Mobiliser Platform – Foundation

Introduction to OSGi

OSGi

What it is and Why it is Needed?

If OSGi is the Solution, Then What's Problem? 1/2

Before any description of OSGi will make sense, we must first understand what issues are currently faced by developers when creating large-scale applications in Java.

If OSGi is the Solution, Then What's Problem? 1/2

Before any description of OSGi will make sense, we must first understand what issues are currently faced by developers when creating large-scale applications in Java.

When Sun Microsystems originally designed the Java language, they were designing a language to control the embedded processors found in televisions.

If OSGi is the Solution, Then What's Problem? 1/2

Before any description of OSGi will make sense, we must first understand what issues are currently faced by developers when creating large-scale applications in Java.

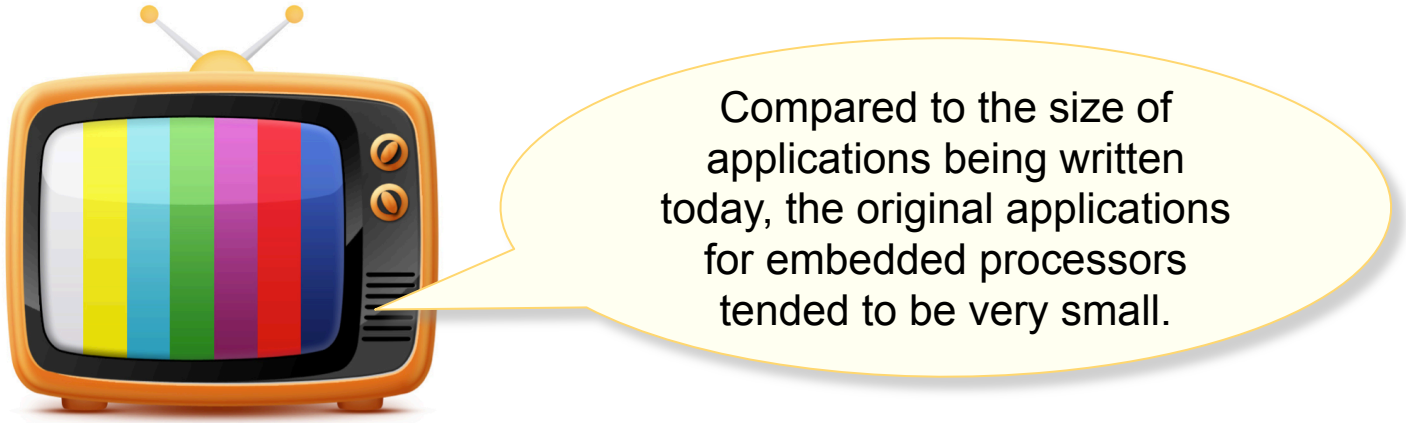
When Sun Microsystems originally designed the Java language, they were designing a language to control the embedded processors found in televisions.

Compared to the size of applications being written today, the original applications for embedded processors tended to be very small.

If OSGi is the Solution, Then What's Problem? 1/2

Before any description of OSGi will make sense, we must first understand what issues are currently faced by developers when creating large-scale applications in Java.

When Sun Microsystems originally designed the Java language, they were designing a language to control the embedded processors found in televisions.



As a result, no attempt was made to create a unit of modularity beyond the Java ***package***, within which are contained the fine-grained, object oriented units called Java ***classes***.

This leads to an "***all or nothing***" concept of class visibility. Java classes are either publicly visible outside the scope of their package, or they are not.

If OSGi is the Solution, Then What's Problem? 2/2

As soon a Java was released in 1995 it began to be used to write applications related to another area of massive technological growth – the World Wide Web.

If OSGi is the Solution, Then What's Problem? 2/2

As soon a Java was released in 1995 it began to be used to write applications related to another area of massive technological growth – the World Wide Web.

The growth of the World Wide Web was the main driver behind the widespread adoption of the Java language in areas far removed from the original intended use case of an embedded processor.

If OSGi is the Solution, Then What's Problem? 2/2

As soon a Java was released in 1995 it began to be used to write applications related to another area of massive technological growth – the World Wide Web.

The growth of the World Wide Web was the main driver behind the widespread adoption of the Java language in areas far removed from the original intended use case of an embedded processor.

Consequently, the contemporary issues of developing maintainable, large-scale business applications in Java present us with challenges that the original language never considered.

If OSGi is the Solution, Then What's Problem? 2/2

As soon a Java was released in 1995 it began to be used to write applications related to another area of massive technological growth – the World Wide Web.

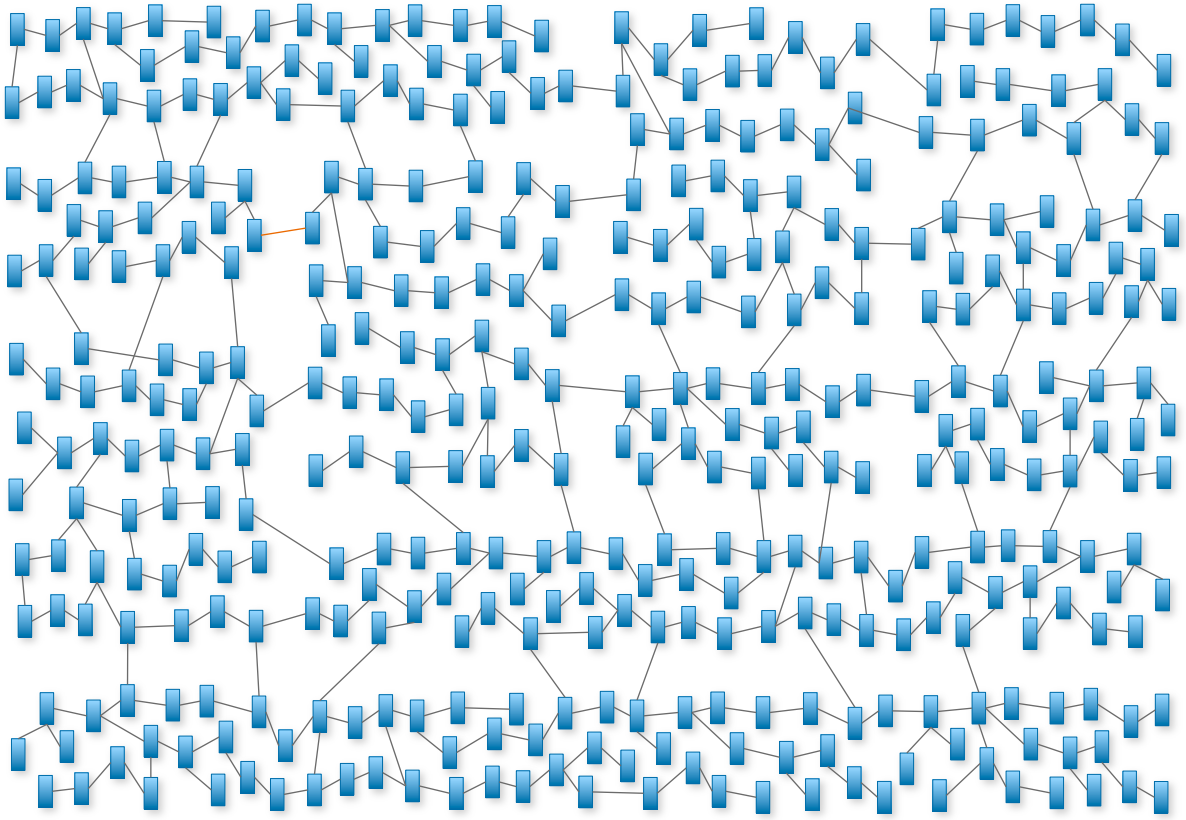
The growth of the World Wide Web was the main driver behind the widespread adoption of the Java language in areas far removed from the original intended use case of an embedded processor.

Consequently, the contemporary issues of developing maintainable, large-scale business applications in Java present us with challenges that the original language never considered.

So the heart of the problem is this:

How should the coding in a large-scale Java application be structured?
In other words, how do we solve the problem of ***MODULARITY****.*

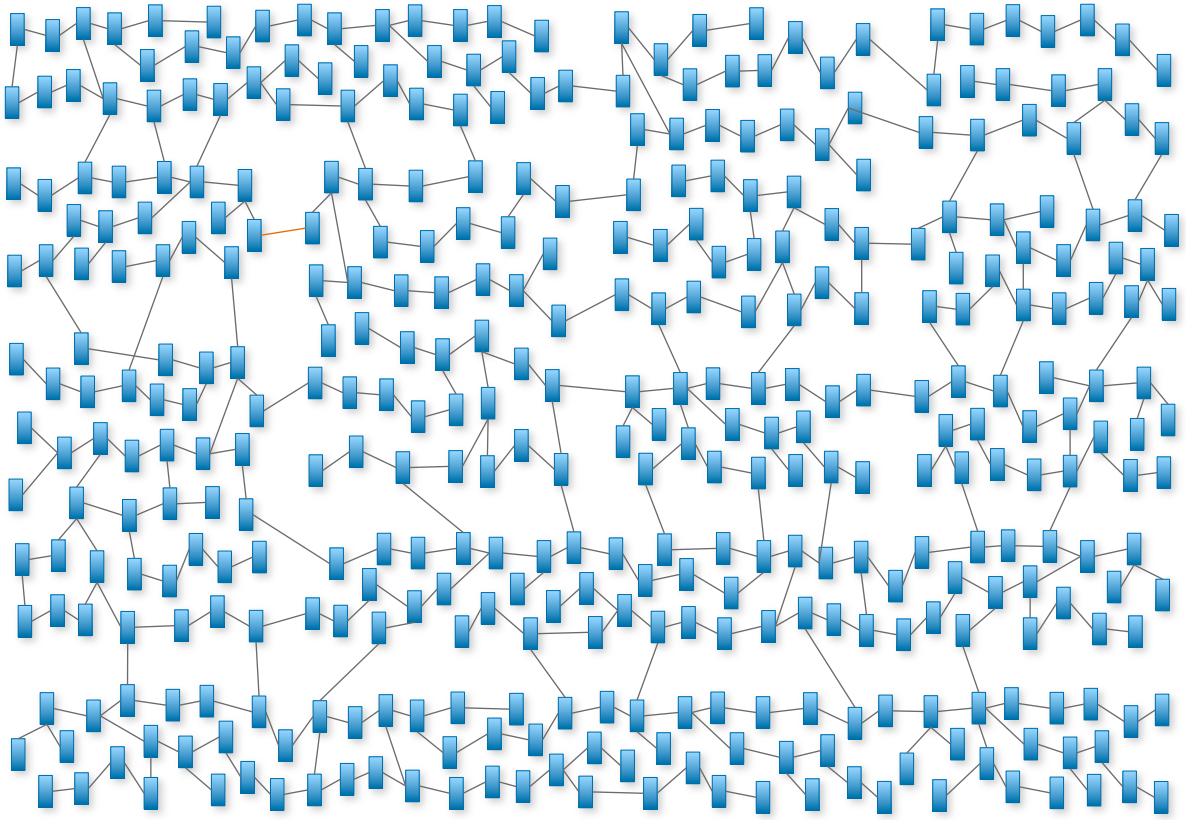
What's Wrong With Java's Modularity Concept? 1/2



Dependencies between classes in a large Java Application

There are several problems with Java's modularity concept:

What's Wrong With Java's Modularity Concept? 1/2

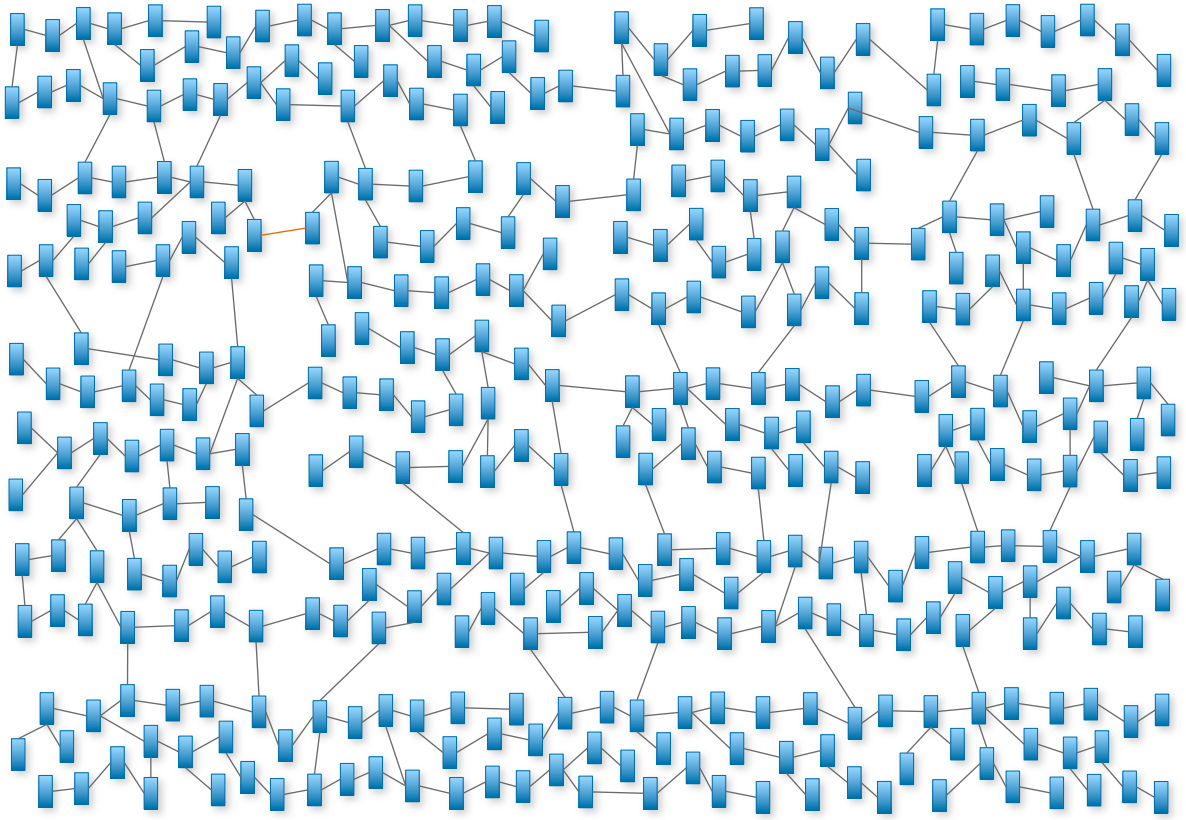


Dependencies between classes in a large Java Application

There are several problems with Java's modularity concept:

1. It focuses only on the technical objects needed to implement a particular solution, and not on the functionality of the solution itself. This results in a highly fragmented view of an overall software solution.

What's Wrong With Java's Modularity Concept? 1/2



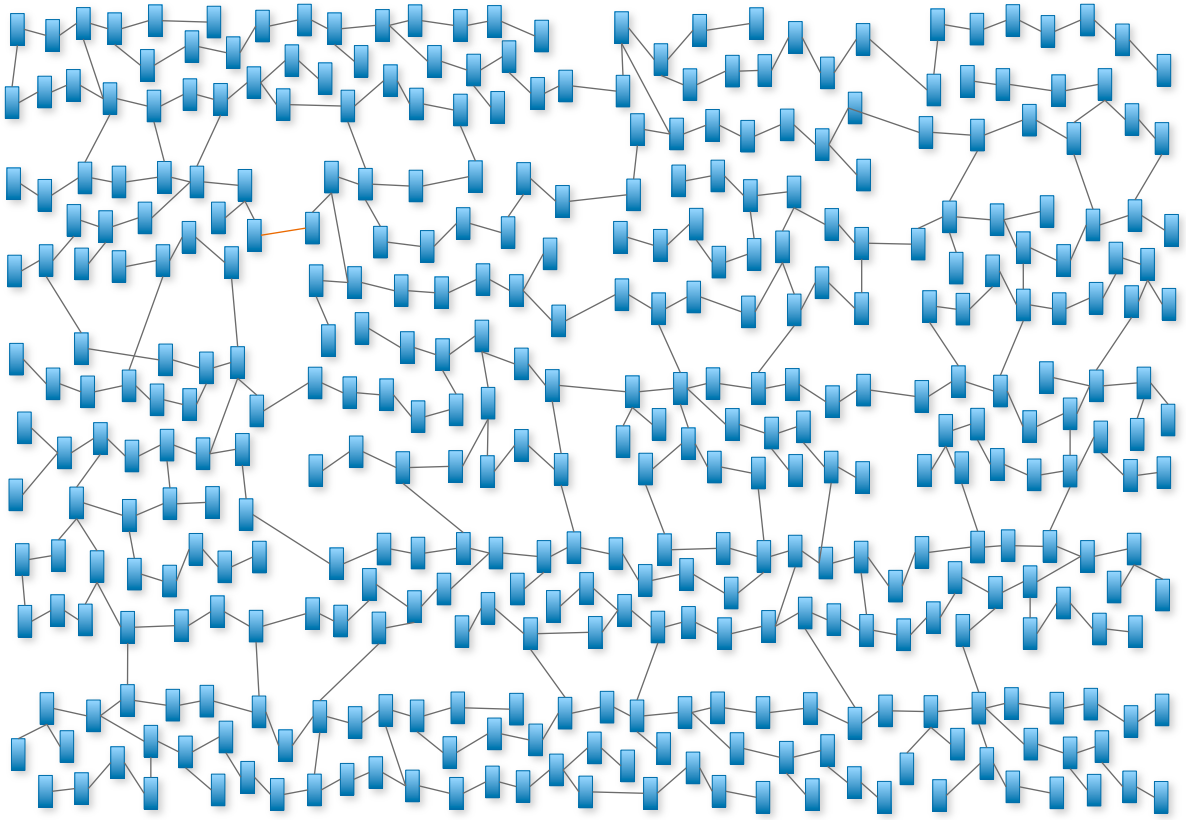
Dependencies between classes in a large Java Application

There are several problems with Java's modularity concept:

1. It focuses only on the technical objects needed to implement a particular solution, and not on the functionality of the solution itself. This results in a highly fragmented view of an overall software solution.
2. When class level visibility is combined with the Java package concept, the result is a crude level of visibility control that often forces developers to expose internal API details to the outside world.

   In other words, it does not allow for classes to be seen as public from within a unit of business functionality, but remain private when that unit of business functionality is used in the larger context of a business application.

What's Wrong With Java's Modularity Concept? 2/2

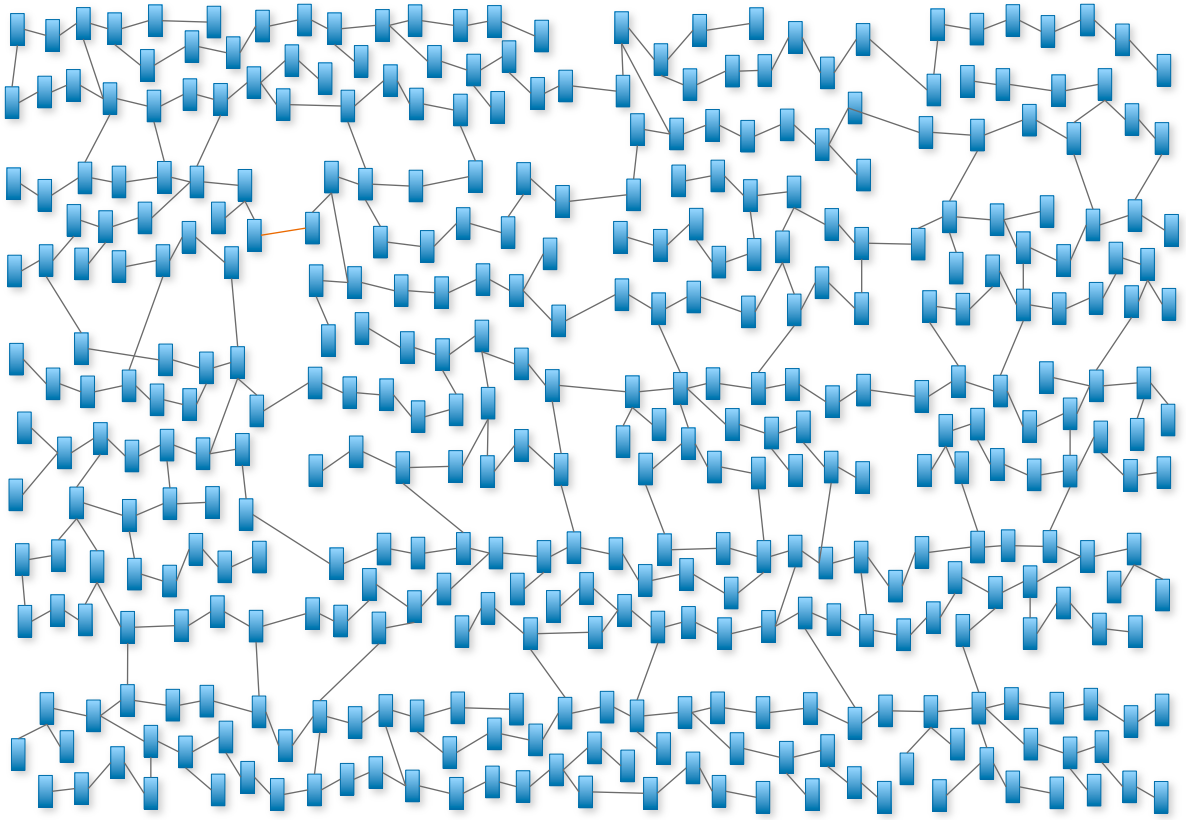


Dependencies between classes in a large Java Application

There are several problems with Java's modularity concept:

3. Java's `CLASS_PATH` concept hides a multitude of modularity gremlins:
   a) No concept of code versioning
   b) No mechanism for defining dependencies between packages
   c) Classes are supplied on a ***first-match-wins*** basis – possibly resulting in a `NoSuchMethodError`

What's Wrong With Java's Modularity Concept? 2/2

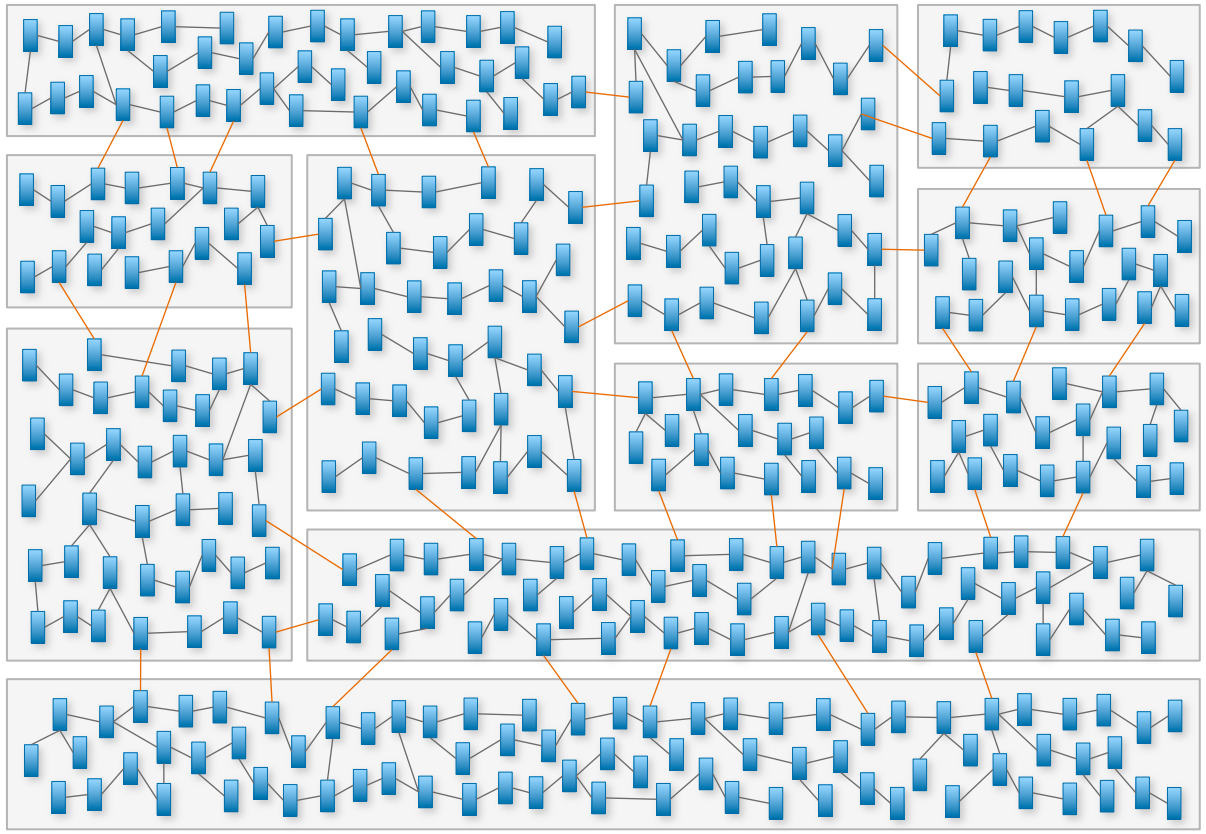


Dependencies between classes in a large Java Application

There are several problems with Java's modularity concept:

3. Java's `CLASS_PATH` concept hides a multitude of modularity gremlins:
   a) No concept of code versioning
   b) No mechanism for defining dependencies between packages
   c) Classes are supplied on a ***first-match-wins*** basis – possibly resulting in a `NoSuchMethodError`
4. No code deployment concept or mechanism

How Can Java's Modularity Concept be Fixed?



