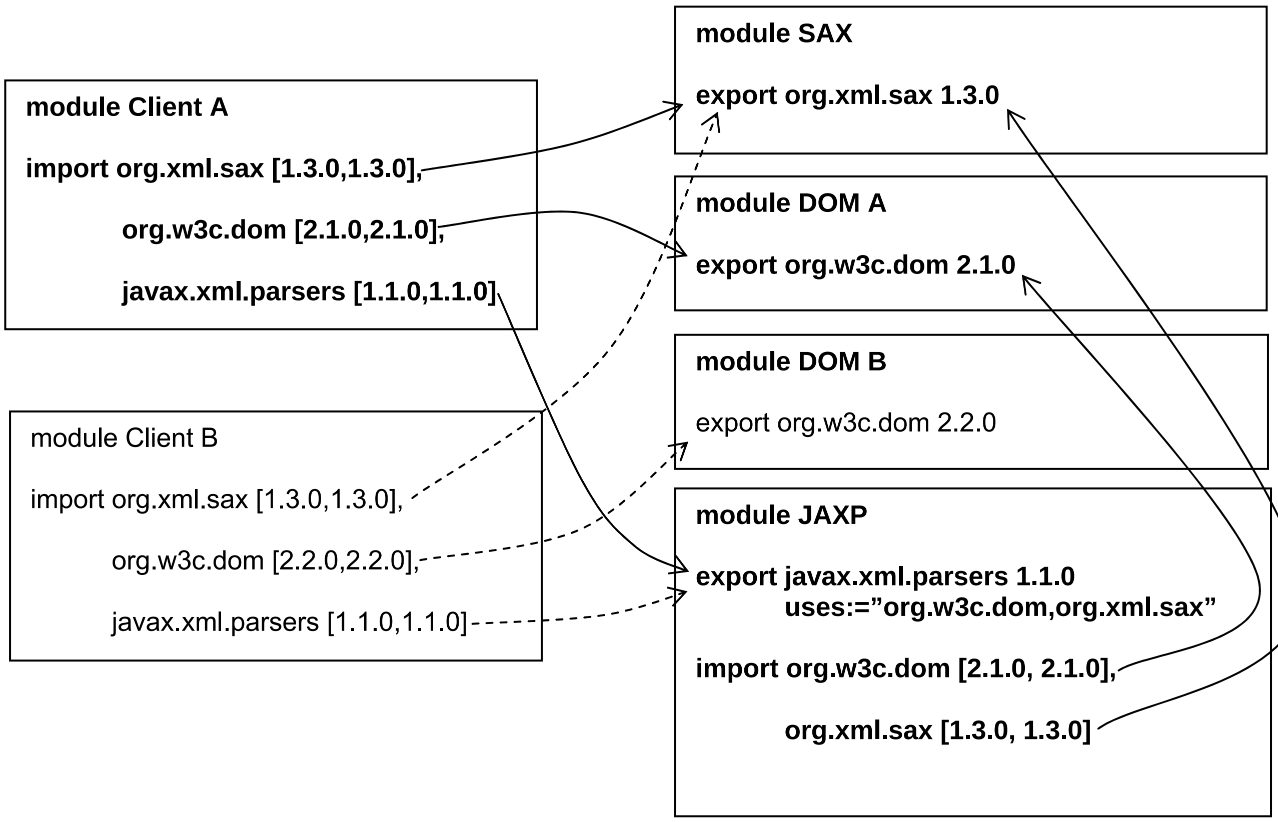

module SAX
export org.xml.sax 1.3.0
module Client A
import org.xml.sax [1.3.0,1.3.0],
org.w3c.dom [2.1.0,2.1.0],
javax.xml.parsers [1.1.0,1.1.0]
module DOM A
export org.w3c.dom 2.1.0
module DOM B
export org.w3c.dom 2.2.0
module Client B
import org.xml.sax [1.3.0,1.3.0],
org.w3c.dom [2.2.0,2.2.0],
javax.xml.parsers [1.1.0,1.1.0]
module JAXP
export javax.xml.parsers 1.1.0
uses:="org.w3c.dom,org.xml.sax"
import org.w3c.dom [2.1.0, 2.1.0],
org.xml.sax [1.3.0, 1.3.0]

Figure 10: Implementation Sharing with Consistency Checking

With these set of modules, the framework would be able to resolve all of them except *Client B*. This is the case because *Client B*'s import of javax.xml.parsers and the dependency on org.w3c.dom expressed in the JAXP module conflicts with the version of org.w3c.dom imported by *Client B*. If a different provider of javax.xml.parsers were introduced that did not have conflicting exports, then *Client B* could resolve to it instead.