Dependencies between modules in an OSGi Java Application

A unit of modularisation is needed that allows the developer to:

- Create units of code that are reusable at a courser level than just the fine-grained concept of individual Java classes
- Make the overall structure of applications easier to design, manage and modify.

How Does OSGi Fill the Gaps in Java's Modularity Concept?

OSGi implements the following functionality that is missing from the standard Java language:

1. Uses the "module" concept to create a logical boundary around a set of Java packages

How Does OSGi Fill the Gaps in Java's Modularity Concept?

OSGi implements the following functionality that is missing from the standard Java language:

1. Uses the "module" concept to create a logical boundary around a set of Java packages
2. Modules are implemented as "bundles". Each bundle holds metadata that describes which Java packages are visible outside the scope of a bundle:
   - OSGi extends Java's concept of "`public`" such that any package within a bundle can be public in either a local sense (within the bundle) or a global sense (outside the bundle)
   - By default a bundle uses a ***shares nothing*** approach. Therefore, any package not explicitly "exported" remains hidden within the bundle's logical boundary

How Does OSGi Fill the Gaps in Java's Modularity Concept?

OSGi implements the following functionality that is missing from the standard Java language:

1. Uses the "module" concept to create a logical boundary around a set of Java packages
2. Modules are implemented as "bundles". Each bundle holds metadata that describes which Java packages are visible outside the scope of a bundle:
   - OSGi extends Java's concept of "`public`" such that any package within a bundle can be public in either a local sense (within the bundle) or a global sense (outside the bundle)
   - By default a bundle uses a ***shares nothing*** approach. Therefore, any package not explicitly "exported" remains hidden within the bundle's logical boundary
3. Bundles are deployed as stand-alone, but interdependent units of code. This means that only those bundles required for a specific version of an application need be installed

How Does OSGi Fill the Gaps in Java's Modularity Concept?

OSGi implements the following functionality that is missing from the standard Java language:

1. Uses the "module" concept to create a logical boundary around a set of Java packages
2. Modules are implemented as "bundles". Each bundle holds metadata that describes which Java packages are visible outside the scope of a bundle:
   - OSGi extends Java's concept of "`public`" such that any package within a bundle can be public in either a local sense (within the bundle) or a global sense (outside the bundle)
   - By default a bundle uses a ***shares nothing*** approach. Therefore, any package not explicitly "exported" remains hidden within the bundle's logical boundary
3. Bundles are deployed as stand-alone, but interdependent units of code. This means that only those bundles required for a specific version of an application need be installed
4. Checks that dependency and version requirements between bundles have been satisfied before executing an application

How Does OSGi Fill the Gaps in Java's Modularity Concept?

OSGi implements the following functionality that is missing from the standard Java language:

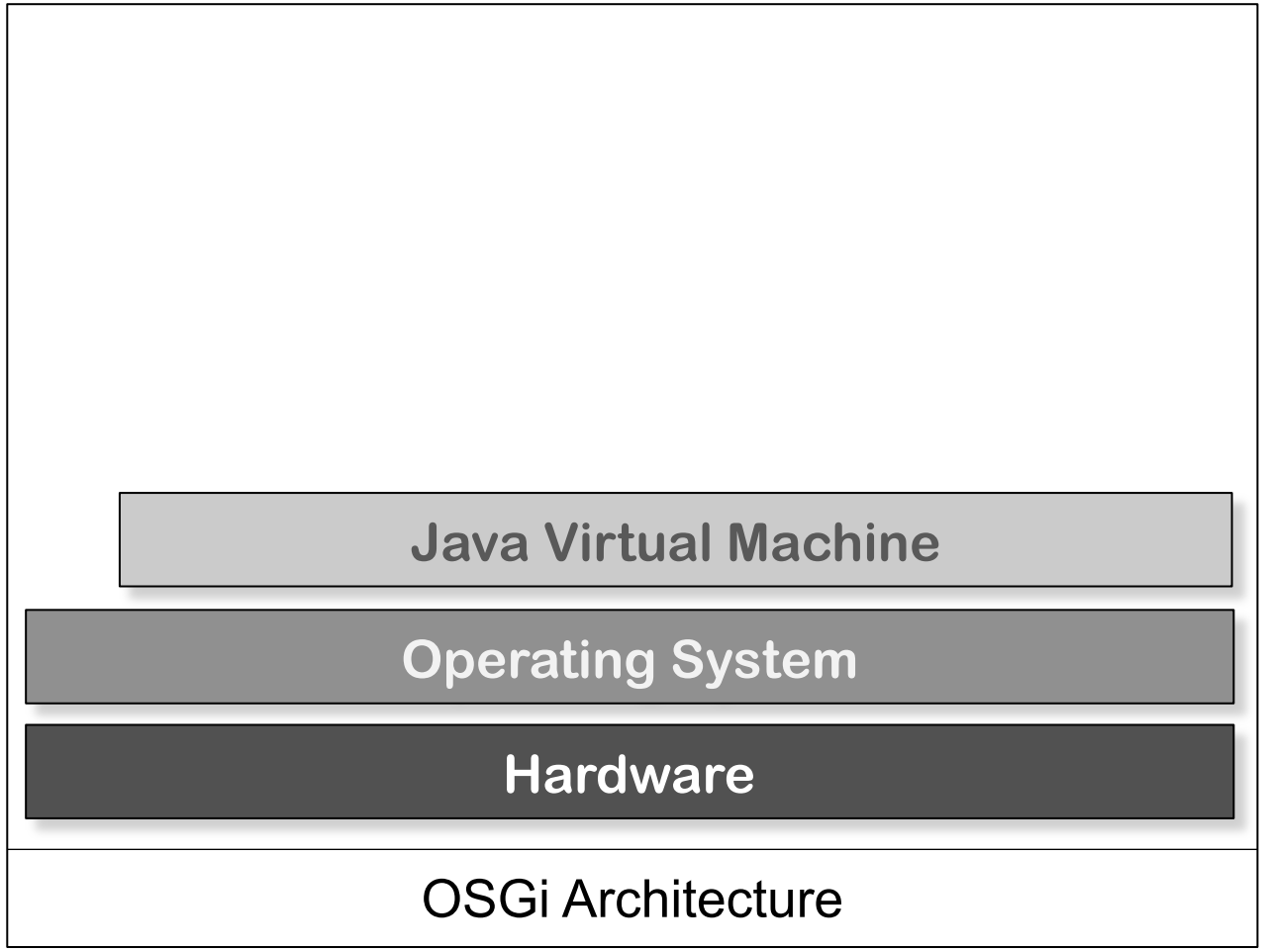
1. Uses the "module" concept to create a logical boundary around a set of Java packages
2. Modules are implemented as "bundles". Each bundle holds metadata that describes which Java packages are visible outside the scope of a bundle:
   - OSGi extends Java's concept of "`public`" such that any package within a bundle can be public in either a local sense (within the bundle) or a global sense (outside the bundle)
   - By default a bundle uses a ***shares nothing*** approach. Therefore, any package not explicitly "exported" remains hidden within the bundle's logical boundary
3. Bundles are deployed as stand-alone, but interdependent units of code. This means that only those bundles required for a specific version of an application need be installed
4. Checks that dependency and version requirements between bundles have been satisfied before executing an application
5. Allows you to create applications that use an extensibility mechanism or dynamic runtime behaviour without having to write low-level (and error prone) Java class loader coding

OSGi

Architecture

OSGi Architecture: Overview 1/2

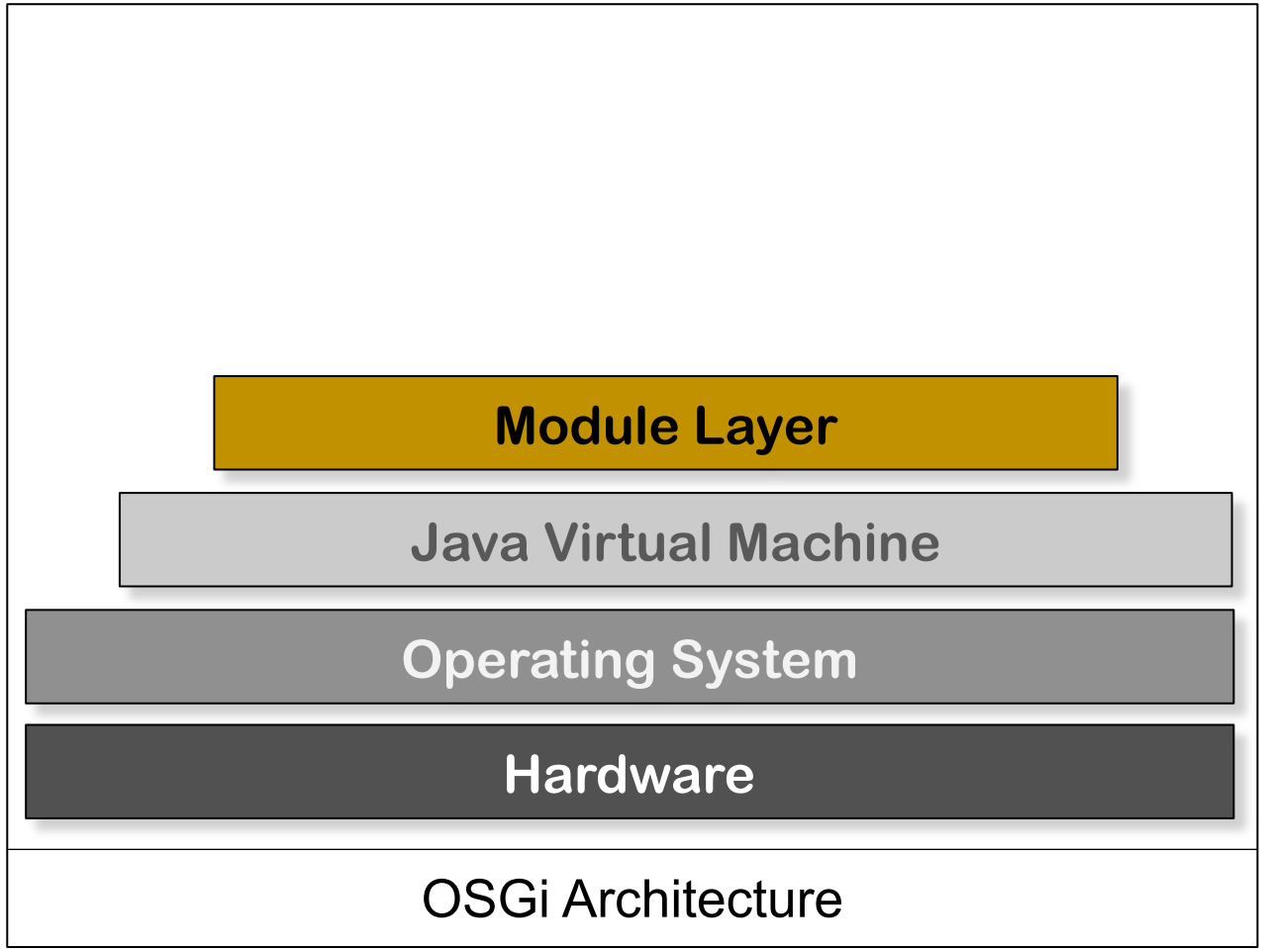
The OSGi architecture is a framework that provides three hierarchical layers and a fourth, orthogonal security layer.



OSGi Architecture: Overview 1/2

The OSGi architecture is a framework that provides three hierarchical layers and a fourth, orthogonal security layer.

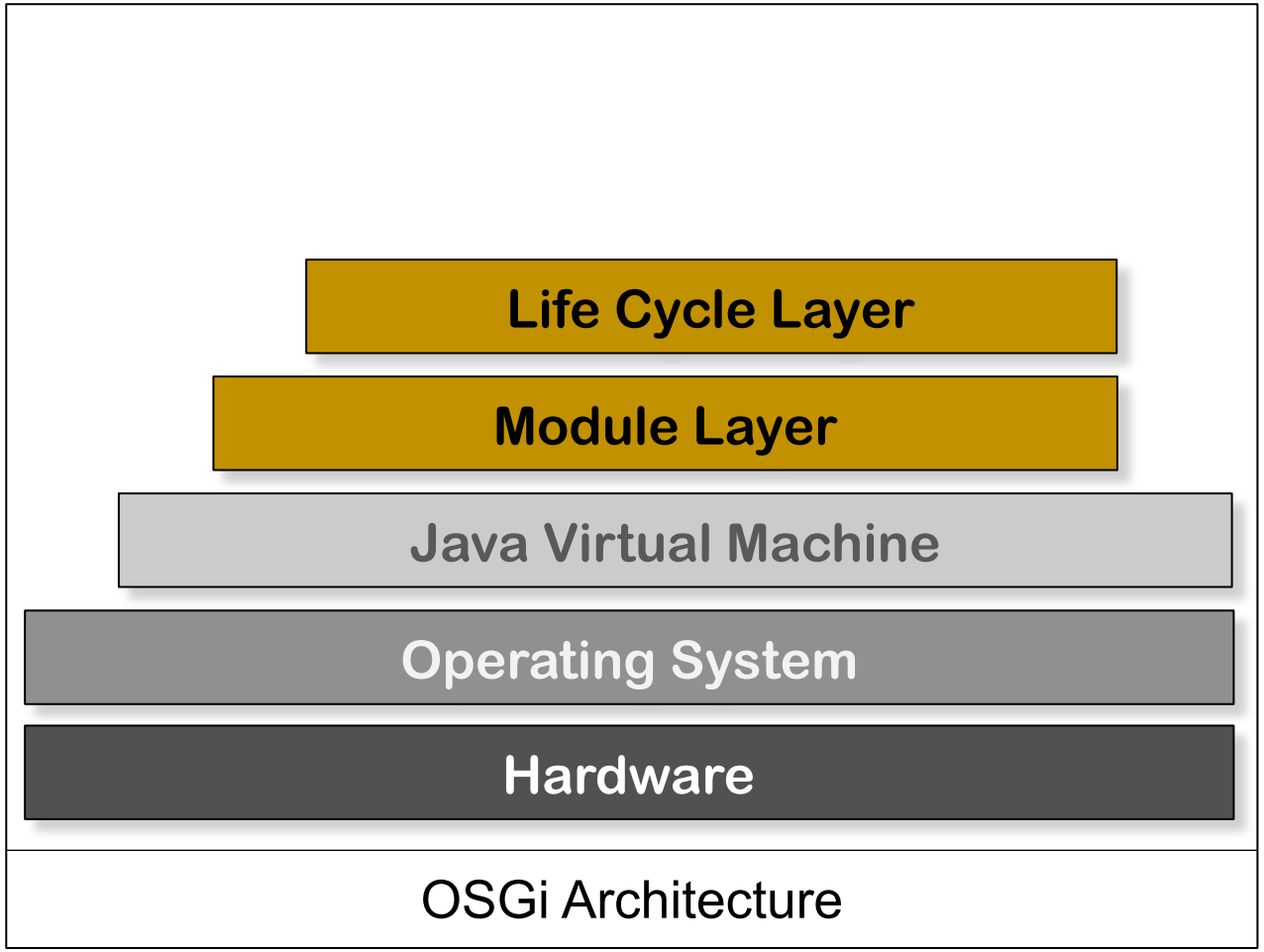
- **Module Layer**
  Defines the OSGi module concept, in which each module is implemented as a “Bundle”. The Module Layer manages how bundles import and export their code and performs dependency resolution.



OSGi Architecture: Overview 1/2

The OSGi architecture is a framework that provides three hierarchical layers and a fourth, orthogonal security layer.

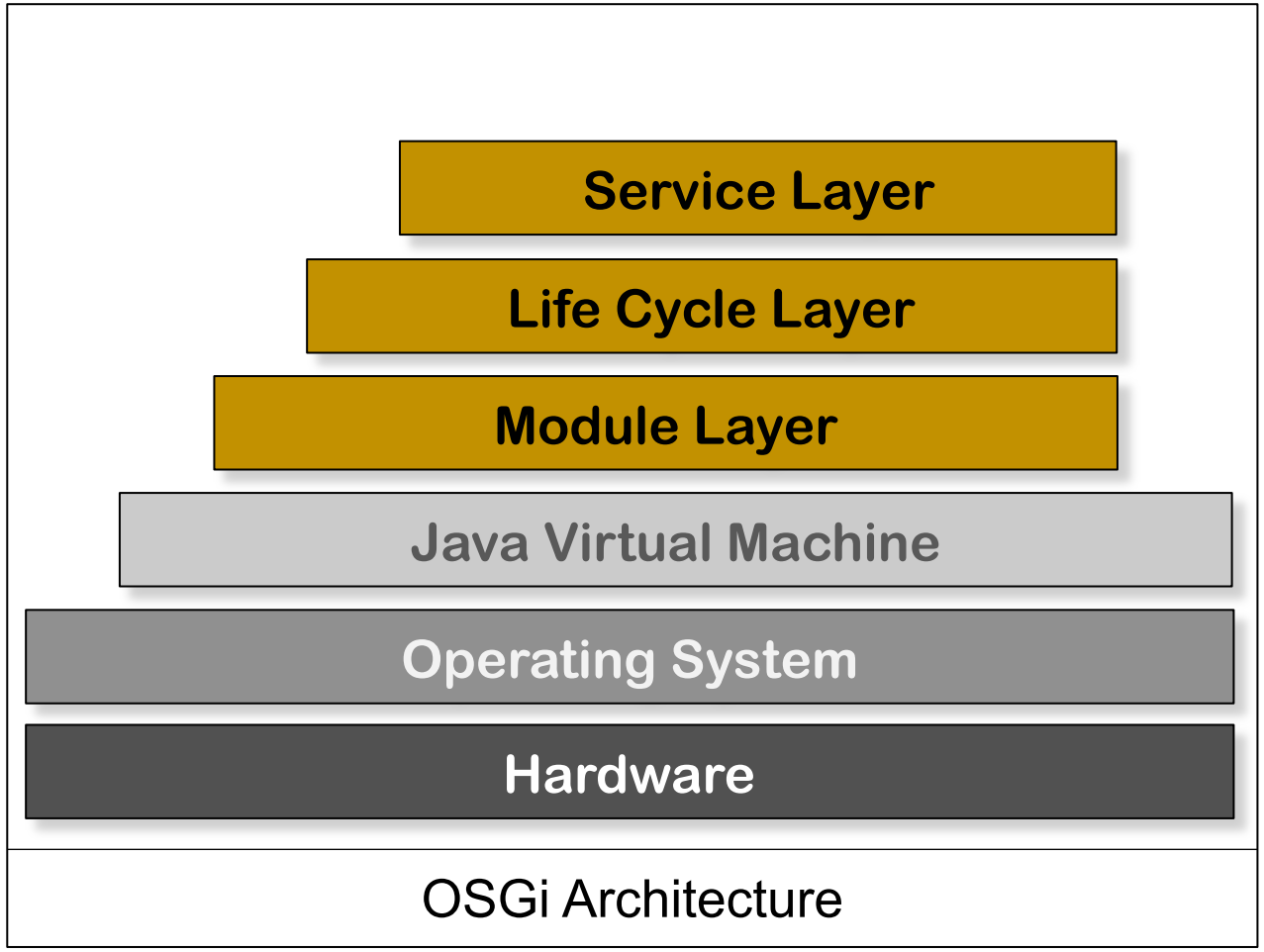
- **Module Layer**
  Defines the OSGi module concept, in which each module is implemented as a “Bundle”. The Module Layer manages how bundles import and export their code and performs dependency resolution.
- **Life Cycle Layer**
  Provides the management API by which bundles are started, stopped, installed and upgraded etc.



OSGi Architecture: Overview 1/2

The OSGi architecture is a framework that provides three hierarchical layers and a fourth, orthogonal security layer.

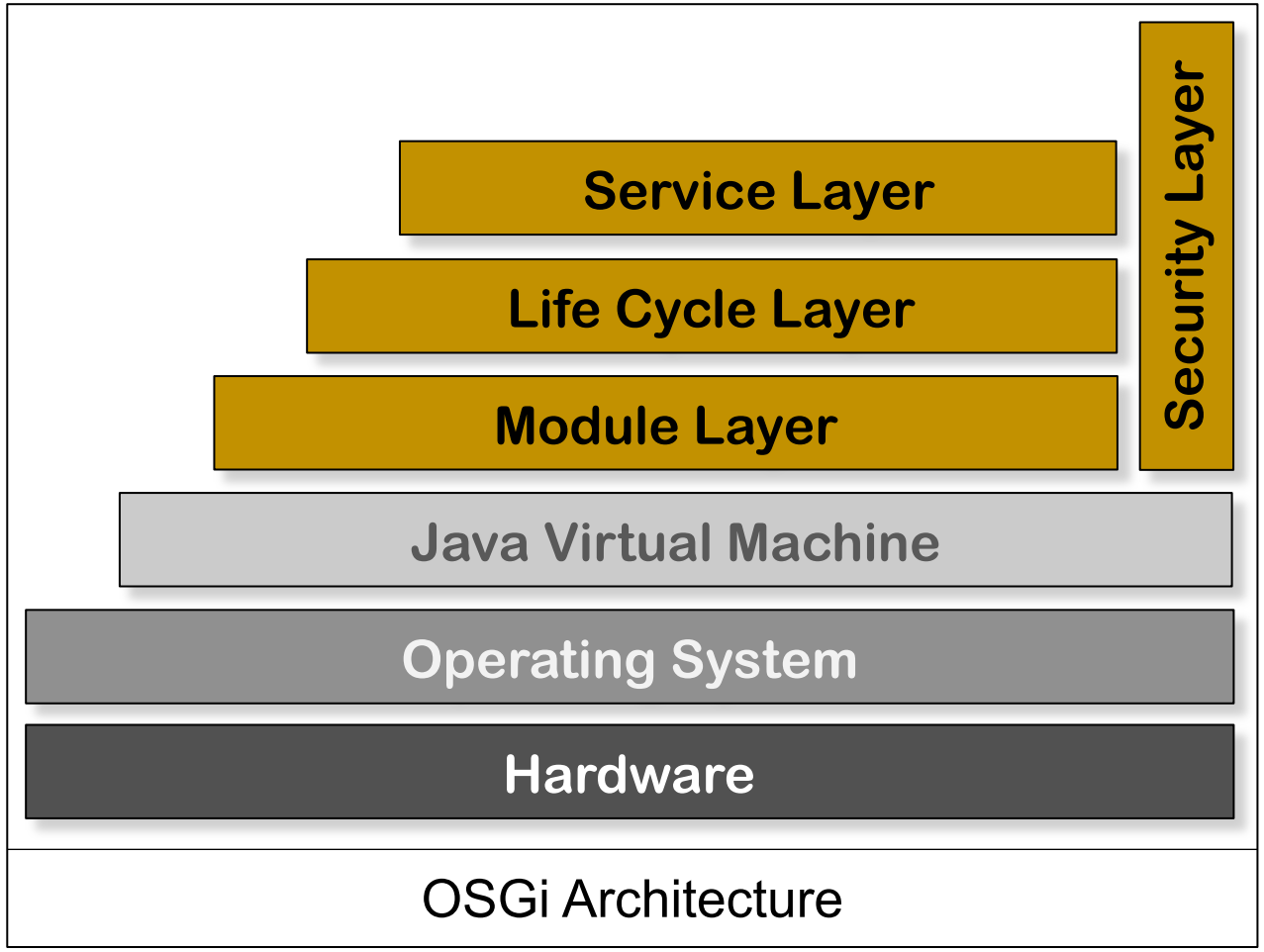
- **Module Layer**
  Defines the OSGi module concept, in which each module is implemented as a “Bundle”. The Module Layer manages how bundles import and export their code and performs dependency resolution.
- **Life Cycle Layer**
  Provides the management API by which bundles are started, stopped, installed and upgraded etc.
- **Service Layer**
  Allows bundles to interact and communicate with each other using the ***publish-find-bind*** interaction pattern



OSGi Architecture: Overview 1/2

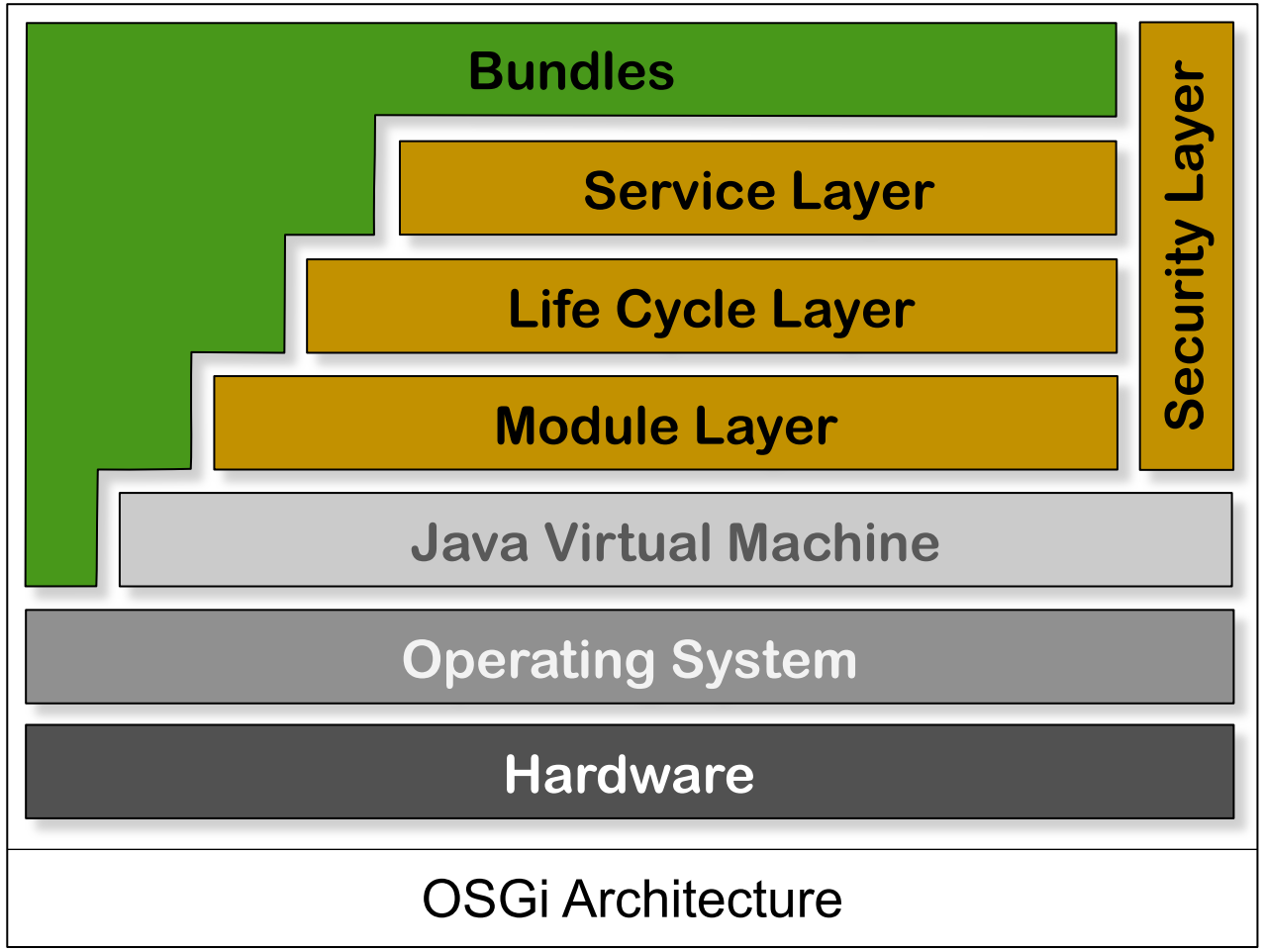
The OSGi architecture is a framework that provides three hierarchical layers and a fourth, orthogonal security layer.

- **Module Layer**
  Defines the OSGi module concept, in which each module is implemented as a “Bundle”. The Module Layer manages how bundles import and export their code and performs dependency resolution.
- **Life Cycle Layer**
  Provides the management API by which bundles are started, stopped, installed and upgraded etc.
- **Service Layer**
  Allows bundles to interact and communicate with each other using the ***publish-find-bind*** interaction pattern
- **Security Layer**
  Used to constrain bundle functionality within pre-defined limits



OSGi Architecture: Overview 2/2

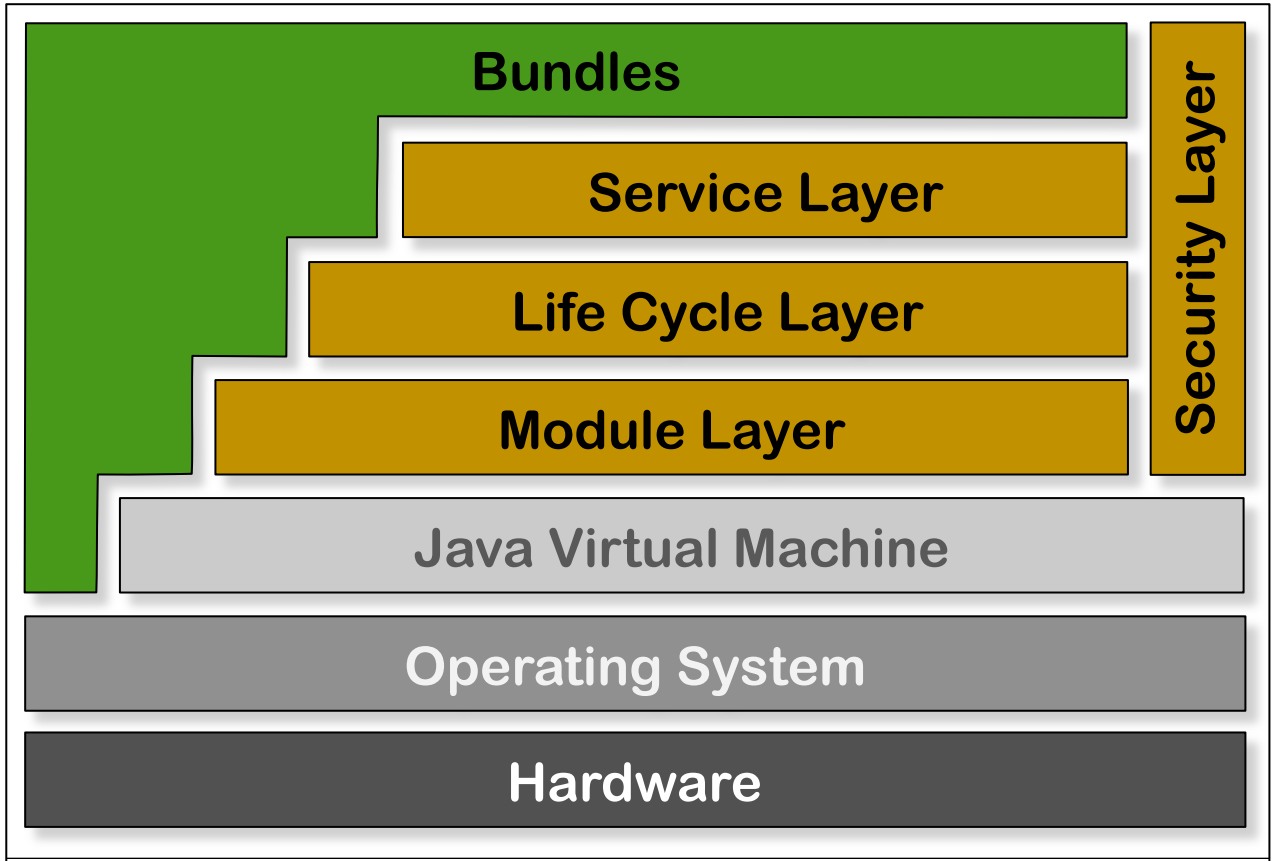
The OSGi "***module***" concept defines how logical boundaries are placed around units of Java code.



OSGi Architecture: Overview 2/2

The OSGi "***module***" concept defines how logical boundaries are placed around units of Java code.

An OSGi "***bundle***" implements the module concept by adding extra metadata to a standard Java Archive (`.jar`) file.



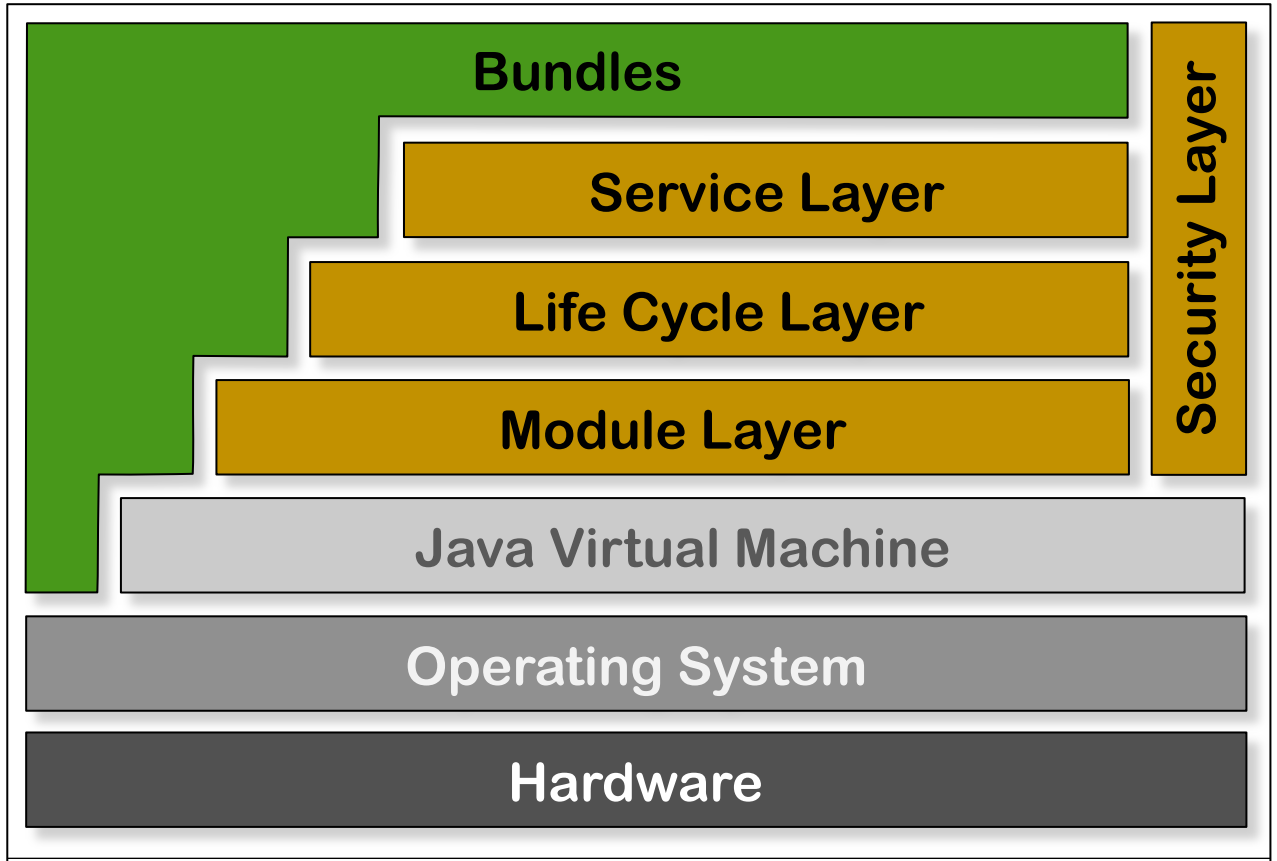
OSGi Architecture

OSGi Architecture: Overview 2/2

The OSGi "***module***" concept defines how logical boundaries are placed around units of Java code.

An OSGi "***bundle***" implements the module concept by adding extra metadata to a standard Java Archive (`.jar`) file.

In OSGi terminology, the terms "bundle" and "module" can be used interchangeably.



OSGi Architecture

OSGi Architecture: Overview 2/2

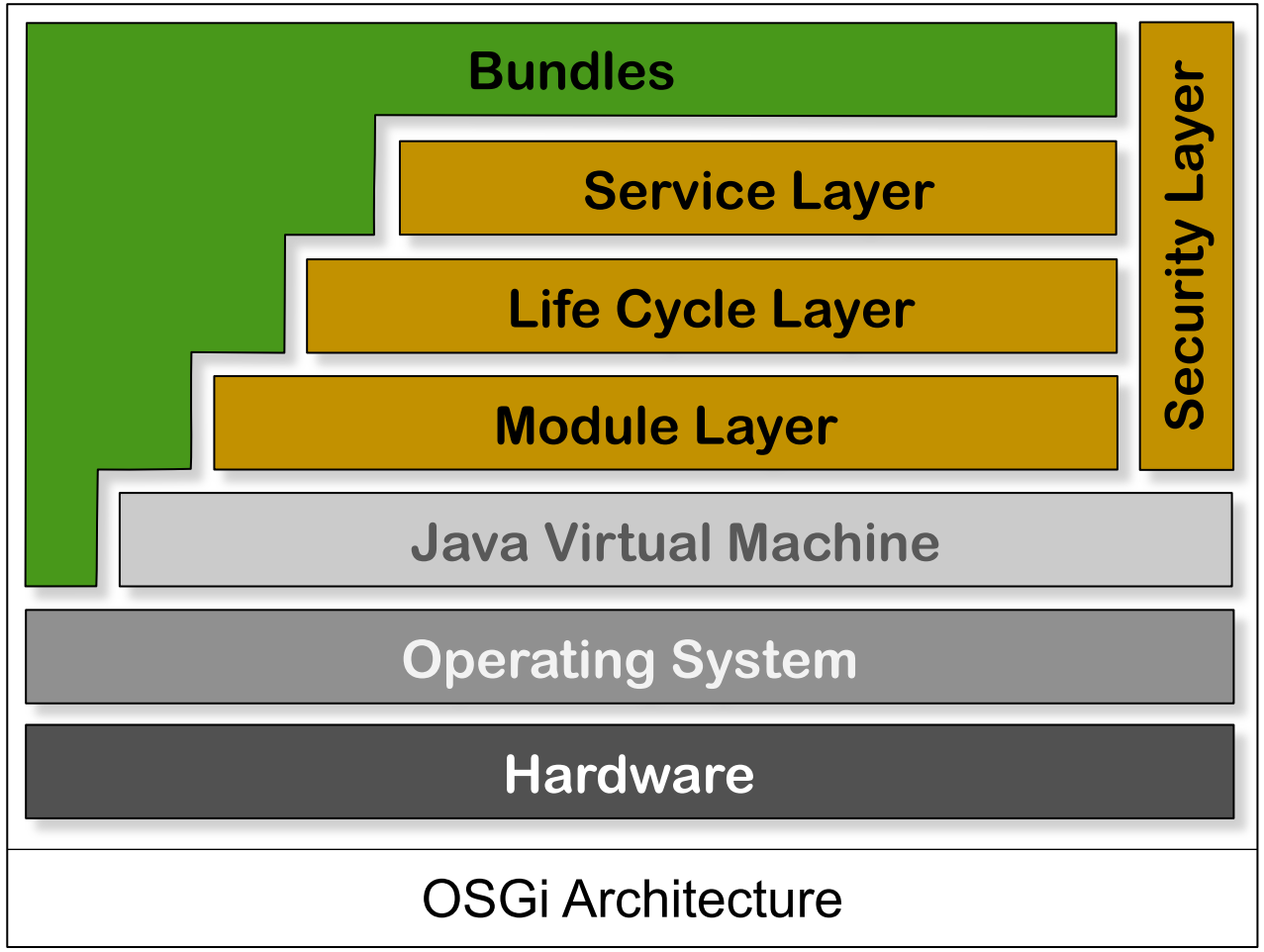
The OSGi "***module***" concept defines how logical boundaries are placed around units of Java code.

An OSGi "***bundle***" implements the module concept by adding extra metadata to a standard Java Archive (`.jar`) file.

In OSGi terminology, the terms "bundle" and "module" can be used interchangeably.

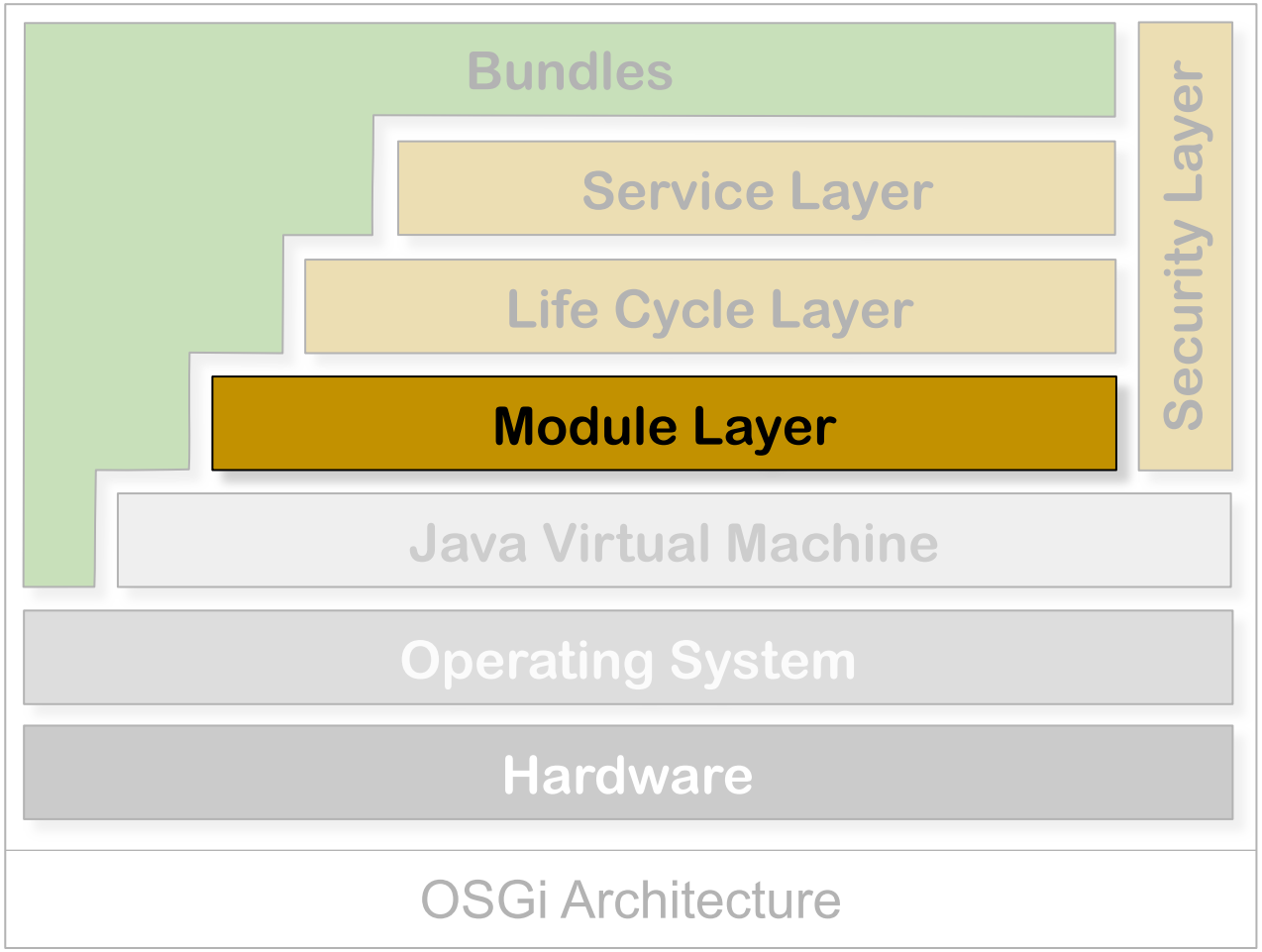
Bundles are managed as self-contained entities that can be installed, uninstalled, upgraded, started and stopped.

Yet at the same time, version-specific dependencies between bundles can be declared in order to form large scale applications.



OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

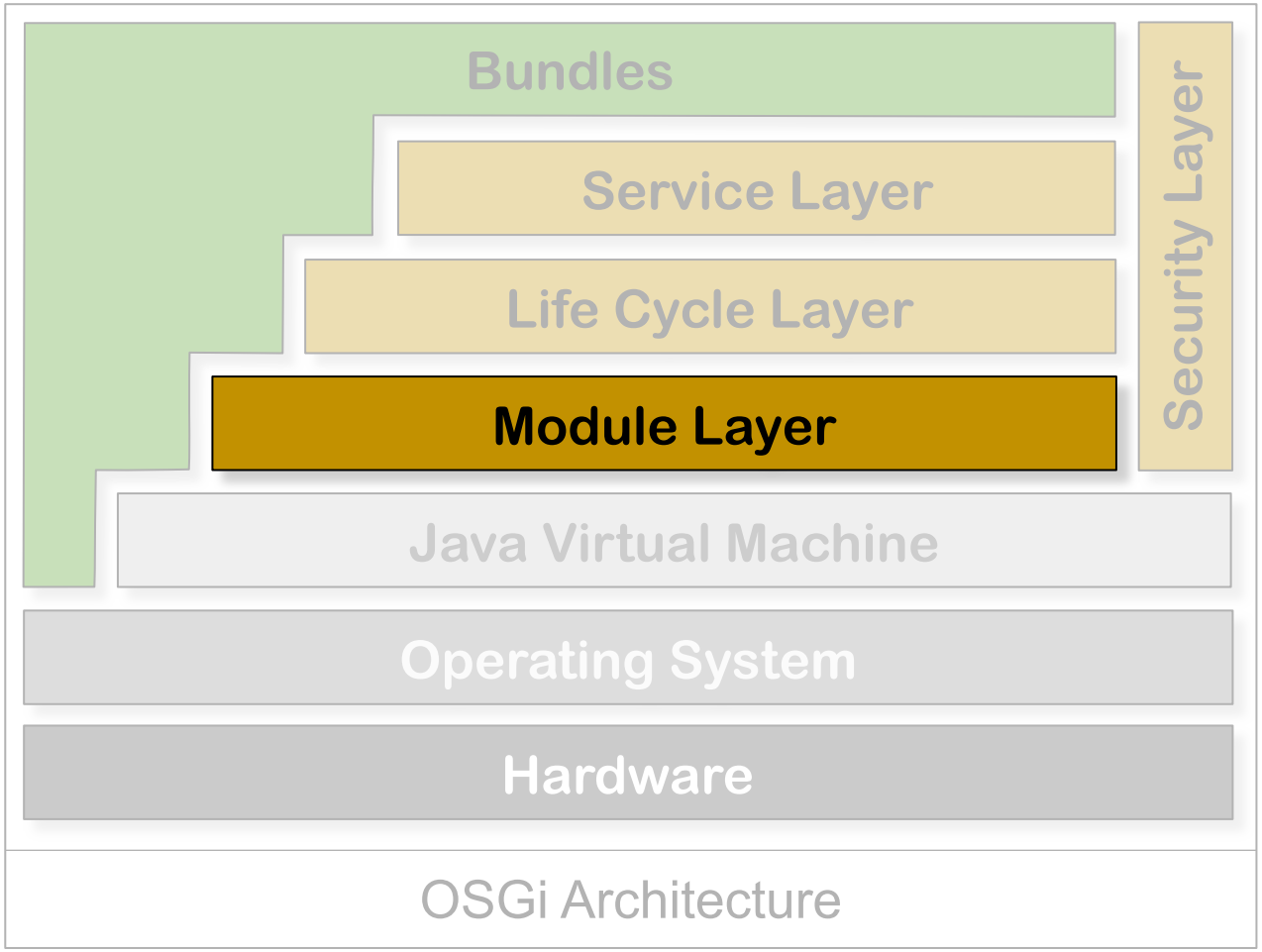


All dependency and version resolution is performed by the Module Layer.

OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

In the simple case of transforming a library JAR file into a bundle, the addition of such metadata is non-invasive to the existing code.