6.7 Explicit Dependencies

Consider a variant of the example in section 4.2.5 (on page 10). Suppose module A exports package P and specifies a dependency on the imported package Q. Suppose A also exports package R with no dependencies. Then the module C in Figure 11 cannot be resolved since C cannot import the same Q as A, whereas D can resolve since R does not depend on Q.

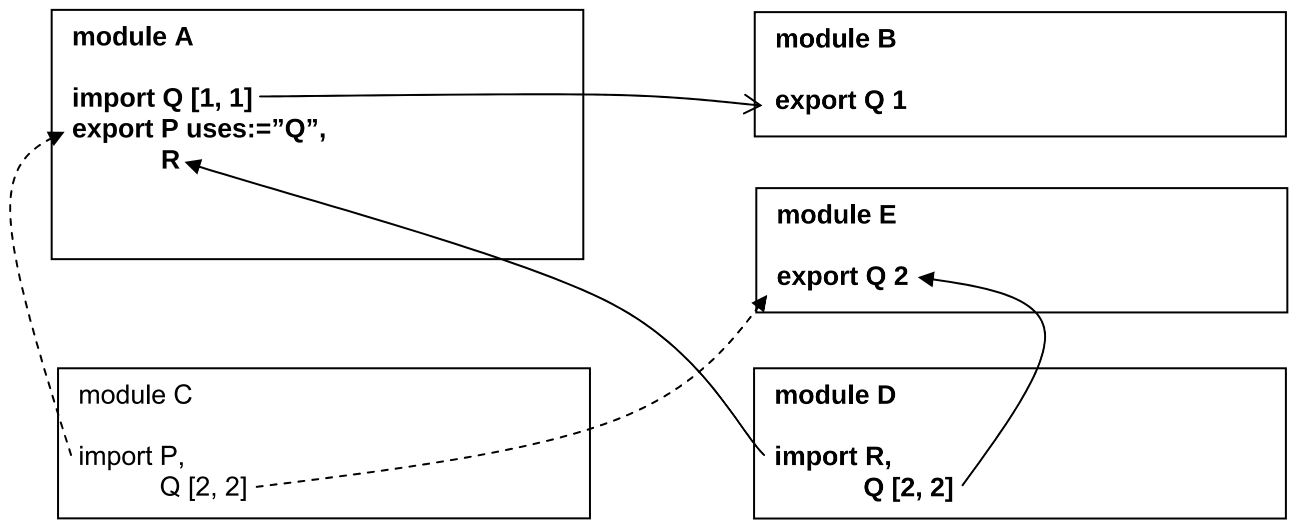


Figure 11: Explicit Dependencies

6.8 Split Packages and Module Composition

Split packages can occur when a module containing a large package is divided into two or more separate pieces for ease of maintenance. Importers of the module can be protected from being impacted by the change.

Suppose 'client' modules Ci import package P using some matching attribute as shown in Figure 12.

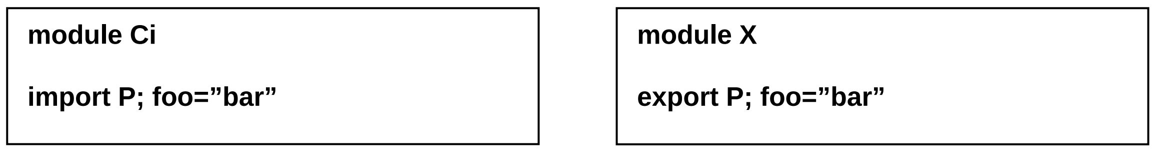


Figure 12: Before Package Split

Now suppose X is divides into modules Y and Z which each export a piece of P. X can be modified to gather together the pieces of P and re-export the package so that the clients Ci can import the package with having to be changed as shown in Figure 13. X is acting as a façade which protects the clients from being impacted by the change.

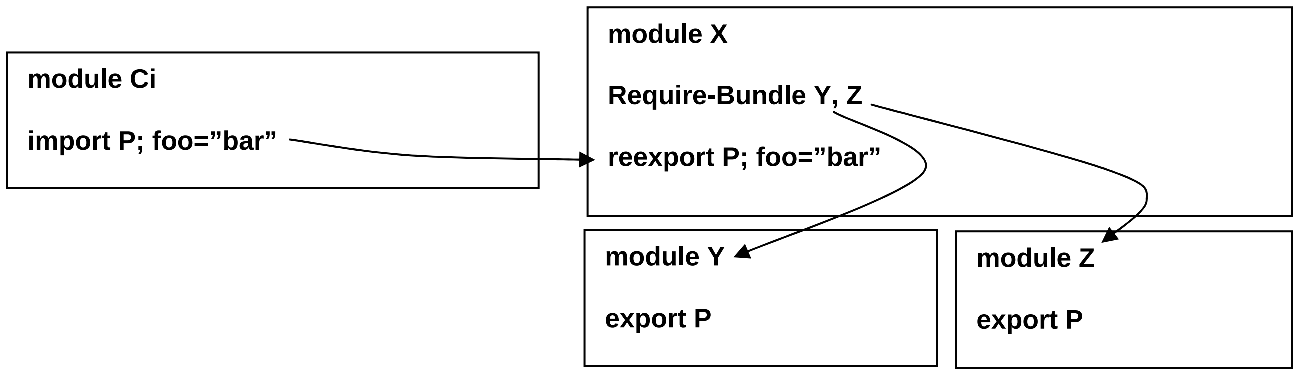


Figure 13: Split Package's Modules Composed into a Façade

6.9 Export Filtering

Suppose it is required to modularize an existing piece of code without changing it because the code is controlled by an external body such as an open source group. Suppose the code contains a package P which contains both interface and implementation classes and that there is a desire to hide the implementation classes inside a module M. Depending on the numbers of classes involved, either the implementations classes could be excluded from the export of P, thus:

```
  export P; exclude:="impl1,impl2"
```

or the interface classes could be included thus:

```
  export P; include:="interface1,interface2"
```

It is even possible to export the same package multiple times with distinct filters. For example, the following export statement would export a subset of the package P to arbitrary importers but the whole of P to importers that specify the arbitrary matching attribute comp="foo":

```
  export P; include:="interface1,interface2",
         P; mandatory:="comp"; comp="foo"
```

7 Security Considerations

7.1 PackagePermission

The existing R3 PackagePermission mechanism is acceptable for the initial release of the changes proposed in this document. Although this document allows for the possibility that multiple versions of the same package may exist and be shared within the framework at the same time, there is no obvious approach for adding explicit security support for this capability. Until real-world feedback is available, the current all-or-nothing approach to package permissions is sufficient. This means that there is no way to directly define a policy that allows a given bundle to export only a limited version range or set of attributes and, likewise, to define a policy that allows a given bundle to import only a limited version range or set of attributes. To implement such policies, the deployment agent is responsible for analyzing bundle meta-data to ensure that it does not violate fine-grained export/import constraints before granting a bundle package permission.

7.2 BundlePermission

Frameworks may support BundlePermission but are not required to do so. A framework which supports BundlePermission must also support Require-Bundle and Reexport-Package.

Previous releases of the OSGi framework supported a single type of dependency among bundles, a *package dependency*. The require-bundle construct proposed in this document introduces a new kind of dependency, a *bundle dependency*. In securing a package dependency, PackagePermission controls which bundles may participate as importers or exporters. In a similar fashion, it is necessary to control which bundles may participate in a bundle dependency; this is the purpose of BundlePermission. A BundlePermission is parameterized by an action and a bundle symbolic name, where the action is defined as one of the following:

- *PROVIDE* – grants the bundle the right to use the specified bundle symbolic name and, consequently, participate as a bundle provider (this in analogous to the *EXPORT* action of PackagePermission).
- *REQUIRE* – grants the bundle the right to require the bundle associated with the specified bundle symbolic name and, ultimately, access all of its exports (this in analogous to the *IMPORT* action of PackagePermission).