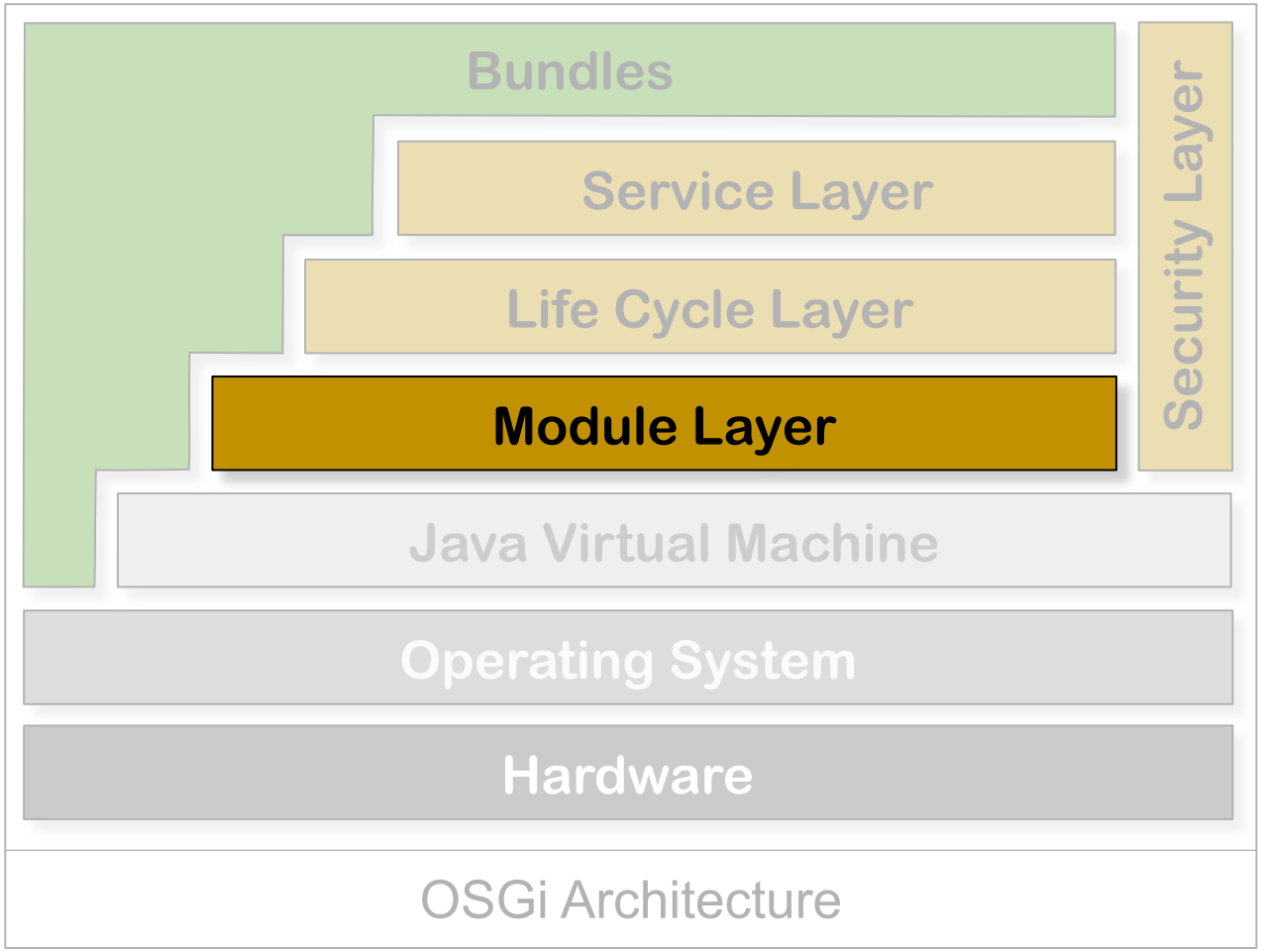
All dependency and version resolution is performed by the Module Layer.

OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

In the simple case of transforming a library JAR file into a bundle, the addition of such metadata is non-invasive to the existing code.

A bundle is more powerful than a regular JAR file for the following reasons:



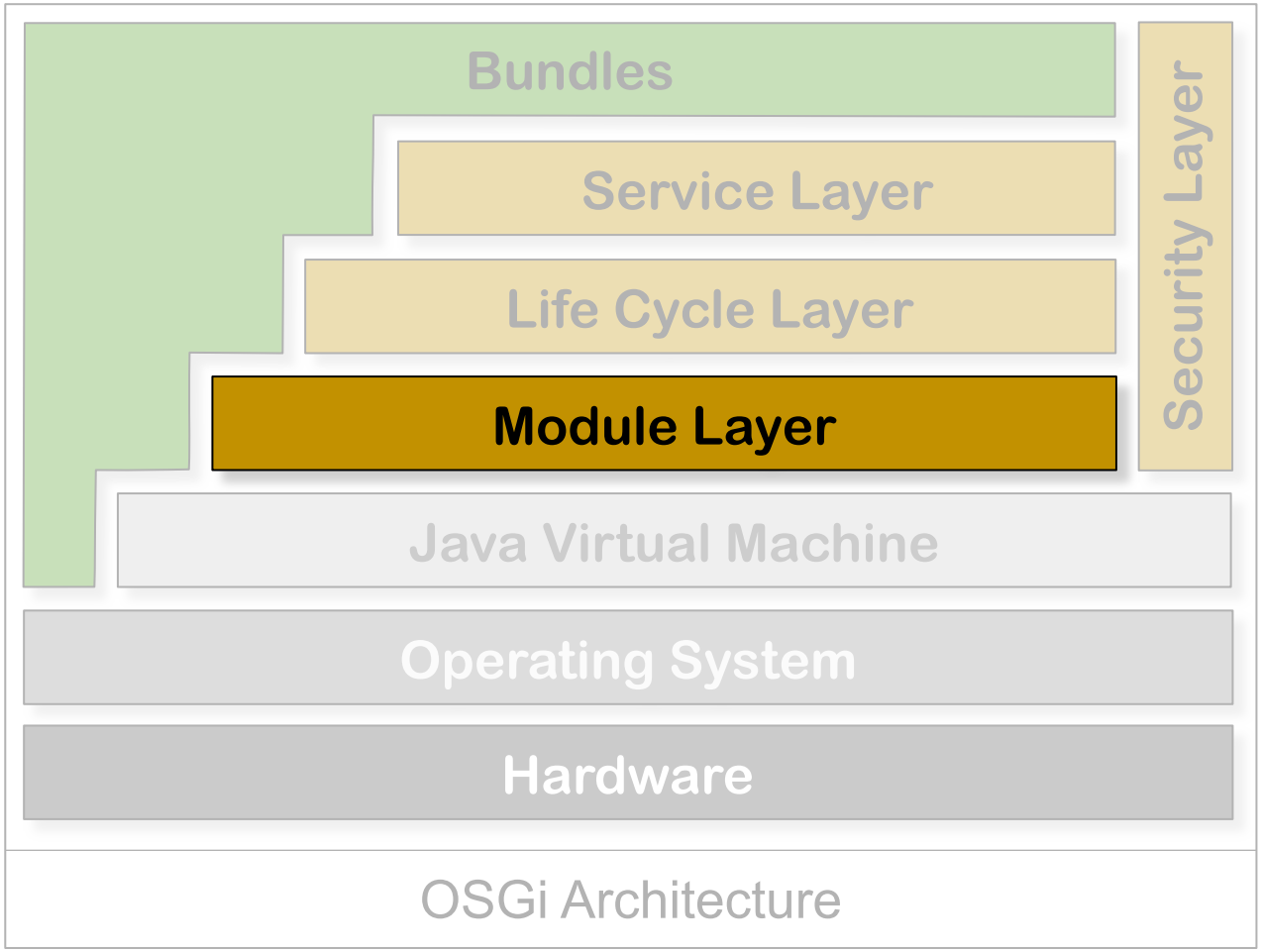
OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

In the simple case of transforming a library JAR file into a bundle, the addition of such metadata is non-invasive to the existing code.

A bundle is more powerful than a regular JAR file for the following reasons:

1. Only those Java packages explicitly “exported” from the bundle will be visible externally



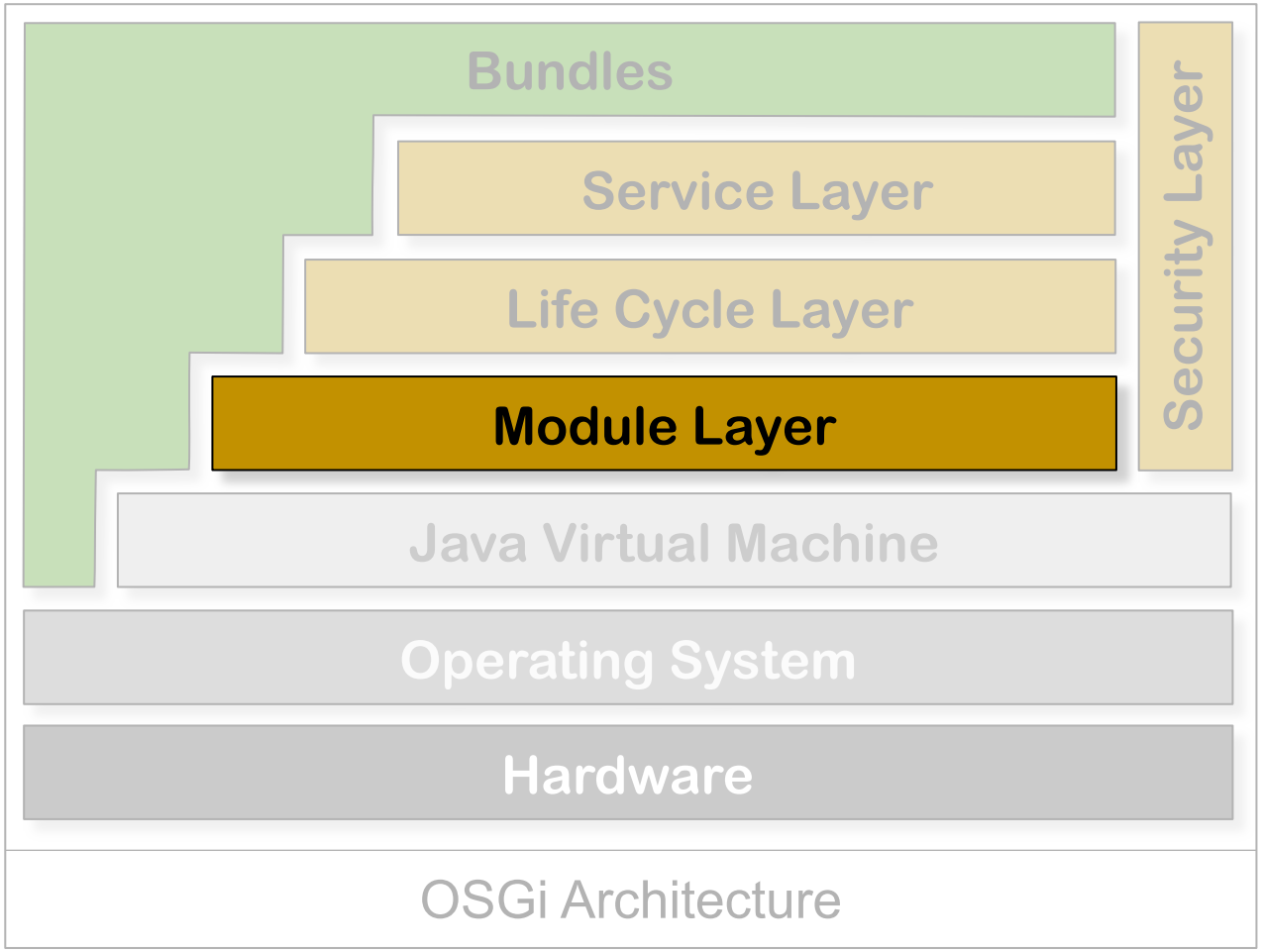
OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

In the simple case of transforming a library JAR file into a bundle, the addition of such metadata is non-invasive to the existing code.

A bundle is more powerful than a regular JAR file for the following reasons:

1. Only those Java packages explicitly “exported” from the bundle will be visible externally
2. Dependencies on Java packages found in other bundles must be “imported” by explicit declaration



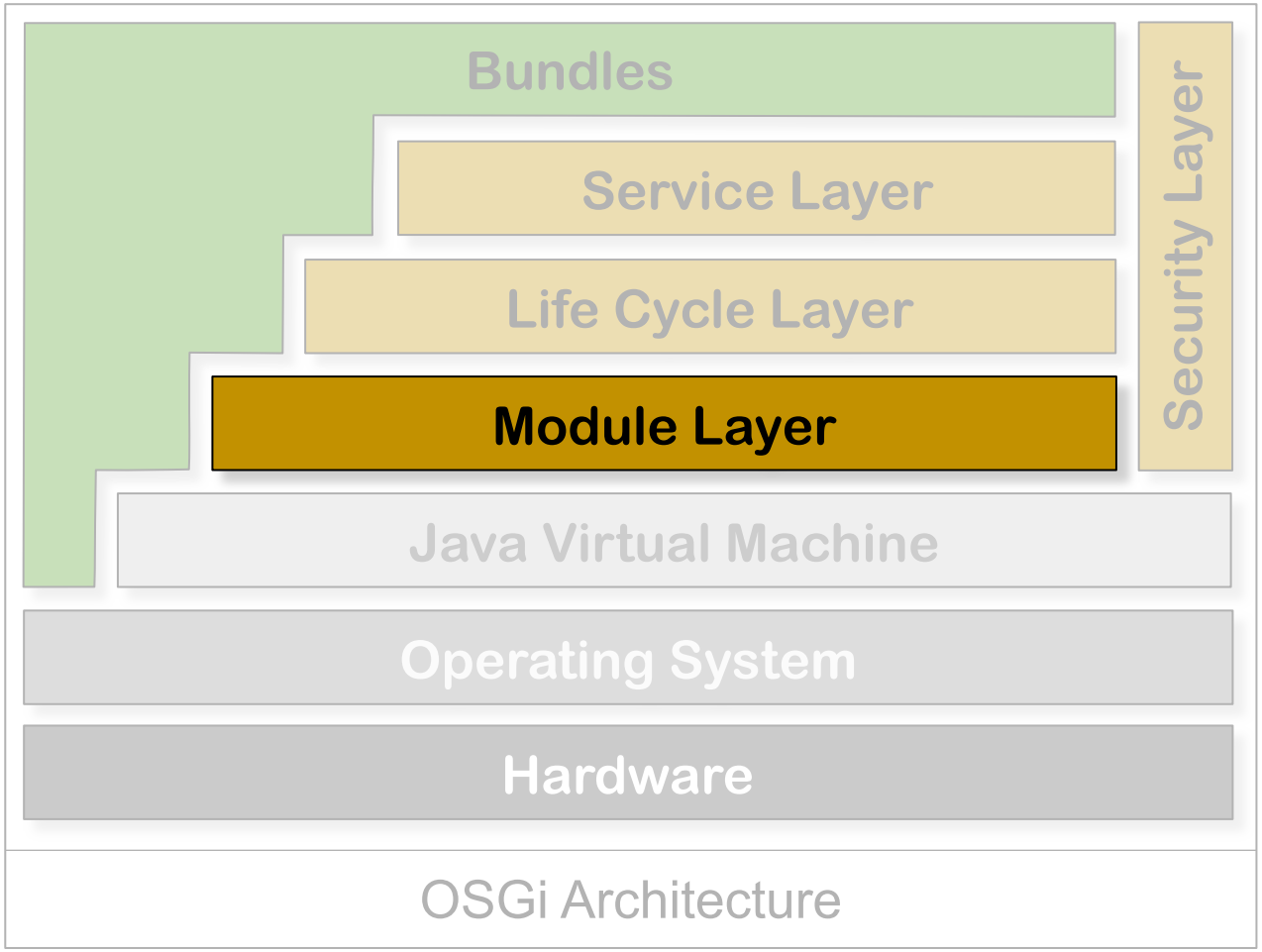
OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

In the simple case of transforming a library JAR file into a bundle, the addition of such metadata is non-invasive to the existing code.

A bundle is more powerful than a regular JAR file for the following reasons:

1. Only those Java packages explicitly “exported” from the bundle will be visible externally
2. Dependencies on Java packages found in other bundles must be “imported” by explicit declaration
3. All package imports and exports are further qualified with version numbers



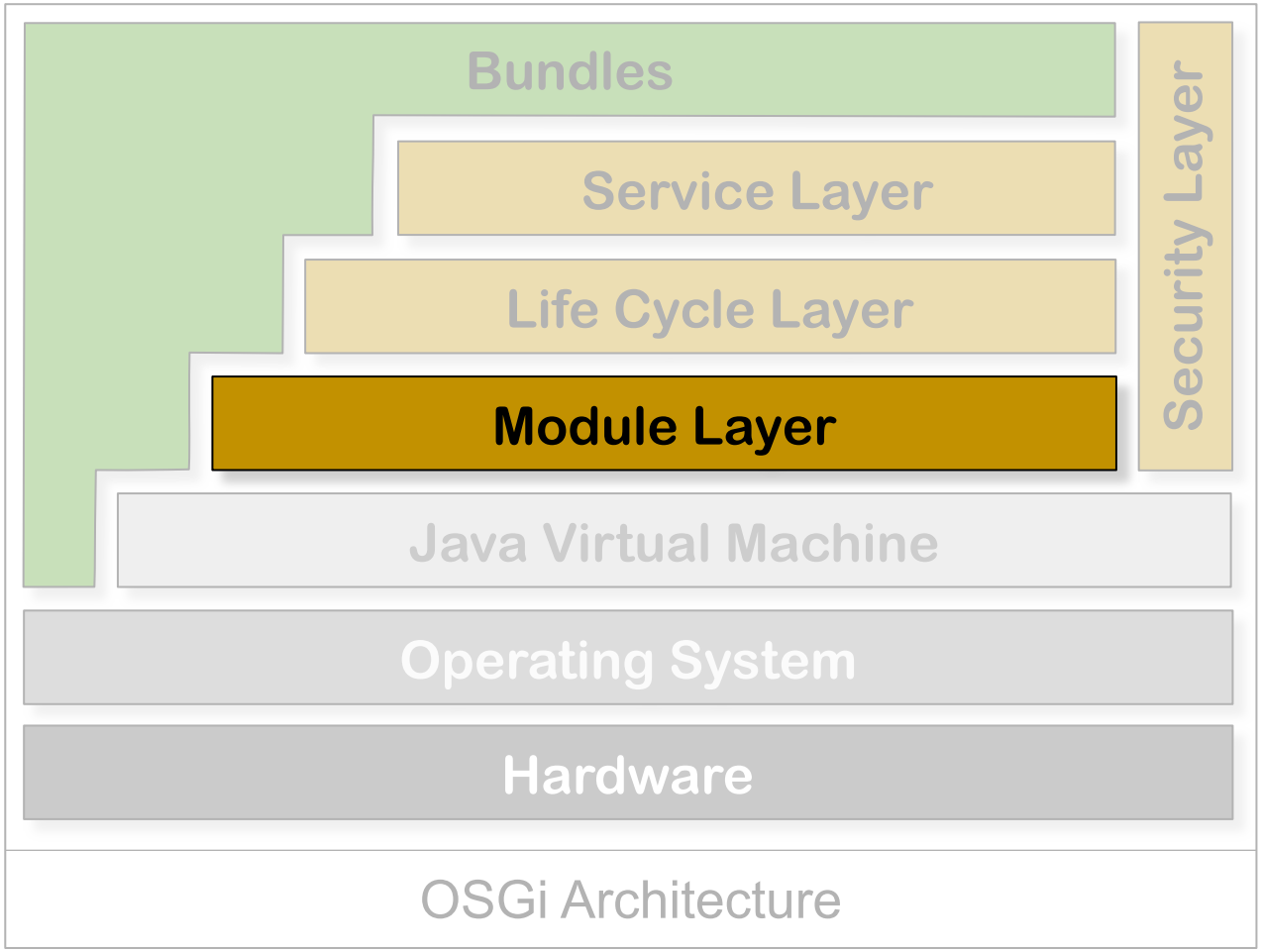
OSGi Architecture: Module Layer

The Module Layer transforms standard `.jar` files into OSGi bundles by the addition of metadata in the file `META-INF/MANIFEST.MF`

In the simple case of transforming a library JAR file into a bundle, the addition of such metadata is non-invasive to the existing code.

A bundle is more powerful than a regular JAR file for the following reasons:

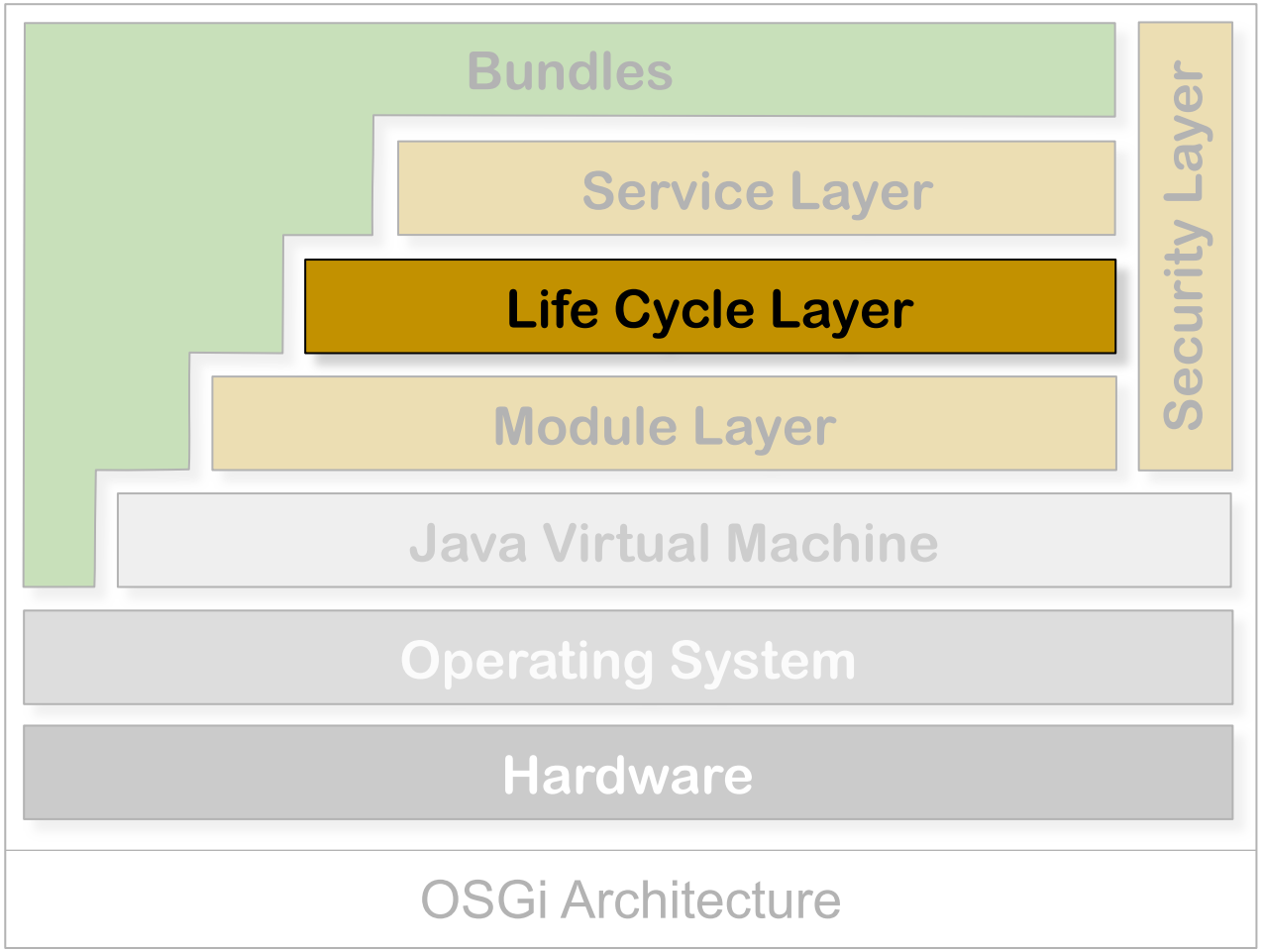
1. Only those Java packages explicitly “exported” from the bundle will be visible externally
2. Dependencies on Java packages found in other bundles must be “imported” by explicit declaration
3. All package imports and exports are further qualified with version numbers



All dependency and version resolution is performed by the Module Layer.