All bundles with a bundle symbolic name must be granted permission to provide their own name. A bundle must have explicit permission to require any other bundle.

8 Considered Alternatives

8.1 Requirement to override version constraints

Since import version constraints are specified during development of a module, these constraints may later turn out to be too restrictive. For example, a module which satisfies such constraints may subsequently evolve in a backward-compatible manner. So there needs to be a way to specify that an exporter supports a previous version of an exported package and, more generally, for the “system” to override the constraints expressed by a module. Something analogous to .NET’s configuration files which allow overrides of assembly metadata to be specified by the ‘application’, the ‘provider’, and the ‘system’ seems to be necessary. Requirements for version overrides are also described in RFP 44.

Since it is anticipated that the above requirements will be addressed in a separate RFC, they are not tacked in this RFC.

8.2 Export Version Ranges

The option of a version range being associated with a package export as a way of asserting full semantic backward compatibility with earlier versions was considered, but thought to be impractical. Rarely can a module developer make such guarantees. It is more likely that the module developer would document circumstances in which a package is likely to behave compatibly with an earlier version, in which case a deployer may opt to override (as described in section 8.1) import version constraints.

8.3 Syntactic Sugar

Supporting old syntax using syntactic sugar was rejected in favor of keeping the old and new syntax separate by introducing bundle manifest version.

8.4 Import Filters

Filter expressions were considered for the Import-Package attribute matching syntax, but the marginal additional function did not appear to justify the additional complexity.

8.5 Avoidance of Singleton Packages

A singleton package is such that at most one resolved bundle is allowed to export any version of the package. This was thought to be unnecessary given that the rules for matching imports to existing exports and later versions of exported packages seemed to be sufficient to ensure uniqueness of exported service interfaces. In general, singleton packages appeared to be heavy-handed for their anticipated uses. Also, since RFC 70's singleton bundles would still be required even if singleton packages were supported, two kinds of singleton could be confusing.

8.6 Package Propagation Variants

It would be possible to distinguish between exposed imports and explicit propagation as the former does not necessitate exporting the exposed packages. However, by extending the concept of grouping to include propagated packages, this distinction was avoided.

Also the grouping directive of Import-Package is related to exporting rather than importing. It would be nice to keep the import statement free from export-related attributes. The difficulty in doing this would be how the export statement would identify the corresponding import statement, especially if wild-carded package names are also considered.

8.7 Fragment Restrictions

A proposal was considered to have a host bundle declare which of its packages could be extended and/or shadowed by fragments in order to enable the host to control the impact of fragments on its behavior. But since fragments are often used for adding national language resources to arbitrary packages, the degree of control would be relatively weak in practice. A restriction on when fragments could be resolved combined with a switch for hosts to prevent fragments from attaching to them was felt to be a more workable solution.

8.8 Default Grouping

An alternative default would be for exports to be grouped by default, either across the whole of a module (or across semicolon-separated subsets of the exported packages), but this would require unique grouping names to specified for 'ungrouped' exports.

8.9 Friendship

The ability for an export to stipulate the bundle symbolic names of the modules that may validly import the exported packages was brittle and less general than allowing the exporter to specify that certain attributes must be specified by importers. For instance, adding a new module to a group of friends would require each of the friend's metadata to be modified. There also appeared to be some overlap with security.

8.10 Separate Manifest Header for Export Filtering

Separating class load (or resource find) time export filtering into a manifest header separate from Import-Package would prevent modules from exporting a package multiple times with distinct filters, such as to enable its 'friends' to access classes which would be filtered out for imports by other modules.

8.11 Optional Support for RFC 70 Syntax

We considered allowing OSGi R4 implementations to support R3 bundles which use the experimental syntax defined by draft 8 of RFC 70. This was not included since it is only of interest to Eclipse.

The following table lists replacements for the experimental syntax defined by draft 8 of RFC 70. Note that the experimental syntax is not part of the OSGi specification.

Experimental RFC 70 Syntax	Equivalent Bundle Manifest Version 2 Syntax
`Provide-Package: <p1>, <p2>, …, <pn>`	`Export-Package: <p1>; <p2>; …; <pn>; uses:=" <p1>,<p2>,…,<pn>"`
`Require-Bundle: <b>`	`Require-Bundle: <b>`
`Require-Bundle: <b>; bundle-version=<bv>`	`Require-Bundle: <b>; bundle-version=<bv>`
`Require-Bundle: <b>; reprovide="true"`	`Require-Bundle: <b>; visibility:="reexport"`
`Require-Bundle: <b>; optional="true"`	`Require-Bundle: <b>; resolution:="optional"`
`Require-Bundle: <b>; require-packages="<p1>, <p2>, …, <pn>"`	`No equivalent`
`Require-Bundle: <b>; optional="true" require-packages="<p1>, <p2>, …, <pn>"`	`No equivalent.`

The last two rows of the table are greyed out since the experimental RFC 70 syntax was not implemented by Eclipse and so does not need an equivalent.

8.12 Static Import of Wild Carded Package Names

A general problem with allowing wild carded package names on (static) imports is that it is not possible when resolving a module to know that all the packages it needs are available. For instance, if a module imports "org.foo.*" but really needs org.foo.a and org.foo.b, the resolution process cannot tell.

This problem could be reduced in certain special cases by the importer supplying the exporter's bundle symbolic name and bundle version, but even in that case, there is no guarantee that the exporter will export the required packages.

There is also the difficulty of giving wild cards reasonable semantics. The natural interpretation would be to import any packages which match the wild carded name. If no packages match the wild carded name, then this would not be an error, but might not be what the user expected.

Finally, by avoiding wild cards, the resolver can limit the scope of its search by package before considering other constraints.

8.13 Framework packages

Framework-related packages, such as org.osgi.framework, are somewhat interesting cases, since in R3 these packages were provided by the system bundle, but it was assumed in R3 that only one version of these packages would ever be available. The changes proposed in this document make it possible for other bundles to offer different versions of these framework-related packages. One possibility is to define framework-related packages as part of the execution environment. However, exporting them from the system bundle leaves open the possibility of multiple versions, which may be useful in some situations for providing backwards compatibility.

8.14 Renaming of 'Bundle' Concepts

The renamings in the table below were considered in order to separate out the modularity support of the framework more clearly. However, bundle symbolic name is probably used elsewhere in R4 to refer to a bundle rather than a module, so such a renaming could cause problems. Also, there does not appear to be a need in OSGi to differentiate between a module and its bundle. There are no known requirements for multiple modules per bundle.

Term	**Alternative considered**	**Impacts would have included**
bundle-symbolic-name	module	Bundle-SymbolicName, Import-Package, DynamicImport-Package, Export-Package, Reexport-Package manifest headers. BundlePermission.
bundle-version	module-version	Bundle-Version, Import-Package, DynamicImport-Package, Export-Package, Reexport-Package, Require-Bundle, Fragment-Host manifest headers.
Require-Bundle	Require-Module	Require-Bundle manifest header.
"system.bundle"	"system.module"	References to system bundle.
"bundle:" URL protocol	"module:" URL protocol	Reference to "bundle:" protocol.

8.15 Pure Bundle Dependencies

Frameworks that do not implement Require-Bundle may still need to express resolution dependencies between bundles. An alternative considered was to make some variant of Require-Bundle part of the mandatory subset of this RFC. However, the existing features of this RFC are sufficient if a convention is adopted. For instance, bundles could export some package with name based on the bundle symbolic name, e.g. "bundlename.present" which could then be imported by a bundle which wanted to ensure that the other bundle was resolved.

8.16 Special Treatment of Consequent Resolutions

When a module imports and exports the same package, an alternative was considered to favour the export when the bundle was being resolved as a consequence of an import in some other bundle. However, this would lead to unpredictable behaviour since a bundle could resolve differently based purely on the order in which bundles are resolved.

The following example shows the difference in behaviour under this alternative. Without this alternative, bundle C fails to resolve.