OSGi Architecture: Life Cycle Layer

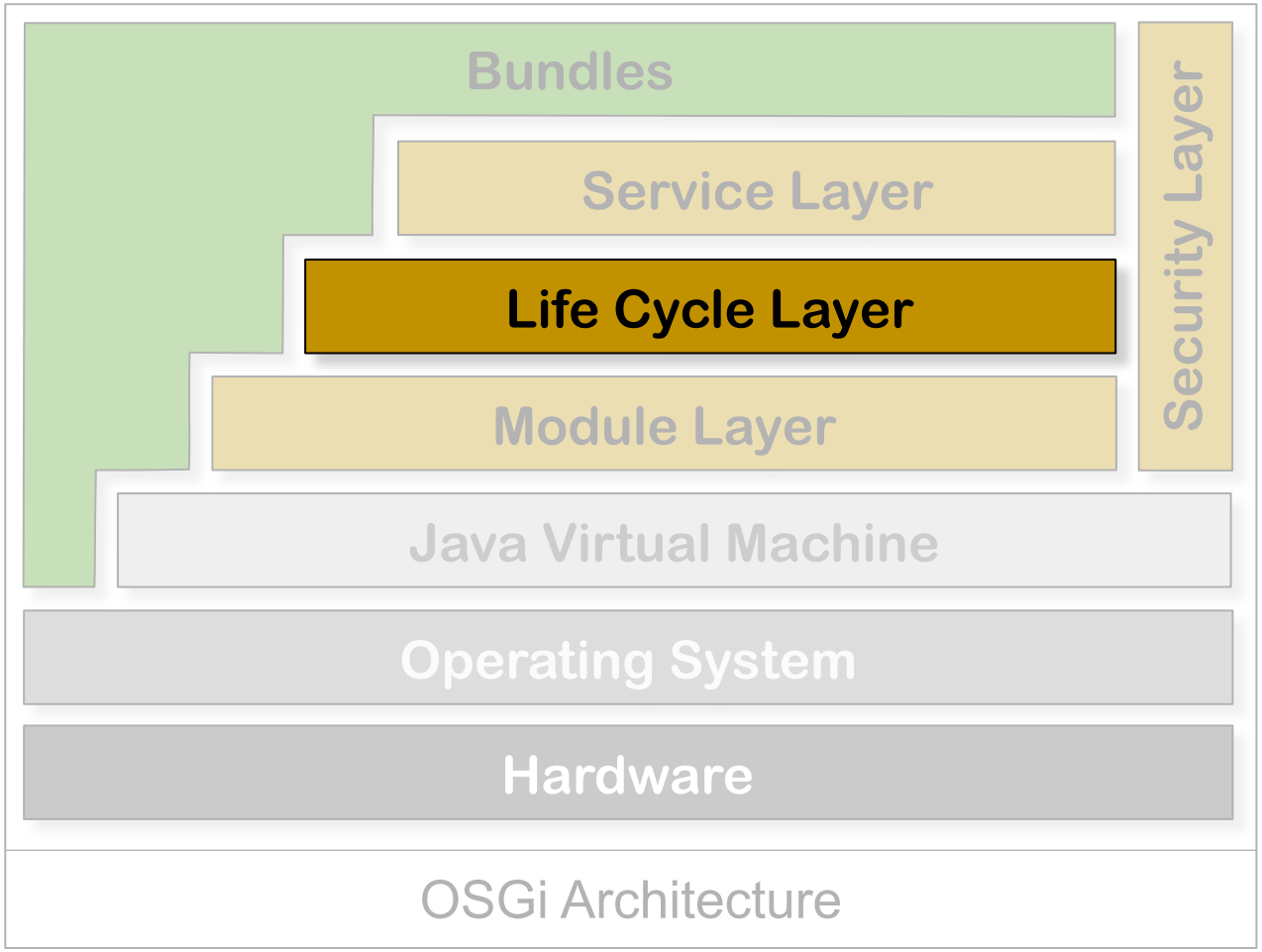
The Life Cycle Layer implements the various tasks involved in bundle management.



OSGi Architecture: Life Cycle Layer

The Life Cycle Layer implements the various tasks involved in bundle management.

From the perspective of the functionality found in a single bundle, the Life Cycle Layer performs two distinct types of task:

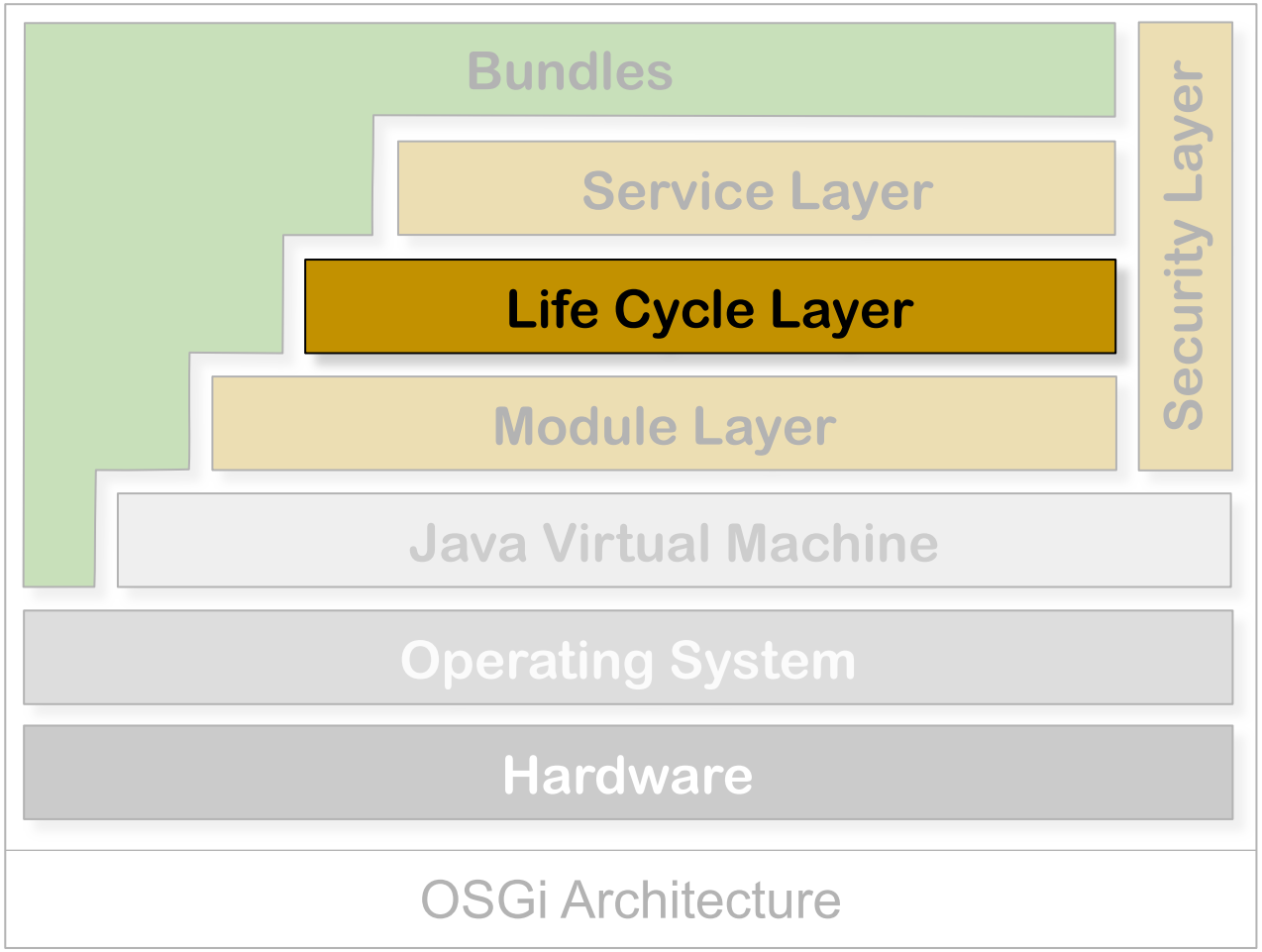


OSGi Architecture: Life Cycle Layer

The Life Cycle Layer implements the various tasks involved in bundle management.

From the perspective of the functionality found in a single bundle, the Life Cycle Layer performs two distinct types of task:

1. **External Tasks**
   The management and administration of bundles as distinct entities (I.E. install, update, uninstall, start and stop)

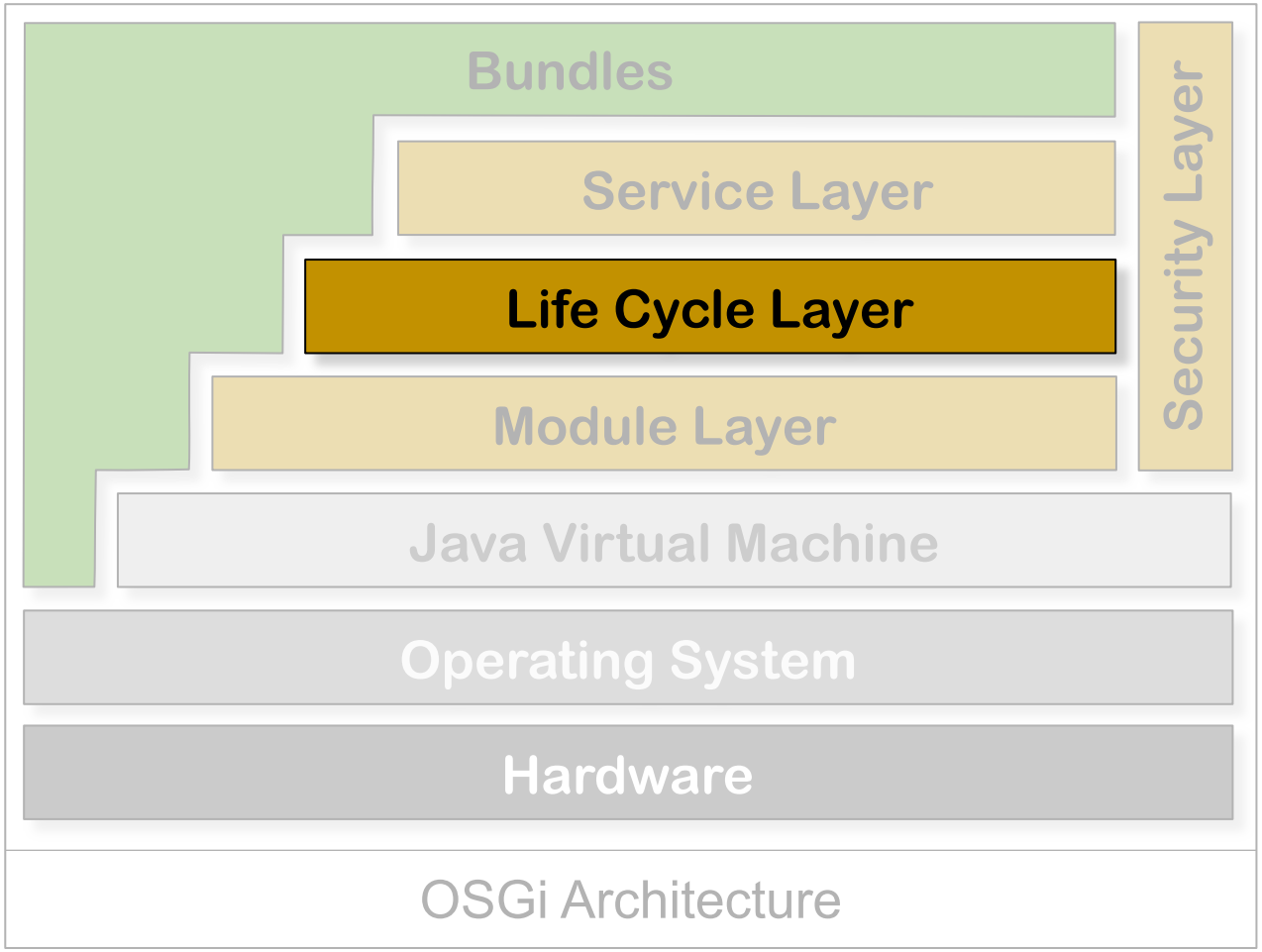


OSGi Architecture: Life Cycle Layer

The Life Cycle Layer implements the various tasks involved in bundle management.

From the perspective of the functionality found in a single bundle, the Life Cycle Layer performs two distinct types of task:

1. **External Tasks**
   The management and administration of bundles as distinct entities (I.E. install, update, uninstall, start and stop)
2. **Internal Tasks**
   Provide an API for the bundle content to interact with the OSGi framework



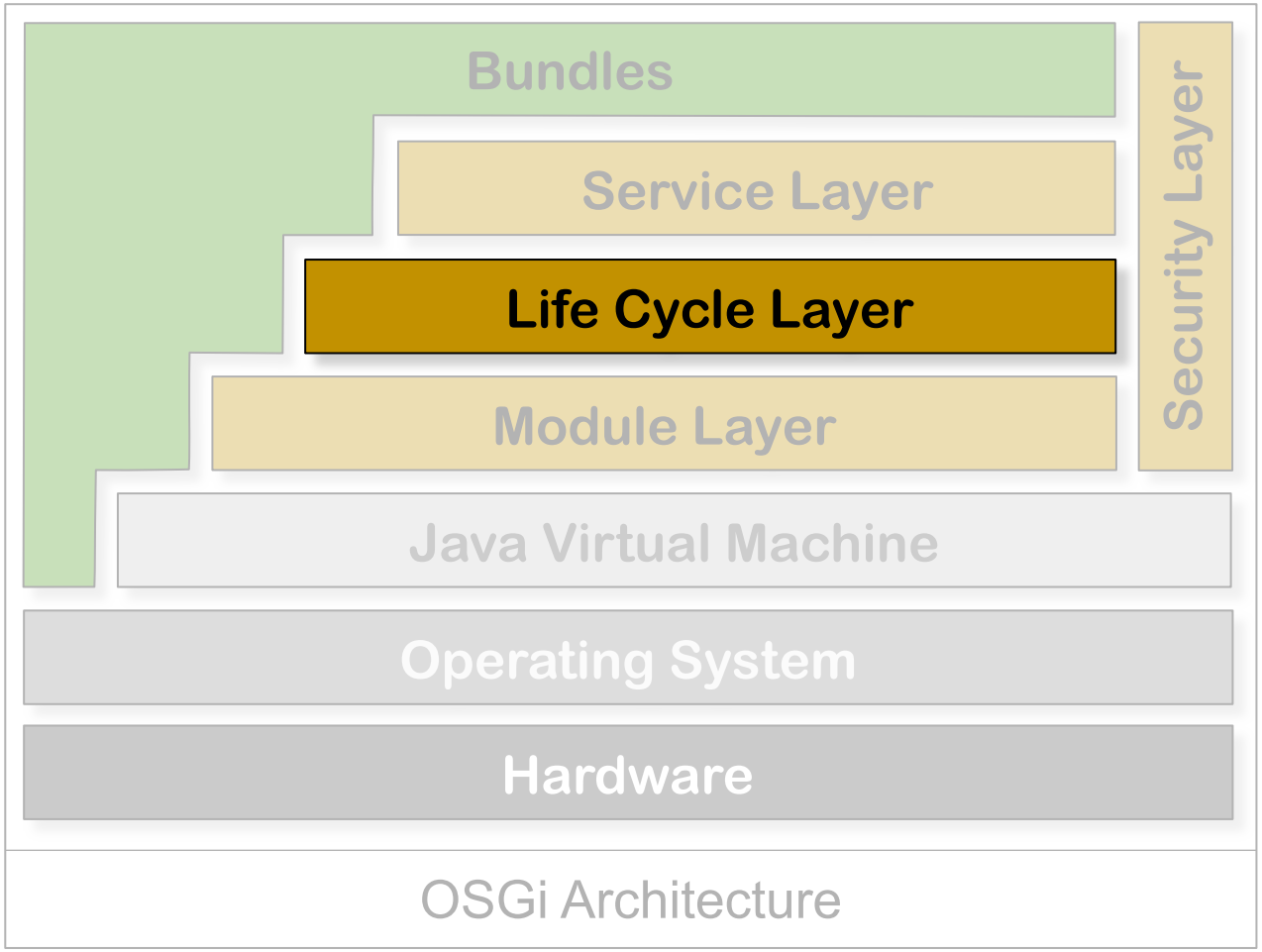
OSGi Architecture: Life Cycle Layer

The Life Cycle Layer implements the various tasks involved in bundle management.

From the perspective of the functionality found in a single bundle, the Life Cycle Layer performs two distinct types of task:

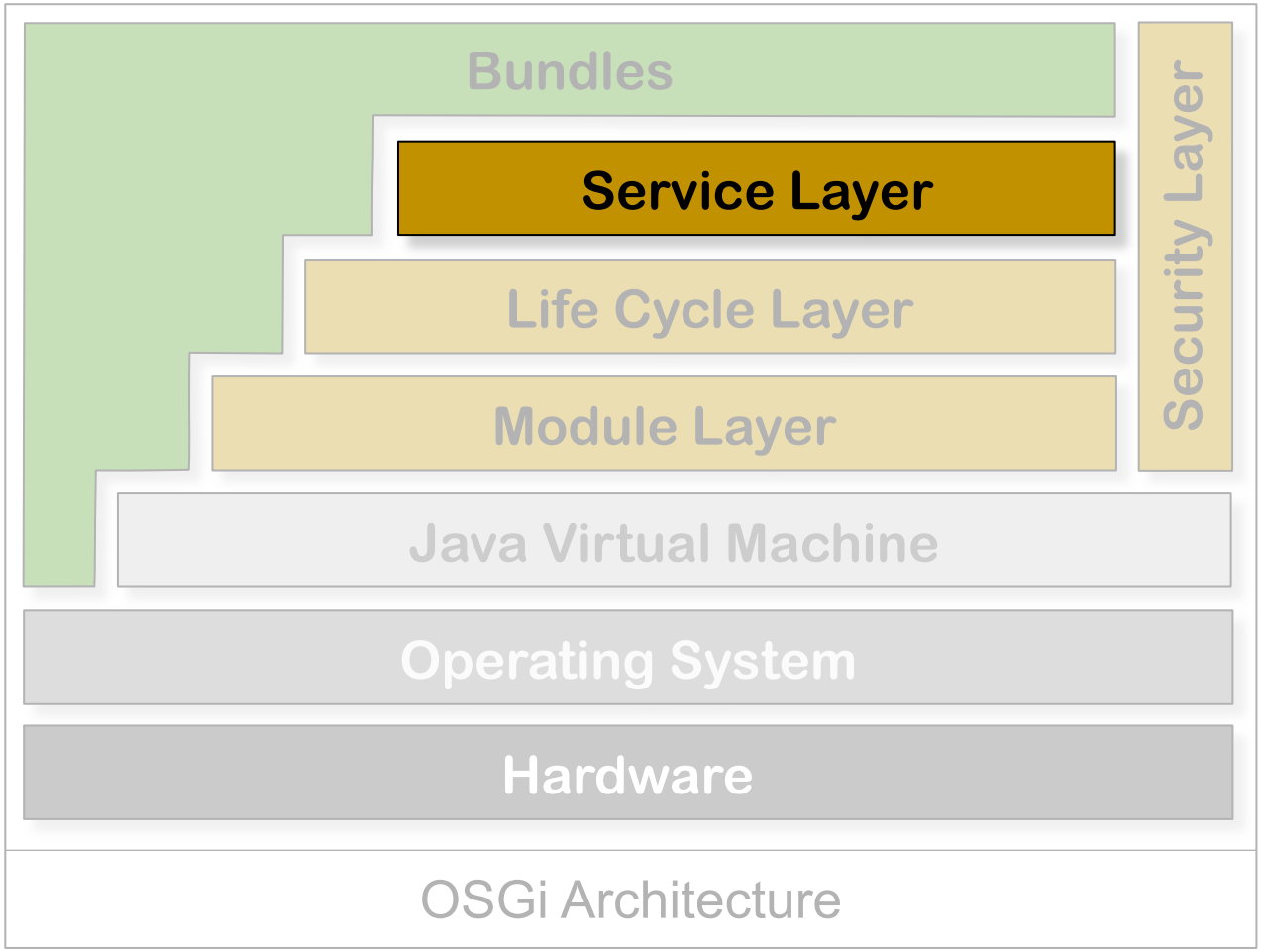
1. **External Tasks**
   The management and administration of bundles as distinct entities (I.E. install, update, uninstall, start and stop)
2. **Internal Tasks**
   Provide an API for the bundle content to interact with the OSGi framework

The Life Cycle Layer allows you to create applications that can be externally managed, self-managed, or some combination of both.



OSGi Architecture: Service Layer 1/2

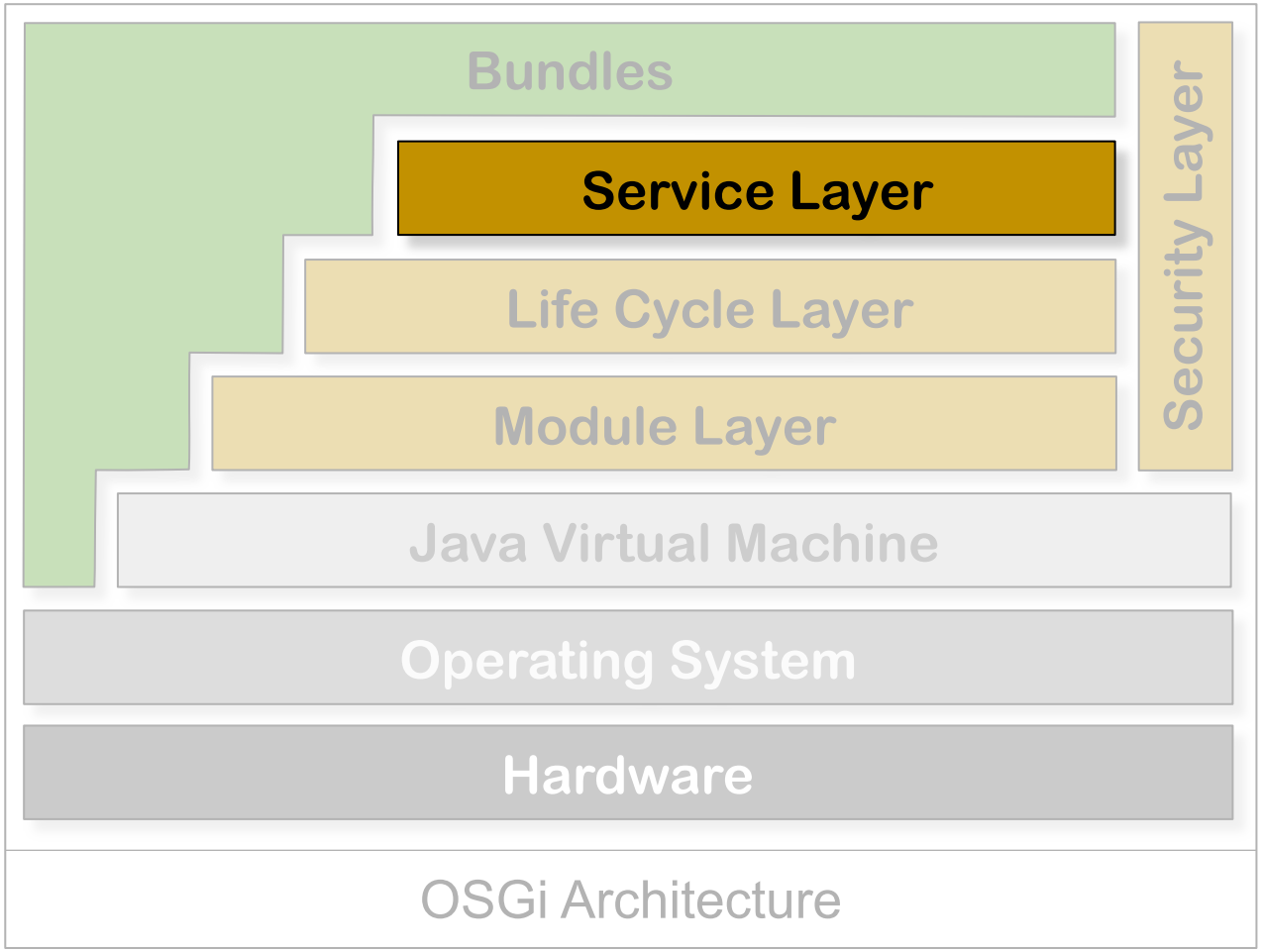
The Service Layer makes bundles accessible via the **Publish-Find-Bind** interaction pattern.



OSGi Architecture: Service Layer 1/2

The Service Layer makes bundles accessible via the **Publish-Find-Bind** interaction pattern.

An OSGi service is simply a Java interface that represents the contractual relationship between a service and the consumers of that service.