In OSGi R3, it was not possible for a bundle to require a specific specification package version; with the more general approach of RFC 79, this is now possible. It is important to note, however, that mixing OSGi R3 use cases with the more generic features of RFC 79 will lead to results that are potentially inconsistent with previous OSGi specifications. The important issue to remember is that these features are not intended to be used in mixed-mode for OSGi, but since the underlying module support is generic, the combinations are possible. The following meta-data depicts a simple use case:

Meta-data for module *A*:

```
Export-Package: javax.servlet; version=“2.1.0”
Import-Package: javax.servlet; version=“2.1.0”
```

Meta-data for module *B*:

```
Export-Package: javax.servlet; version=“2.2.0”
Import-Package: javax.servlet; version=“2.2.0”
```

Meta-data for module *C*:

```
Import-Package: javax.servlet; version=“[2.1.0, 2.1.0]”
```

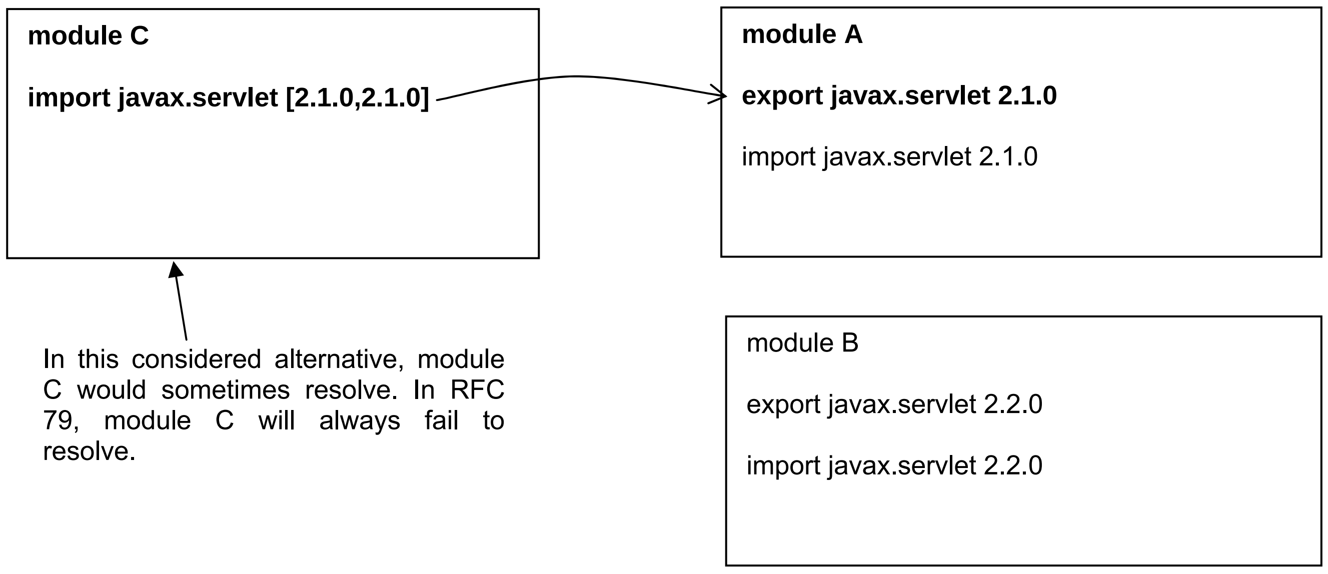

module C
import javax.servlet [2.1.0,2.1.0]
module A
export javax.servlet 2.1.0
import javax.servlet 2.1.0
module B
export javax.servlet 2.2.0
import javax.servlet 2.2.0
In this considered alternative, module C would sometimes resolve. In RFC 79, module C will always fail to resolve.

Figure 14: Mixed OSGi Example with Specific Version Constraint

Assume for the above example meta-data that the framework has not resolved any modules yet. If module *C* is the first module resolved, the framework finds all available candidate providers and only finds module *A*. Consequently, it selects module *A* and starts to resolve it, passing in the targeted package of javax.servlet. In the consequent resolution of module *A*, its corresponding import of javax.servlet is ignored, since it is the passed in target package. The framework determines that this package is not resolved in the global space, thus module *A* is free to export it. Thus, the framework did not select the greatest available version of javax.servlet. Subsequently, module *C*'s import is bound to module *A*.