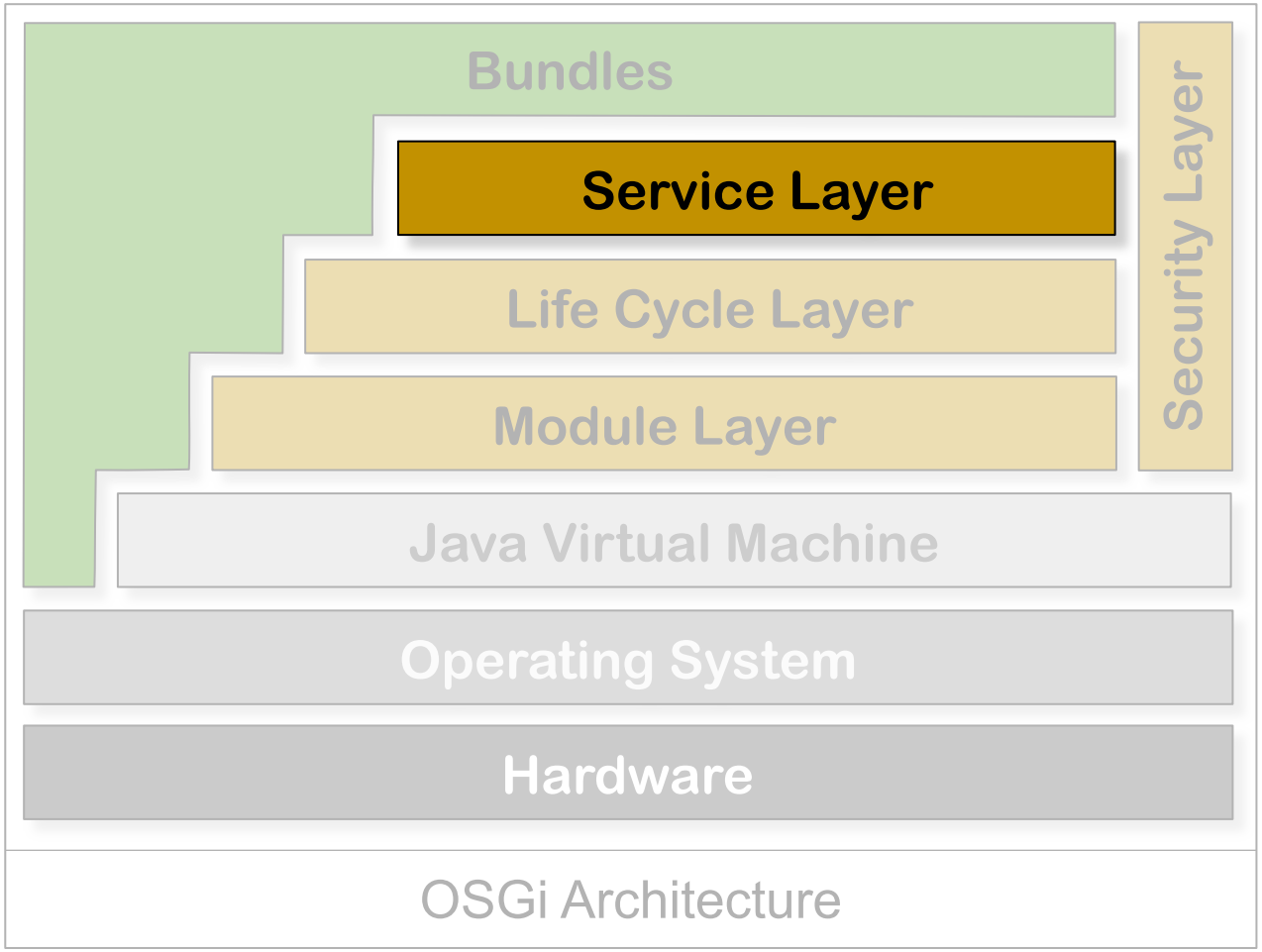


OSGi Architecture: Service Layer 1/2

The Service Layer makes bundles accessible via the **Publish-Find-Bind** interaction pattern.

An OSGi service is simply a Java interface that represents the contractual relationship between a service and the consumers of that service.

Consequently, the service layer is very lightweight, since service providers are typically just POJOs accessed via direct method invocation.



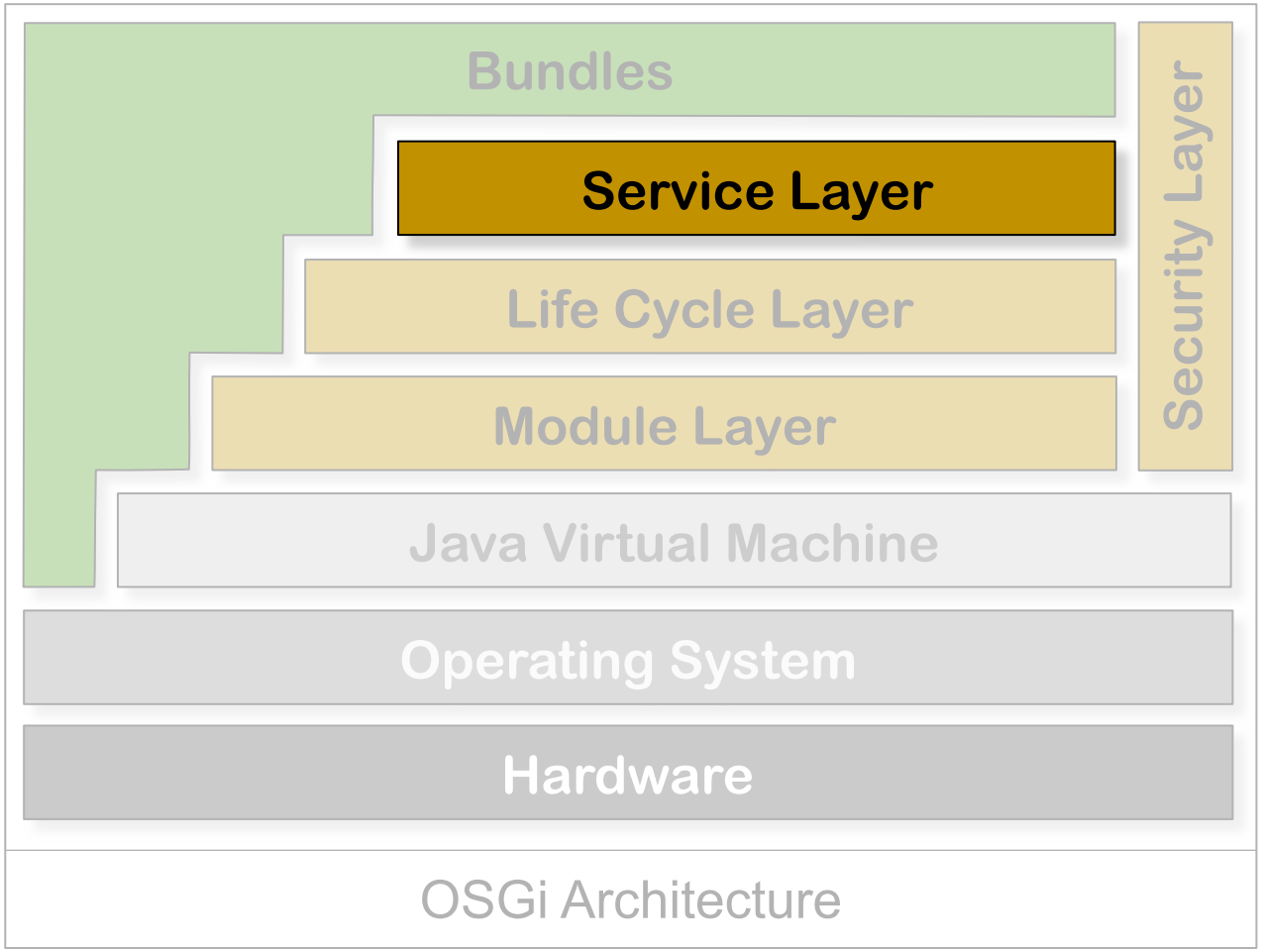
OSGi Architecture: Service Layer 1/2

The Service Layer makes bundles accessible via the **Publish-Find-Bind** interaction pattern.

An OSGi service is simply a Java interface that represents the contractual relationship between a service and the consumers of that service.

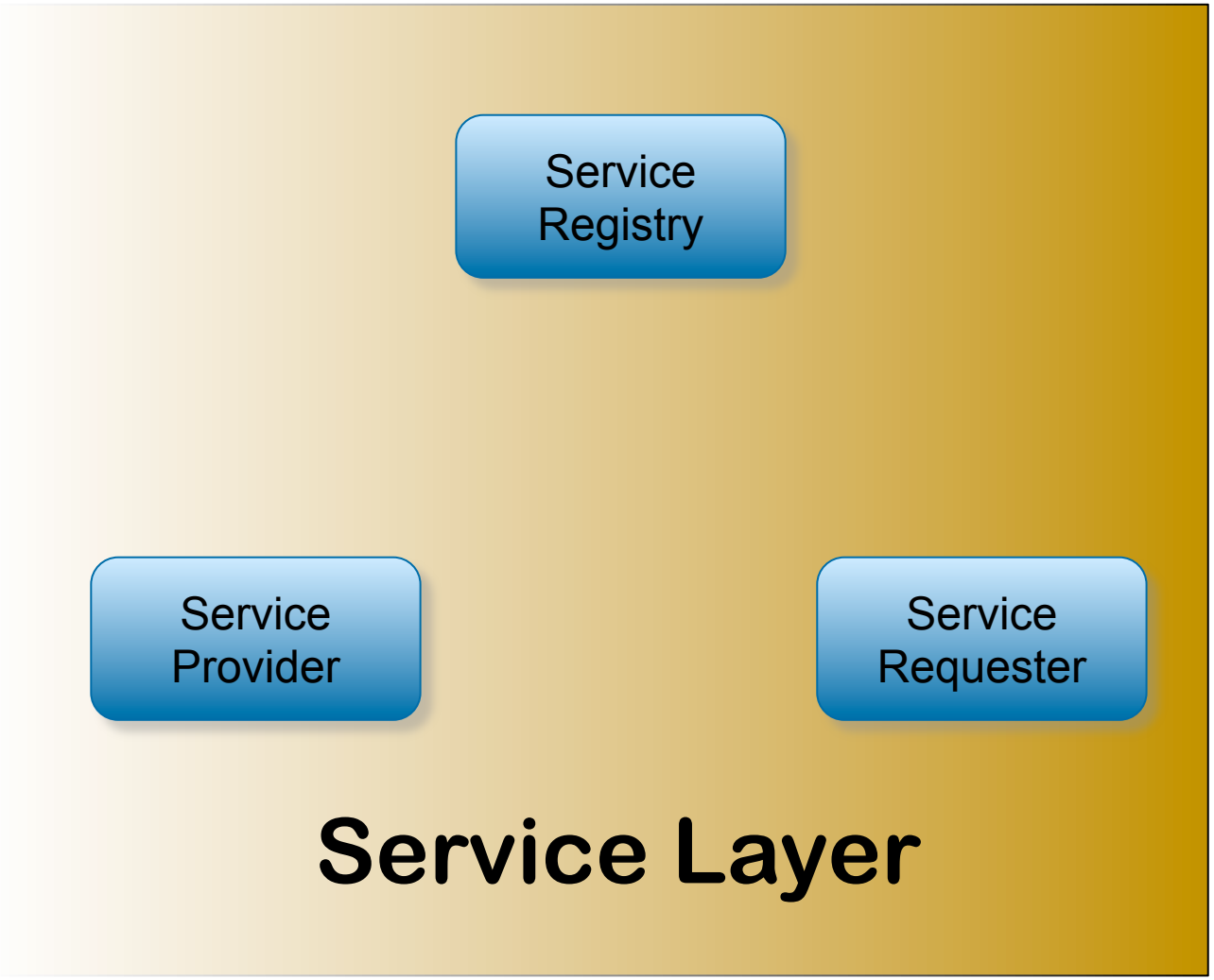
Consequently, the service layer is very lightweight, since service providers are typically just POJOs accessed via direct method invocation.

Additionally, the Service Layer allows bundles in the underlying Life Cycle Layer to appear or disappear in a completely dynamic manner to create a highly flexible, modular runtime system.



OSGi Architecture: Service Layer 2/2

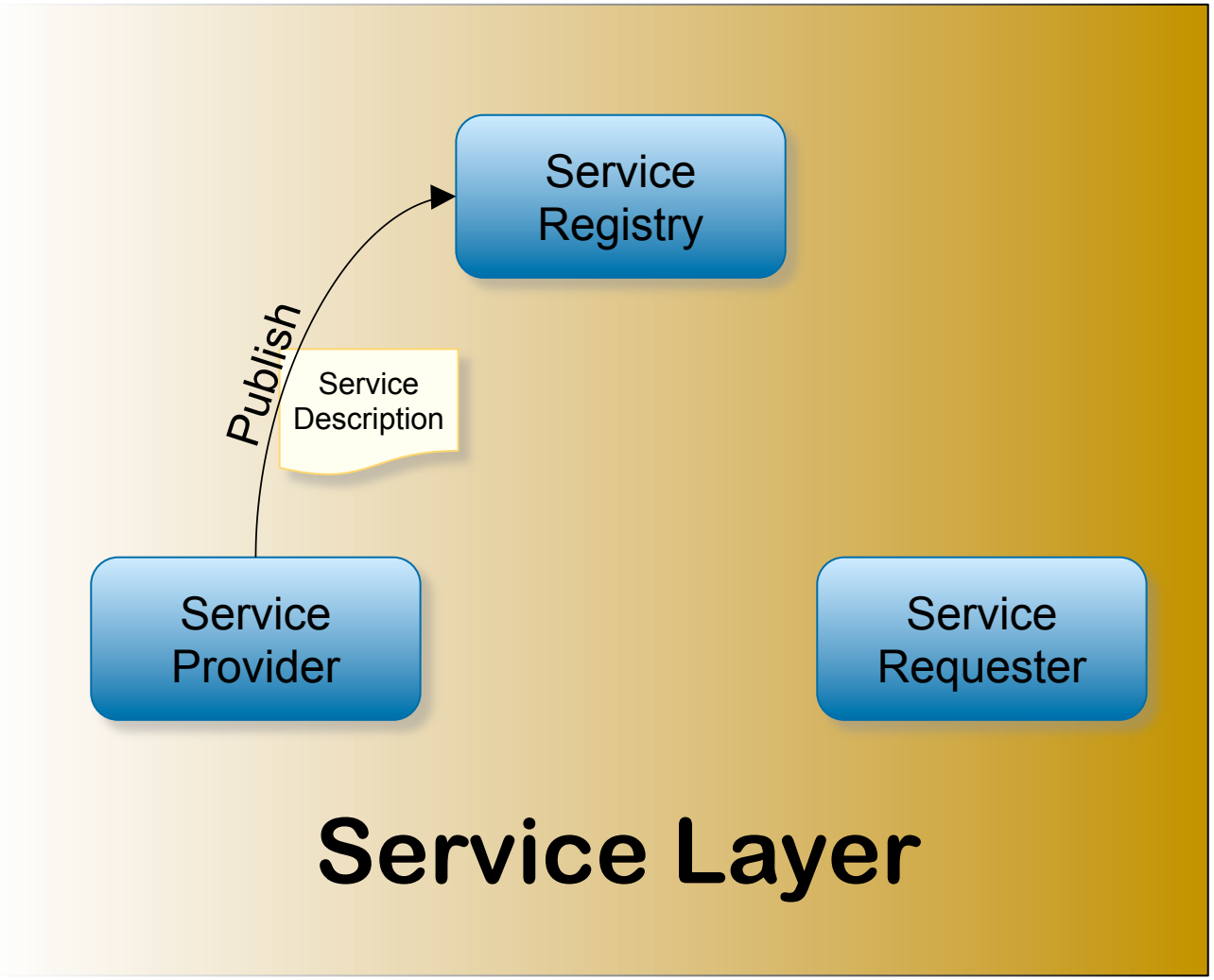
The **Publish-Find-Bind** interaction pattern is based on the following three stages:



OSGi Architecture: Service Layer 2/2

The **Publish-Find-Bind** interaction pattern is based on the following three stages:

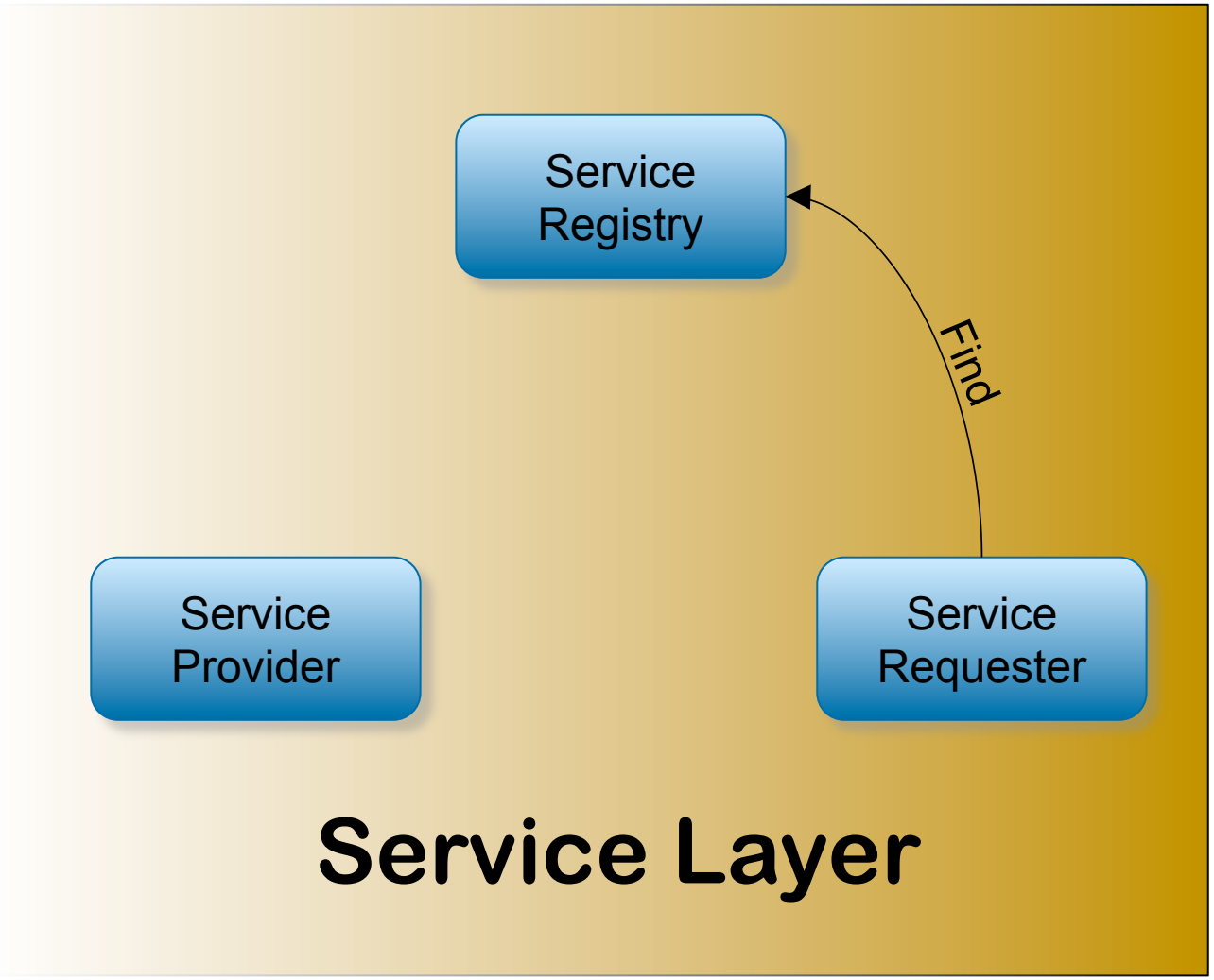
1. **Publish**
   The provider publishes a new service description into the Service Registry



OSGi Architecture: Service Layer 2/2

The **Publish-Find-Bind** interaction pattern is based on the following three stages:

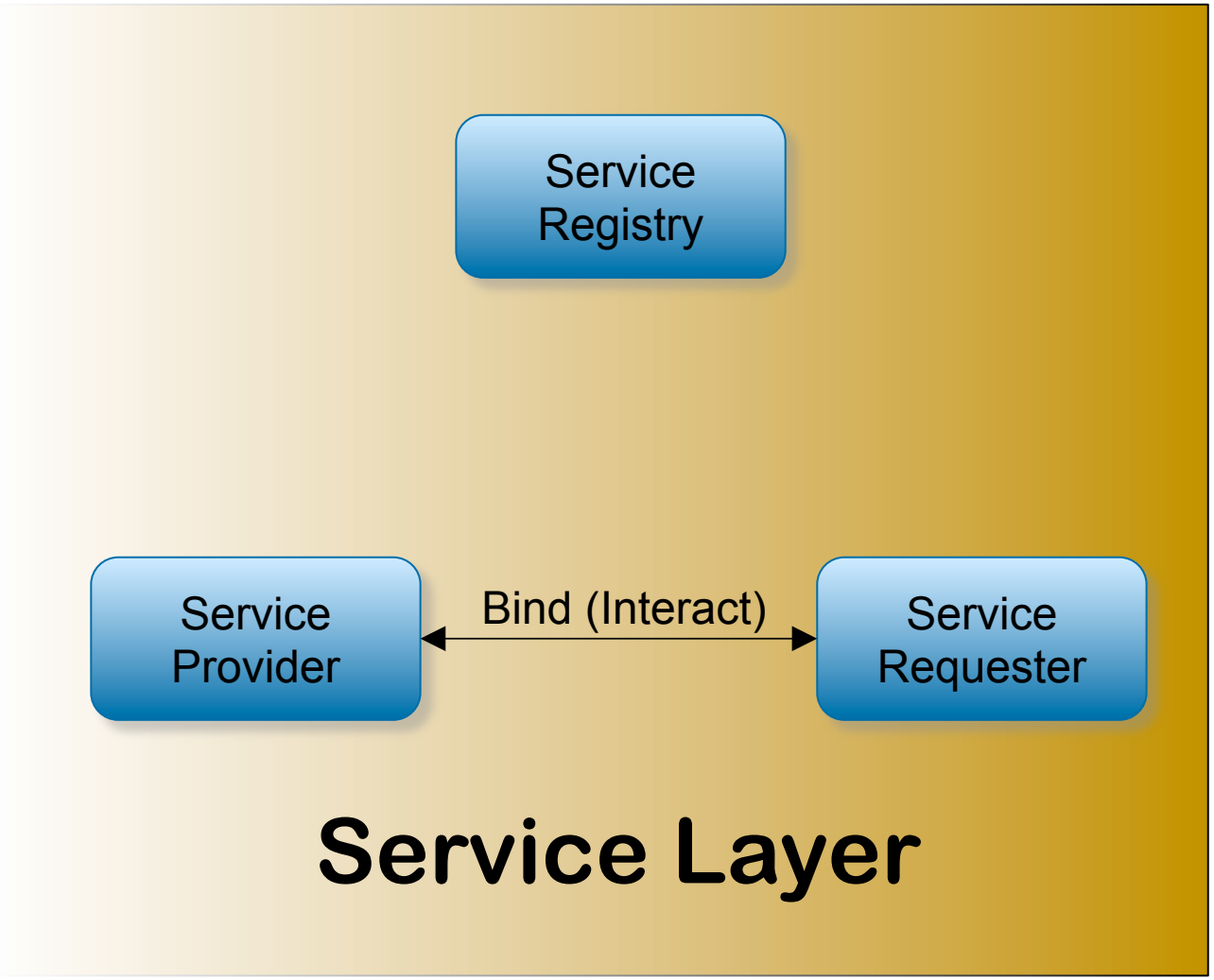
1. **Publish**
   The provider publishes a new service description into the Service Registry
2. **Find**
   The Service Requester then searches the repository for the required service



OSGi Architecture: Service Layer 2/2

The **Publish-Find-Bind** interaction pattern is based on the following three stages:

1. **Publish**
   The provider publishes a new service description into the Service Registry
2. **Find**
   The Service Requester then searches the repository for the required service
3. **Bind**
   If the required service exists in the repository, then the service requester interacts with it directly

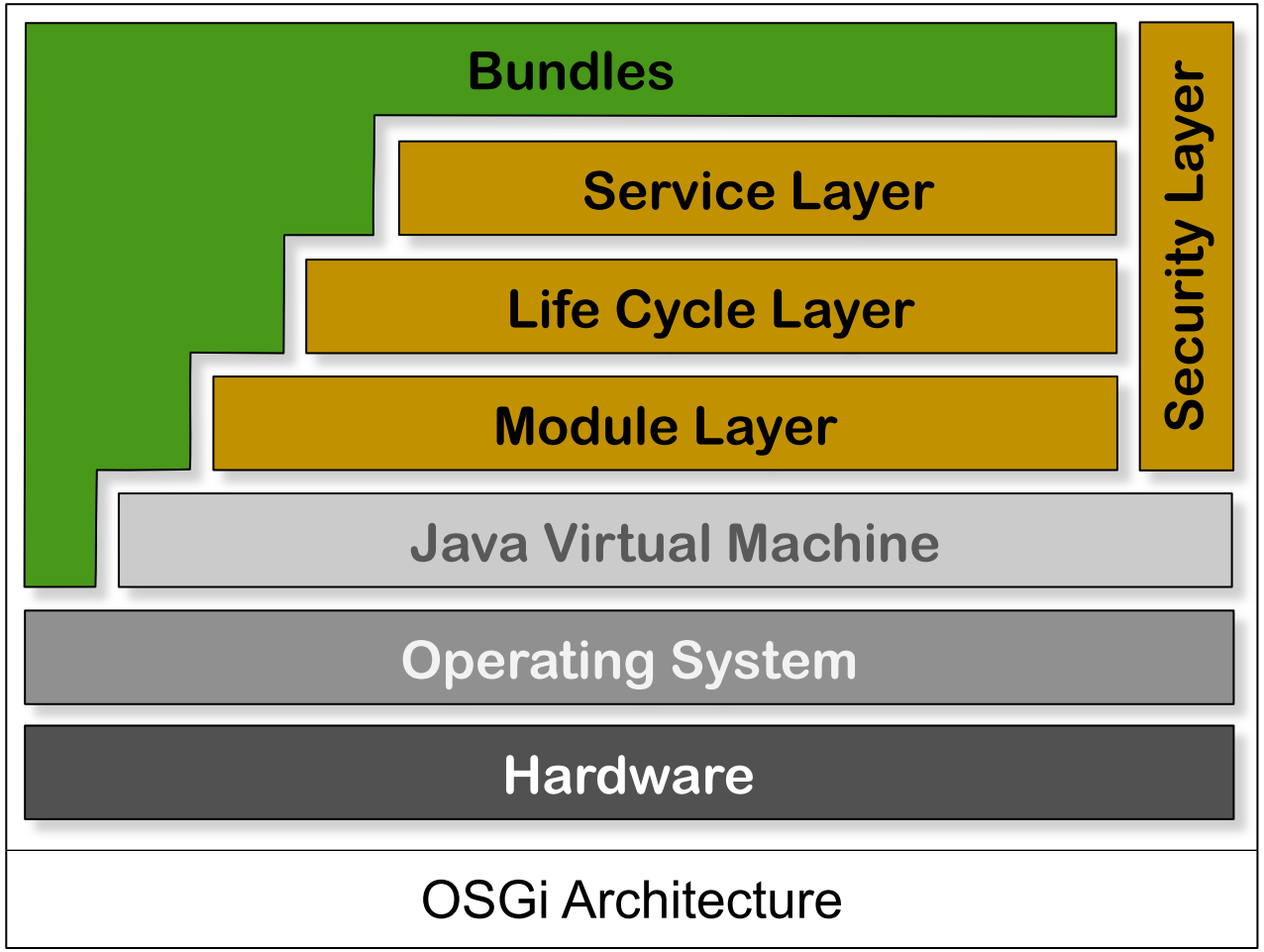


OSGi Architecture: Usage of OSGi Layers

The OSGi bundle and layer architecture ***does not*** require you to use all three layers. However, a top-down dependency does exist between the layers:

Service → Life Cycle → Module

You are free to start using OSGi at whichever layer meets your needs.



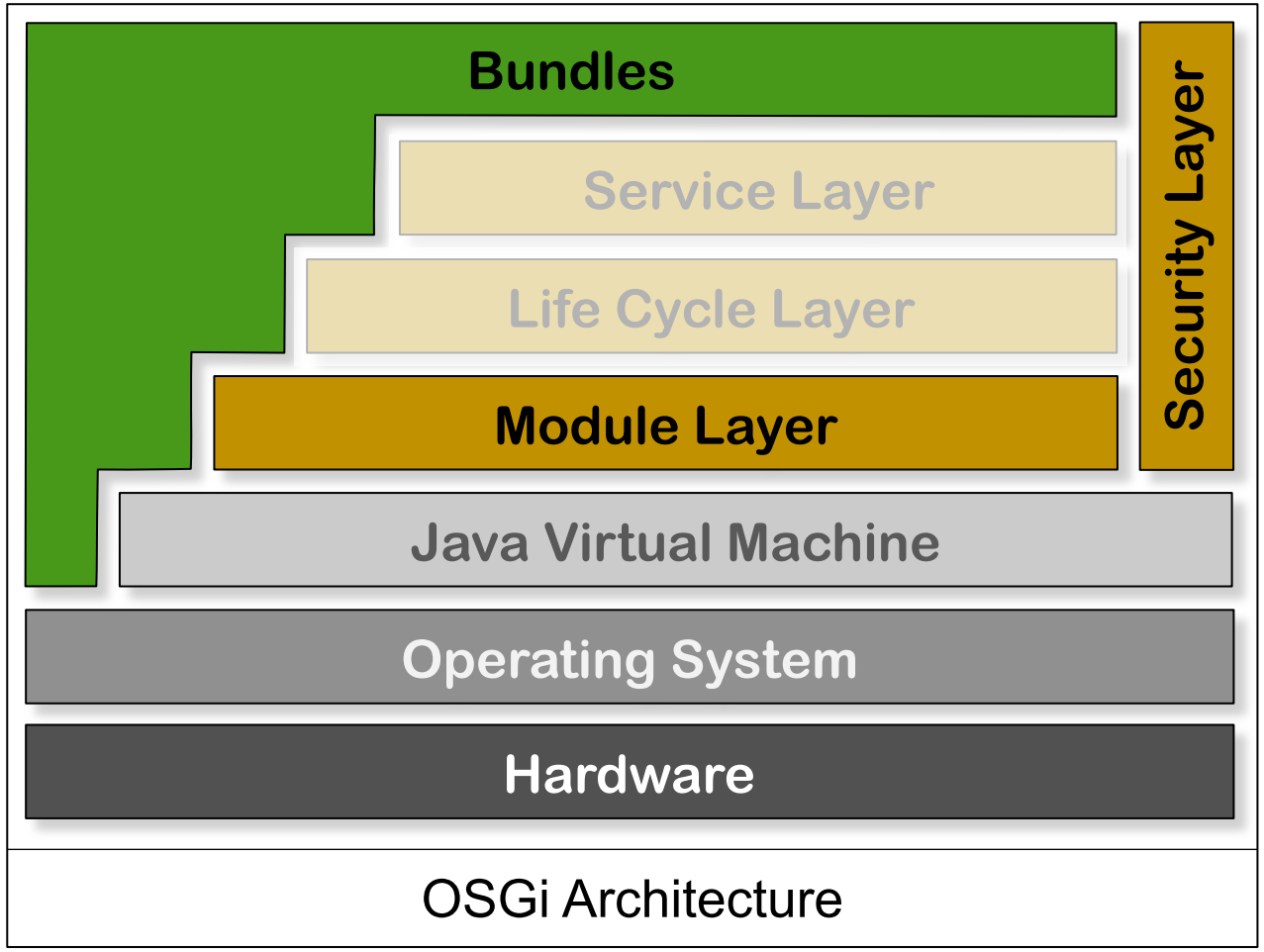
OSGi Architecture: Usage of OSGi Layers

The OSGi bundle and layer architecture ***does not*** require you to use all three layers. However, a top-down dependency does exist between the layers:

Service → Life Cycle → Module

You are free to start using OSGi at whichever layer meets your needs.

If you simply need better modularity, use only the Module Layer.



OSGi Architecture: Usage of OSGi Layers

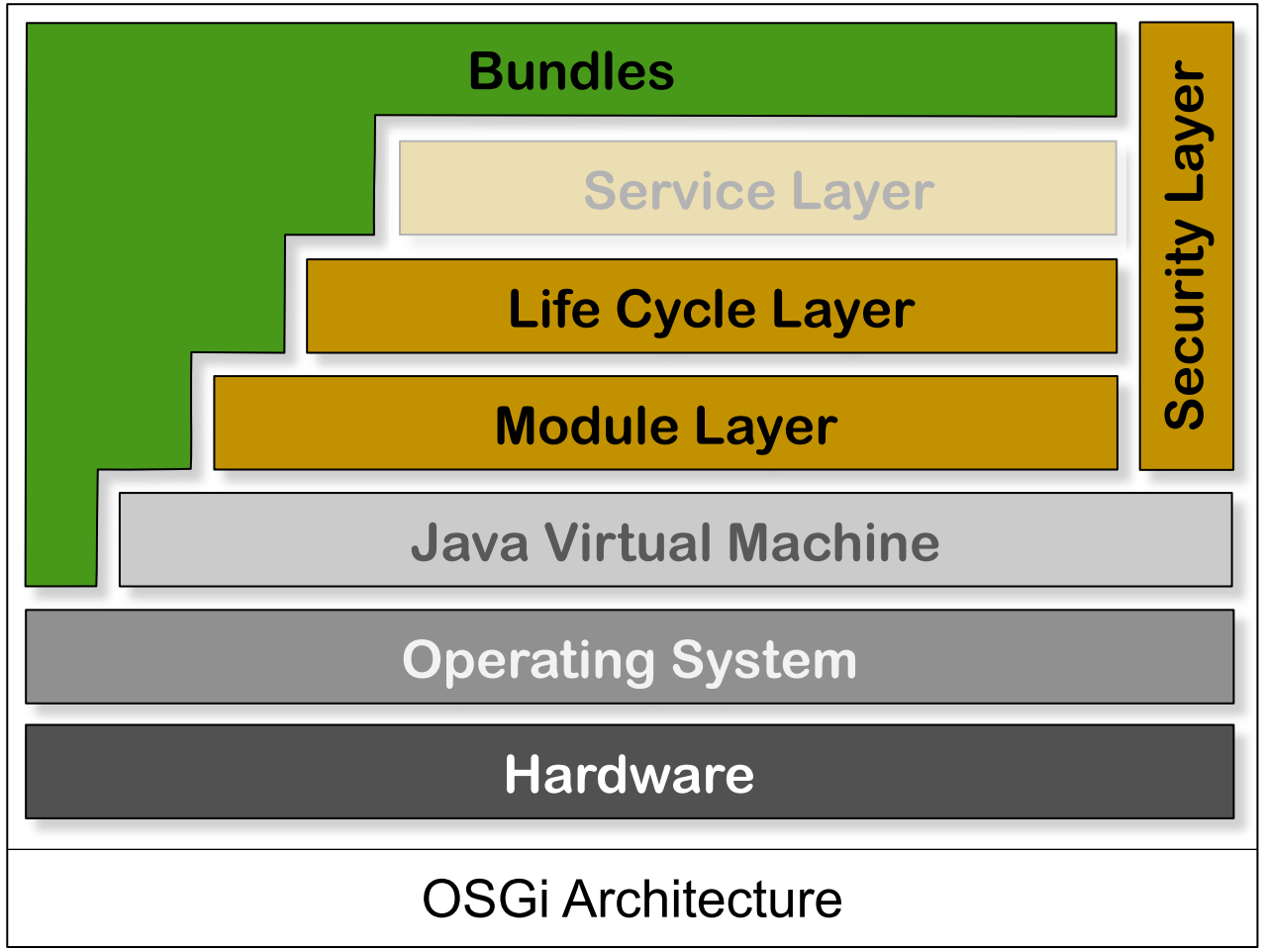
The OSGi bundle and layer architecture ***does not*** require you to use all three layers. However, a top-down dependency does exist between the layers:

Service → Life Cycle → Module

You are free to start using OSGi at whichever layer meets your needs.

If you simply need better modularity, use only the Module Layer.

If you want to administer and manage bundle lifecycles, then use the Life Cycle and Module Layers.



OSGi Architecture: Usage of OSGi Layers

The OSGi bundle and layer architecture ***does not*** require you to use all three layers. However, a top-down dependency does exist between the layers:

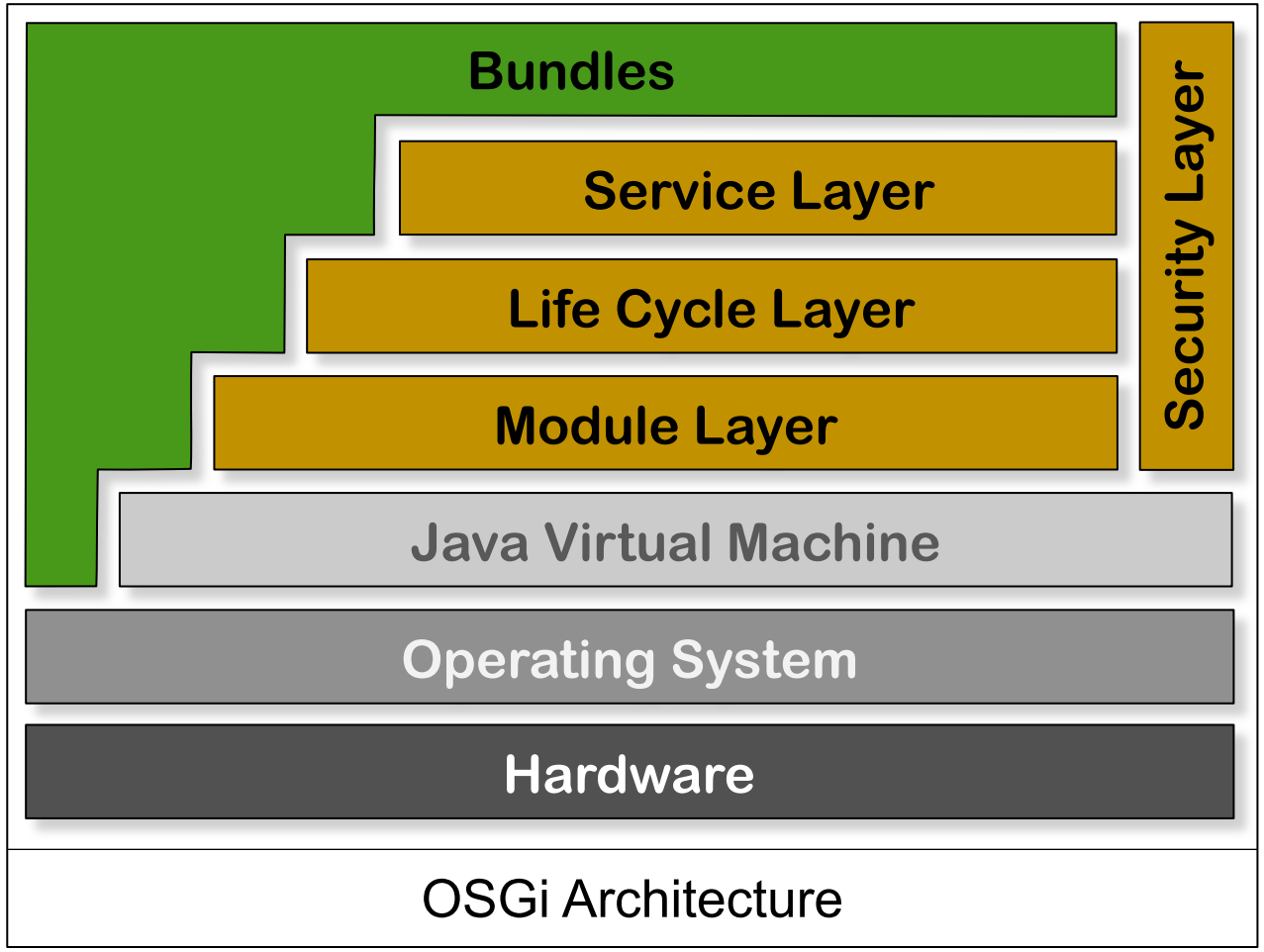
Service → Life Cycle → Module

You are free to start using OSGi at whichever layer meets your needs.

If you simply need better modularity, use only the Module Layer.

If you want to administer and manage bundle lifecycles, then use the Life Cycle and Module Layers.

If you want to use dynamic, interface-based development, then use all three layers.



OSGi Architecture

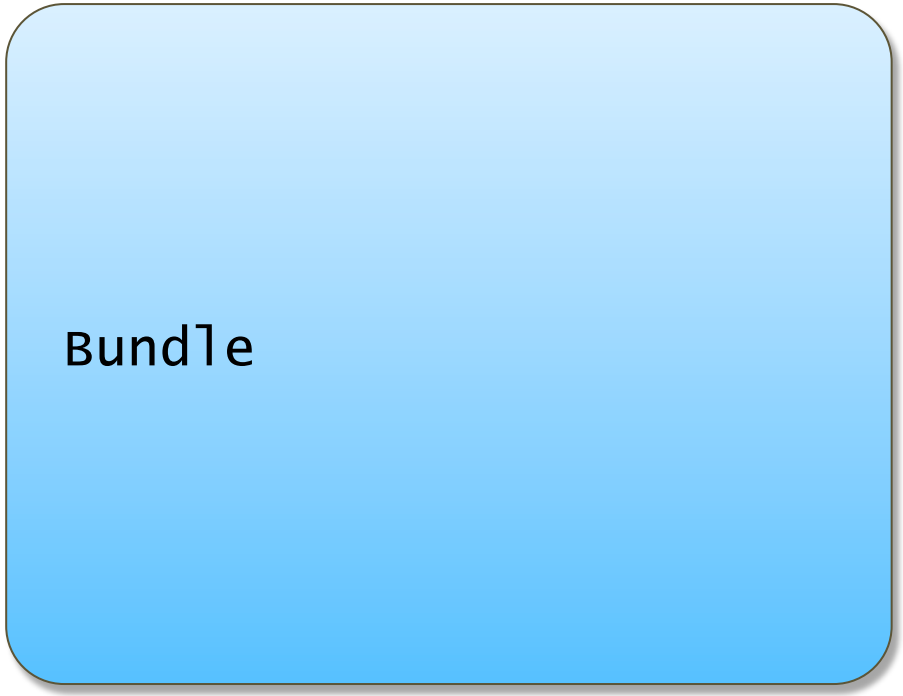
OSGi

The Module Concept and its Implementation in Bundles

The OSGi Module Concept

An OSGi module is a logical entity that defines the boundaries of a large-scale unit of functionality.

The OSGi module concept is implemented as an entity called a ***bundle***, where a bundle is simply:

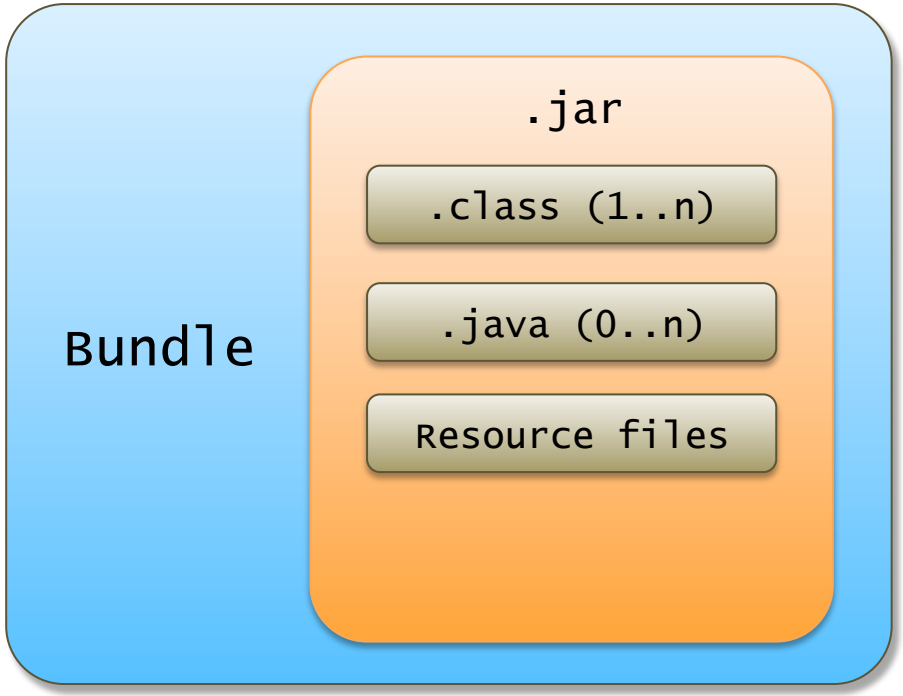


The OSGi Module Concept

An OSGi module is a logical entity that defines the boundaries of a large-scale unit of functionality.

The OSGi module concept is implemented as an entity called a ***bundle***, where a bundle is simply:

- A regular Java Archive file (`.jar`)

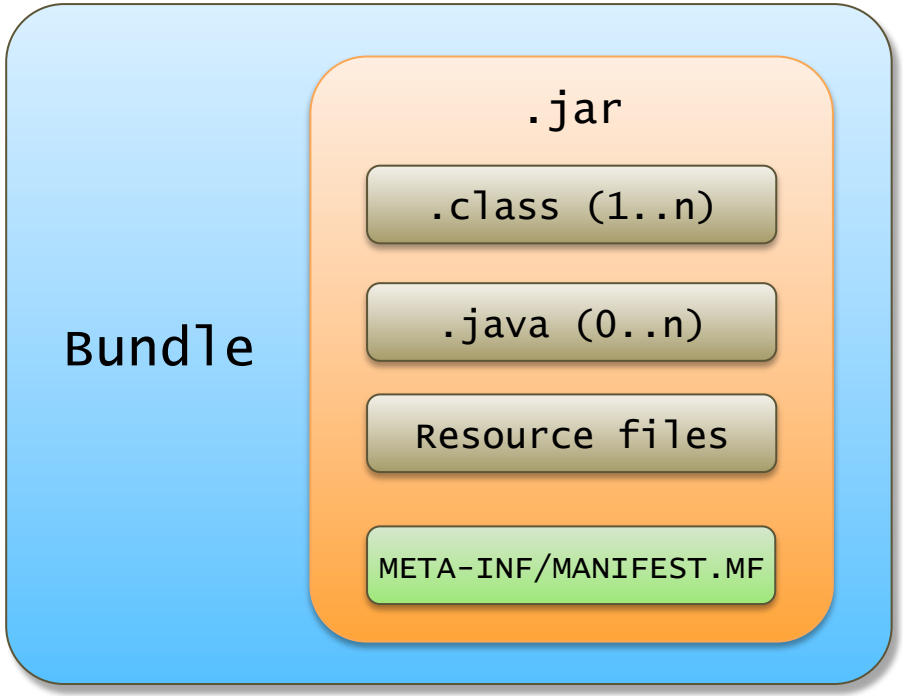


The OSGi Module Concept

An OSGi module is a logical entity that defines the boundaries of a large-scale unit of functionality.