After resolving module *C*, module *B* may also be successfully resolved. If modules *A* or *B* were resolved first, then they would resolve successfully but it would then be impossible to resolve module *C*.

8.17 Security

Before reverting to R3 security for package plus bundle permission for Require-Bundle, the security section contained the following ideas. This was deferred to a future RFC as the requirements were not clear.

In R3, `PackagePermission` is used to control whether or not a bundle is allowed to import or export a given package. This approach can still be used for the changes proposed in this document with some slight extensions to make it more fine grained. The current approach only allows for all-or-nothing scenarios when importing or exporting a package, but other scenarios might be useful. For example, it may be permissible for a given bundle to export a particular package, but only if it does not attach reserved attributes to the export. This might be necessary if a developer wants to guarantee that an import is satisfied by a particular bundle using particular attributes; if other bundles are allowed to attach the same attributes to their exports, then it is not possible to guarantee from where an import will come.

To provide this flexibility, `PackagePermission` must be extended to not only accept an action and package name, but also a list of allowable attribute name-value pairs for mandatory attributes. For an export permission, the list of attribute name-value pairs indicates which mandatory attributes and values the bundle can attach to the specified package. For an import declaration, the list of attribute name-value pairs indicates which mandatory attributes and values the bundle is allowed to read. It is possible to grant permission to an attribute with the value of “*” to indicate that the bundle can attach or read any value. By default, a bundle does not have permission to attach or read any mandatory attributes. Note that a bundle needs permission to read all mandatory attributes that are attached to a corresponding export statement during the resolve process, even if the import statements does not explicitly refer to all of the attributes.

With these permissions in place, a bundle is only allowed to import or export a package if the package name and any supplied mandatory attribute name-value pairs correspond to a given permission declaration. Assuming that no attributes are attached to import or export declarations, then the permission works exactly as it did in R3. Attribute matching is then only performed for those bundles needing mandatory attributes. With the wild-card value, this approach allows flexibility, but with precise values it also enables specific bundles to be given permission for precise mandatory attributes, such as those that might uniquely identify the bundle and should not be duplicated by other bundles.

Consider the following simple scenario:

```
Bundle A:
Bundle-SymbolicName: A
Export-Package: org.foo; version=“1.0.0”

Bundle B:
Bundle-SymbolicName: B
Import-Package: org.foo; version=“1.0.0”
```

In this scenario, bundle *A* requires `PackagePermission[EXPORT, "org.foo", null]` and bundle *B* requires `PackagePermission[IMPORT, "org.foo", null]`, which is equivalent to R3. The following scenario addresses a mandatory `Bundle-SymbolicName` attribute:

```
Bundle A:
Bundle-SymbolicName: A
Export-Package: org.foo; version="1.0.0"; mandatory:="bundle-symbolic-name"
Bundle B:
Bundle-SymbolicName: B
Import-Package: org.foo; bundle-symbolic-name="A"
```

In this scenario, bundle *A* requires

```
PackagePermission[EXPORT, "org.foo", "bundle-symbolic-name=A"]
```

because it wants to attach a mandatory attribute and bundle *B* requires

```
PackagePermission[IMPORT, "org.foo", "bundle-symbolic-name=A"]
```

in order to import from bundle *A*. The package permission granted to bundle *A* specifically allows it to attach a mandatory bundle symbolic name of `A` to its `org.foo` export. The package permission granted to bundle *B* specifically allows it to read the mandatory bundle symbolic name of A, which means that bundle *B* can only import `org.foo` from bundle *A*. To allow bundle *B* to import `org.foo` from any bundle symbolic name, it should be granted

```
PackagePermission[IMPORT, "org.foo", "bundle-symbolic-name=*"]
```

The following example addresses multiple export attributes:

```
Bundle A:
Bundle-SymbolicName: A
Export-Package: org.foo; version="1.0.0"; arbitrary="value"
mandatory:="bundle-symbolic-name,arbitrary"

Bundle B:

Bundle-SymbolicName: B
Import-Package: org.foo; bundle-symbolic-name="A"
```

In this scenario, bundle *A* requires

```
PackagePermission[EXPORT, "org.foo", "bundle-symbolic-name=A,arbitrary=value"]
```

so that it can attach both of its mandatory attributes. Bundle *B* requires

```
PackagePermission[IMPORT, "org.foo", "bundle-symbolic-name=A,arbitrary=value"]
```

in order to import from bundle *A*., even though its import declaration does not reference the attribute `arbitrary`. This is necessary because the attribute is mandatory and as such an importing bundle must have permission to read the value no matter what, otherwise it would not be possible to guarantee strict import/export resolutions.

From the export perspective, this approach is nearly identical to R3 in that it grants export permission to a bundle that can be independently verified. From the import perspective, this approach is a little different to R3 in that a granted import permission can no longer be independently verified; instead, an import permission must be verified against a target export declaration. This means that a given import permission may fail against one import, but succeed against another. For example.

```
Bundle A:
Bundle-SymbolicName: A
Export-Package: org.foo; version="1.0.0"; mandatory:="bundle-symbolic-name"
Bundle B:
Bundle-SymbolicName: B
Import-Package: org.foo; bundle-symbolic-name="A"

Bundle C:
Bundle-SymbolicName: C
Export-Package: org.foo; version="1.0.0"
```

Assume that the exporting bundles in the above example are granted the proper export permission. If the bundle *B* is granted

```
PackagePermission[IMPORT, "org.foo", null]
```

it will fail when verified against bundle *A*, but it will succeed when verified against bundle *C*.

As these examples illustrate, this approach allows the framework to guarantee that only certain exporters can attach certain mandatory attributes and that only certain importers can access those exports. In the simplest case, it is identical to R3 and the complexity for the user is only visible when it is needed.

8.18 Grouping

Before dependencies were expressed with "uses", the alternative was to specify grouping names on imports and exports. Dependencies were felt to capture the dependencies (signature or implementation) more faithfully and would avoid the need to invent grouping names.

8.19 Module Resolution Prescription

Earlier versions of the RFC attempted to prescribe how the resolver should work in addition to what specified post-conditions it should satisfy. This was felt to limit innovation in resolver algorithms and the freedom of framework implementations to satisfy requirements specific to their environment.

Even prescribing what happens when there is a free choice in wiring an import to more than one export was felt to be overly constraining. Some frameworks may wish to minimize partitioning of the service registry whereas others may prefer always to favour newer versions of bundles. Some frameworks may opt for more reproducible behaviour whereas others may prefer to optimize according to other requirements.

9 Document Support

9.1 References

[1]. Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC2119, March 1997.

[2]. Michael Jackson, "Software Requirements & Specifications", ISBN 0-201-87712-0.

[3]. John Corwin, David F. Bacon, David Grove, Chet Murthy, "MJ: a rational module system for Java and its applications", OOPSLA '03 Proceedings, http://pag.csail.mit.edu/reading-group/corwin03mj.pdf.

[4]. Richard Hall, "A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks", 2nd International Working Conference on Component Deployment, Edinburgh, May 2004.

[5]. RFP 46 "OSGI on Embedded Systems", http://membercvs.osgi.org/rfps/rfp-0046-embedded.doc

[6]. RFP 44 "Framework Next Generation", http://membercvs.osgi.org/rfps/rfp-0044-FrameworkNG.doc

[7]. RFC 70 "Bundle Dependency and Class Loading Changes", http://membercvs.osgi.org/rfcs/rfc0070

[8]. Interval Notation, http://www.math.ohio-state.edu/courses/math104/interval.pdf

[9]. RFC 73 "Permission Update", http://membercvs.osgi.org/rfcs/rfc0073

9.2 Authors' Addresses

Name	Glyn Normington
Company	IBM
Address	IBM United Kingdom Limited, Hursley Park, Winchester, Hampshire, SO21 2JN, ENGLAND
Voice	+44-(0)1962-815826
e-mail	glyn_normington@uk.ibm.com

Name	Richard S Hall
Company	Invited Researcher
Address	220 Rue de la Chimie, Domain Universitaire, BP 53, 38041 Grenoble Cedex 9, FRANCE
Voice	+33-476-635-574
e-mail	heavy@ungoverned.org

9.3 Acronyms and Abbreviations

None.

9.4 End of Document