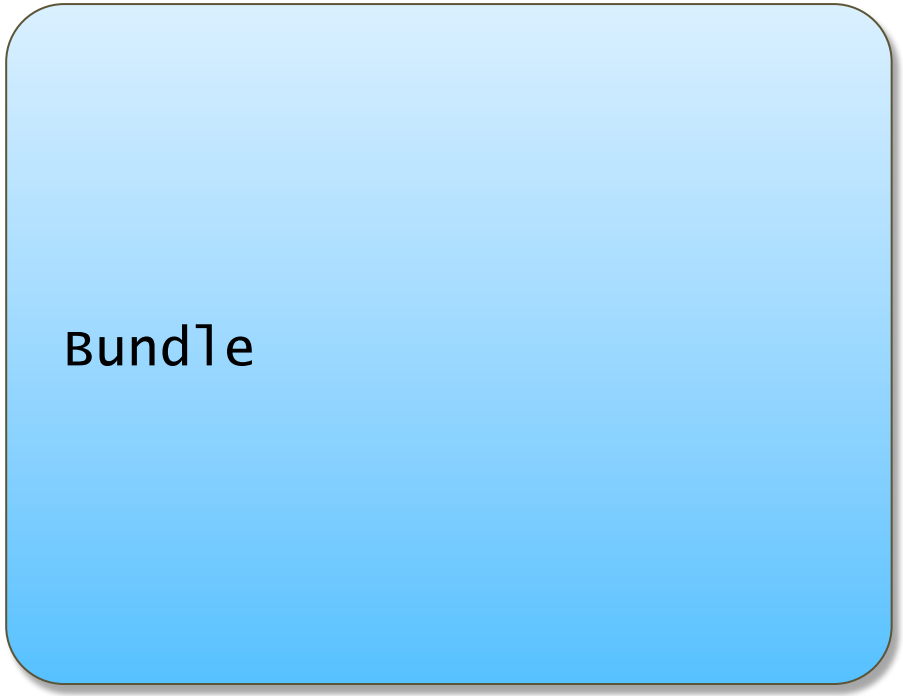
The OSGi module concept is implemented as an entity called a ***bundle***, where a bundle is simply:

- A regular Java Archive file (`.jar`)
- Supplemented by a metadata file stored in the JAR as `META-INF/MANIFEST.MF`



OSGi Bundles: Physical & Logical Modularity

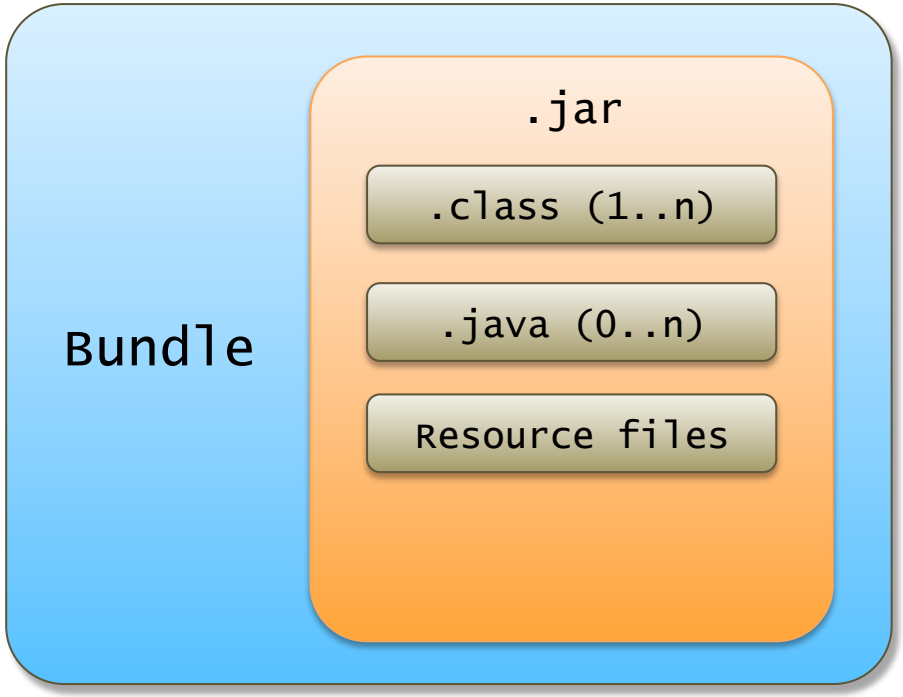
An OSGi bundle encapsulates 2 aspects of modularity:



OSGi Bundles: Physical & Logical Modularity

An OSGi bundle encapsulates 2 aspects of modularity:

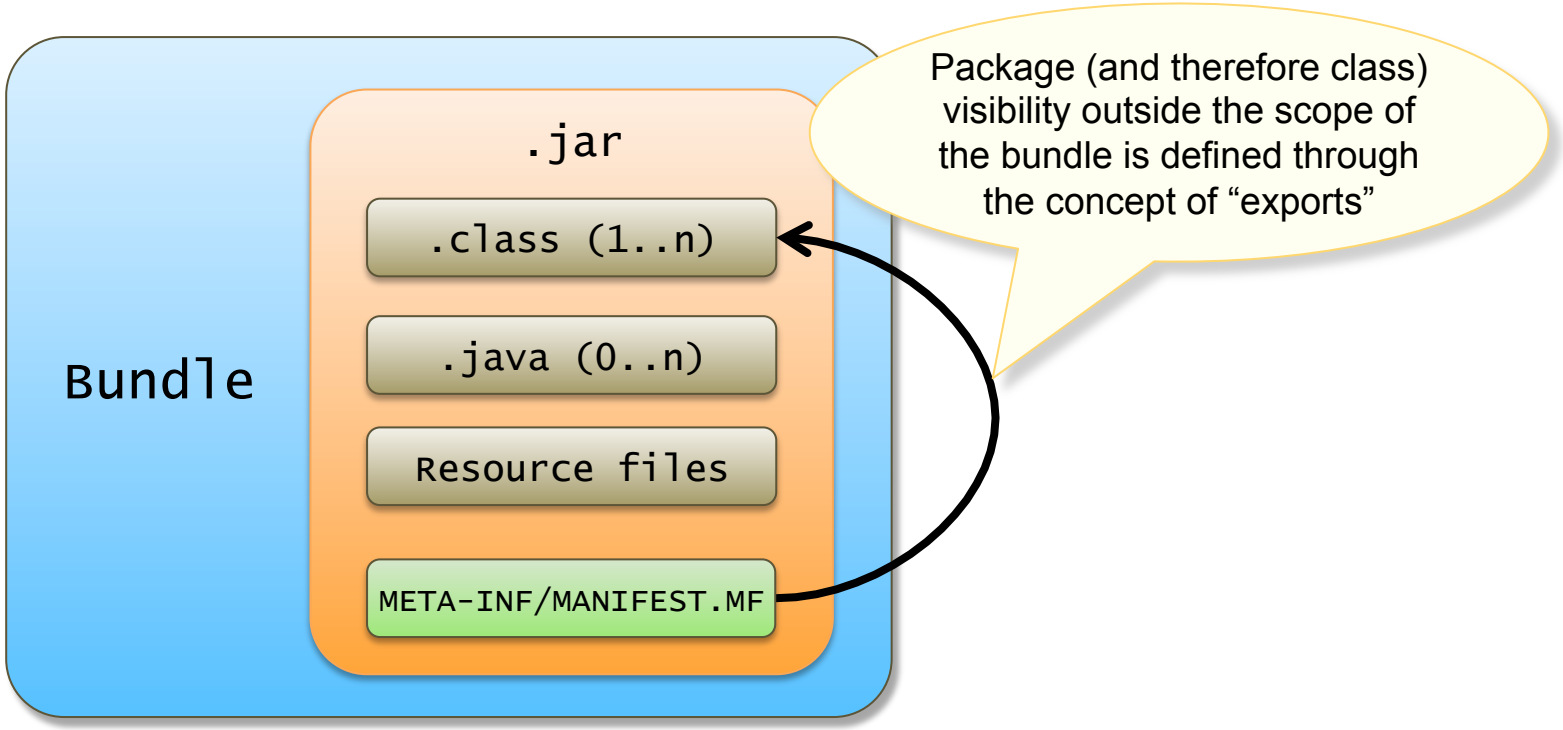
1. The physical unit of modularity required to deploy the software – I.E. the actual JAR file



OSGi Bundles: Physical & Logical Modularity

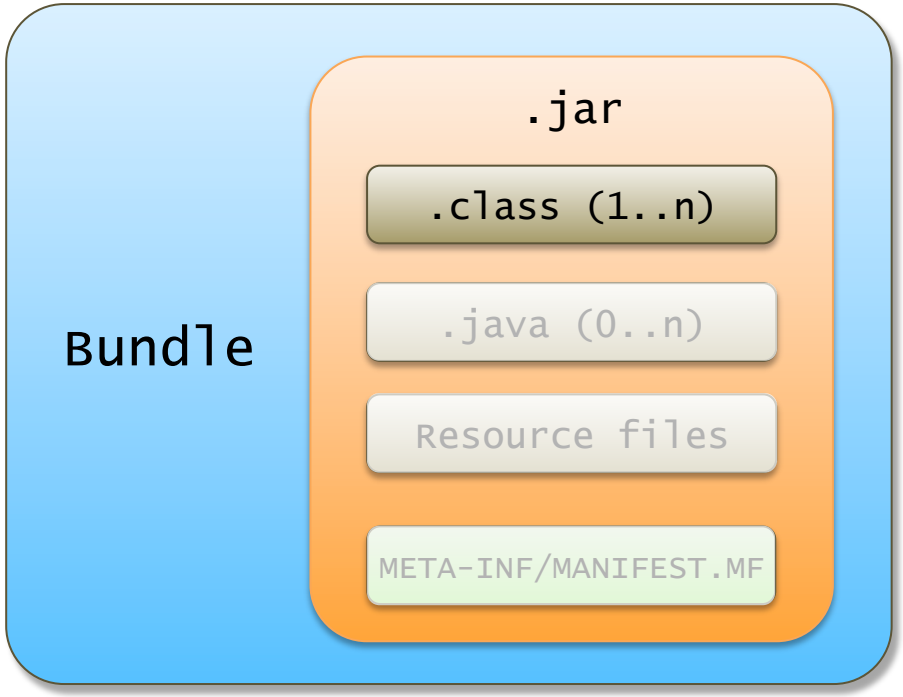
An OSGi bundle encapsulates 2 aspects of modularity:

1. The physical unit of modularity required to deploy the software – I.E. the actual JAR file
2. The logical unit of modularity required at runtime – I.E. the bundle metadata defines the visibility of the packages contained within the bundle



OSGi Bundles: Physical Modularity

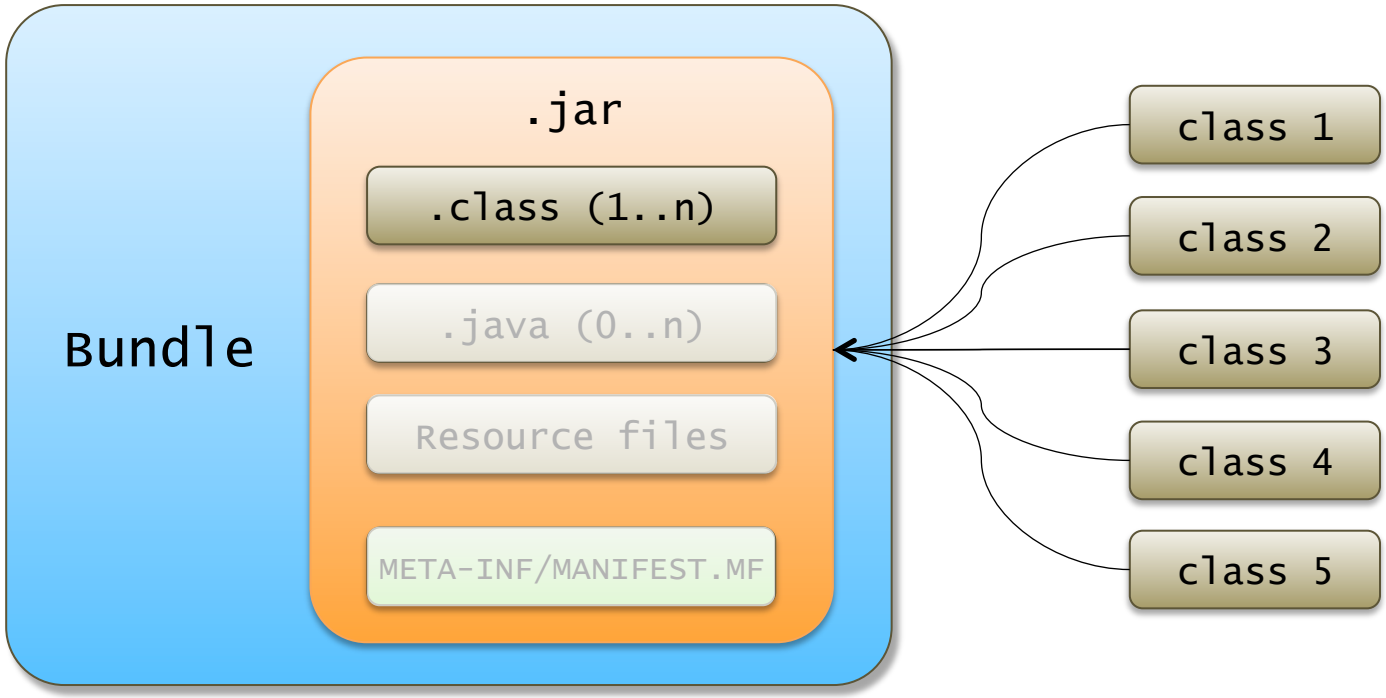
From the perspective of physical modularity, an OSGi bundle determines ***module membership***.



OSGi Bundles: Physical Modularity

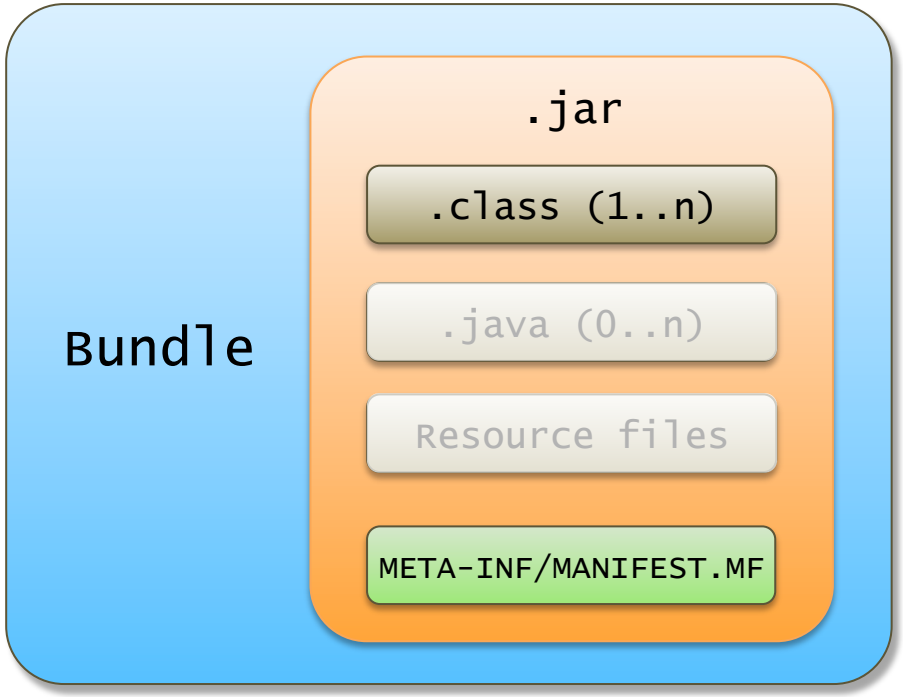
From the perspective of physical modularity, an OSGi bundle determines ***module membership***.

In other words, simply adding a Java class to the JAR file makes it a member of the bundle. The bundle (I.E. the JAR file + metadata) then becomes a distinct unit of deployment and administration.



OSGi Bundles: Logical Modularity

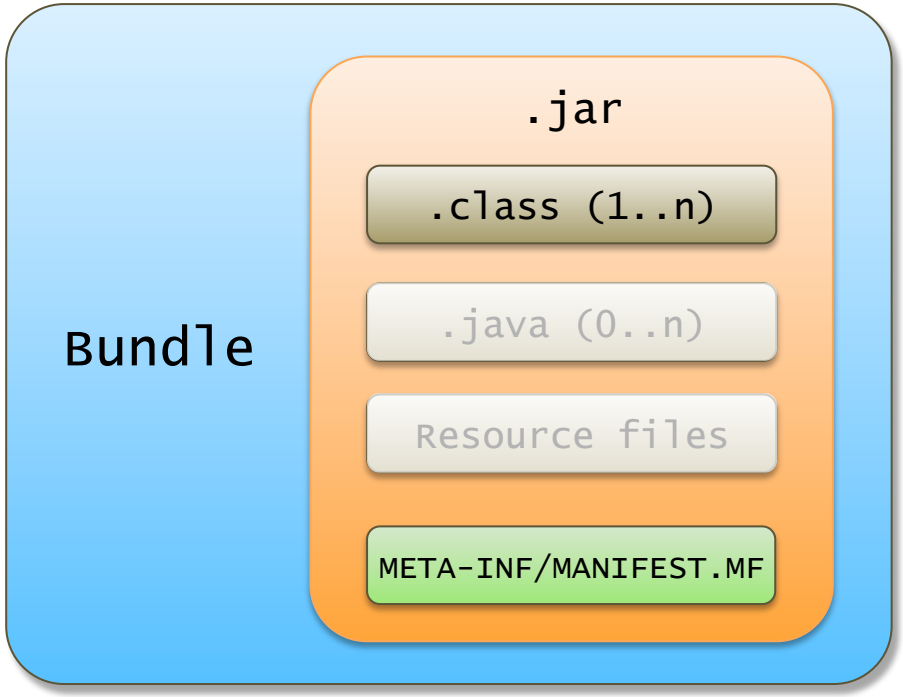
From the perspective of logical modularity, an OSGi bundle determines ***code visibility***. An OSGi bundle creates a logical boundary within which a Java class can be hidden, even though that class may have been defined as `public` within the package to which it belongs.



OSGi Bundles: Logical Modularity

From the perspective of logical modularity, an OSGi bundle determines ***code visibility***. An OSGi bundle creates a logical boundary within which a Java class can be hidden, even though that class may have been defined as `public` within the package to which it belongs.

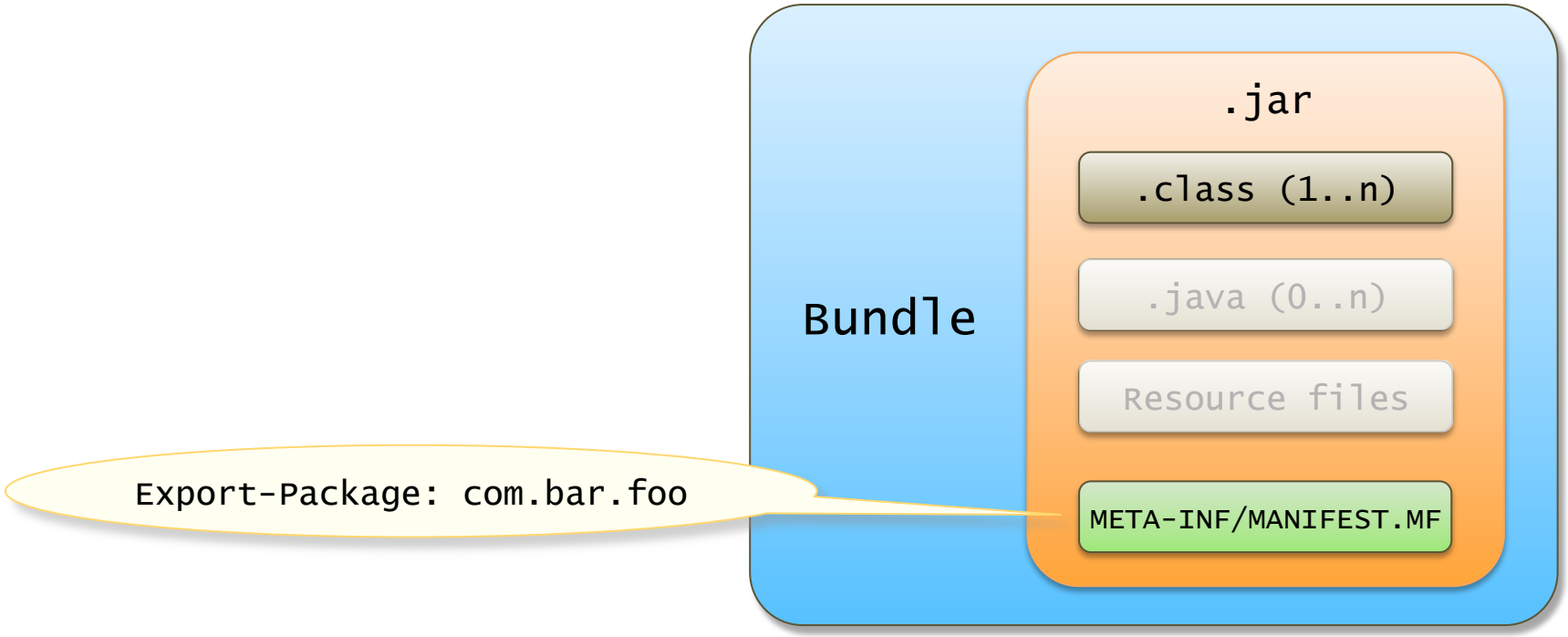
In other words, Java's concept of `public` has been extended so that we now have classes that are either "locally public" (I.E. visible only within the bundle) or "globally public" (I.E. visible outside the bundle).



OSGi Bundles: Logical Modularity

From the perspective of logical modularity, an OSGi bundle determines ***code visibility***. An OSGi bundle creates a logical boundary within which a Java class can be hidden, even though that class may have been defined as `public` within the package to which it belongs.

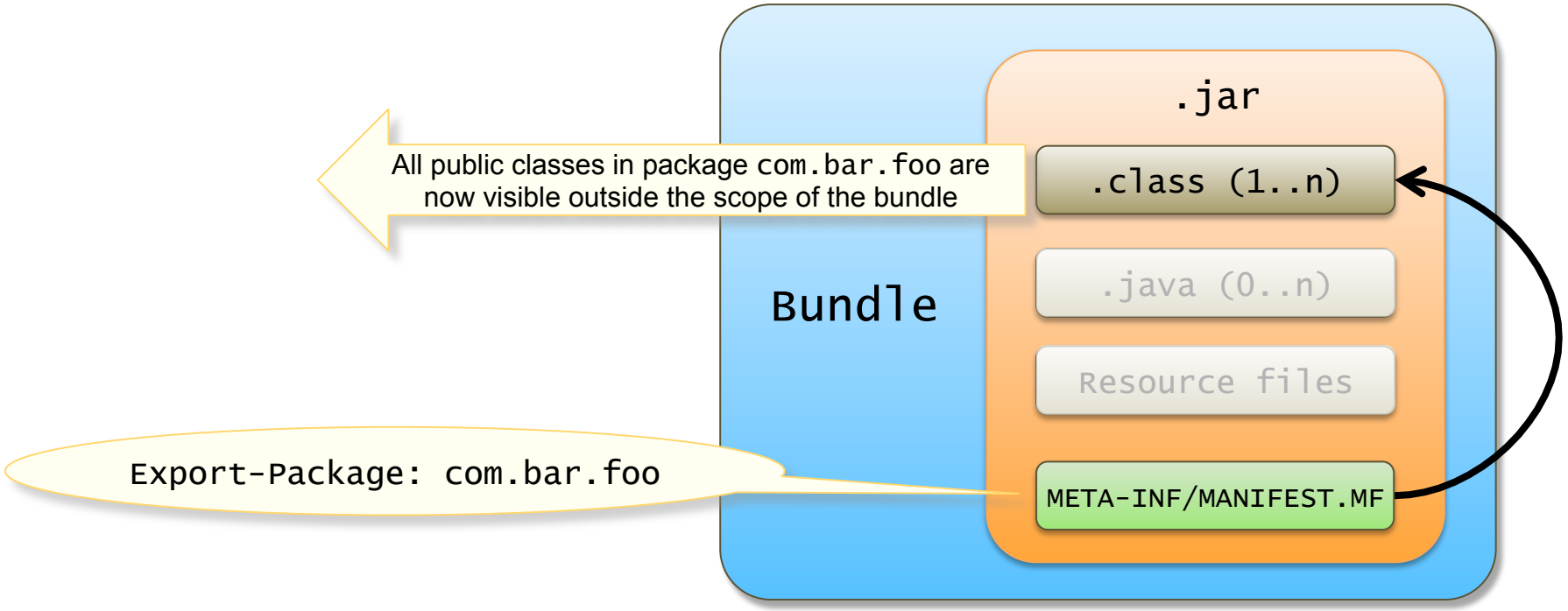
In other words, Java's concept of `public` has been extended so that we now have classes that are either "locally public" (I.E. visible only within the bundle) or "globally public" (I.E. visible outside the bundle).



OSGi Bundles: Logical Modularity

From the perspective of logical modularity, an OSGi bundle determines ***code visibility***. An OSGi bundle creates a logical boundary within which a Java class can be hidden, even though that class may have been defined as `public` within the package to which it belongs.

In other words, Java's concept of `public` has been extended so that we now have classes that are either "locally public" (I.E. visible only within the bundle) or "globally public" (I.E. visible outside the bundle).



OSGi

OSGi Metadata – The Syntax of `META-INF/MANIFEST.MF`

MANIFEST.MF: File Syntax 1/4

Bundle metadata properties are specified in plain text file in the form of <name>": "<value> pairs

```
Manifest-Version: 1.0
Created-By: 1.4 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.api
Bundle-Version: 1.0.0.SNAPSHOT
Bundle-Name: "Simple Paint API"
Export-Package: org.bar.foo.api
Import-Package: javax.swing
Bundle-License: http://www.opensource.org/licenses/apache2.0.php
```

MANIFEST.MF: File Syntax 1/4

Bundle metadata properties are specified in plain text file in the form of <name>": "<value> pairs

- The property name is ***not*** case-sensitive

```
Manifest-Version: 1.0
Created-By: 1.4 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.api
Bundle-Version: 1.0.0.SNAPSHOT
Bundle-Name: "Simple Paint API"
Export-Package: org.bar.foo.api
Import-Package: javax.swing
Bundle-License: http://www.opensource.org/licenses/apache2.0.php
```

MANIFEST.MF: File Syntax 1/4

Bundle metadata properties are specified in plain text file in the form of <name>": "<value> pairs

- The property name is ***not*** case-sensitive
- The property name is separated from the property value by a colon ***and*** a space character!

```
Manifest-Version: 1.0
Created-By: 1.4 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.api
Bundle-Version: 1.0.0.SNAPSHOT
Bundle-Name: "Simple Paint API"
Export-Package: org.bar.foo.api
Import-Package: javax.swing
Bundle-License: http://www.opensource.org/licenses/apache2.0.php
```

MANIFEST.MF: File Syntax 1/4

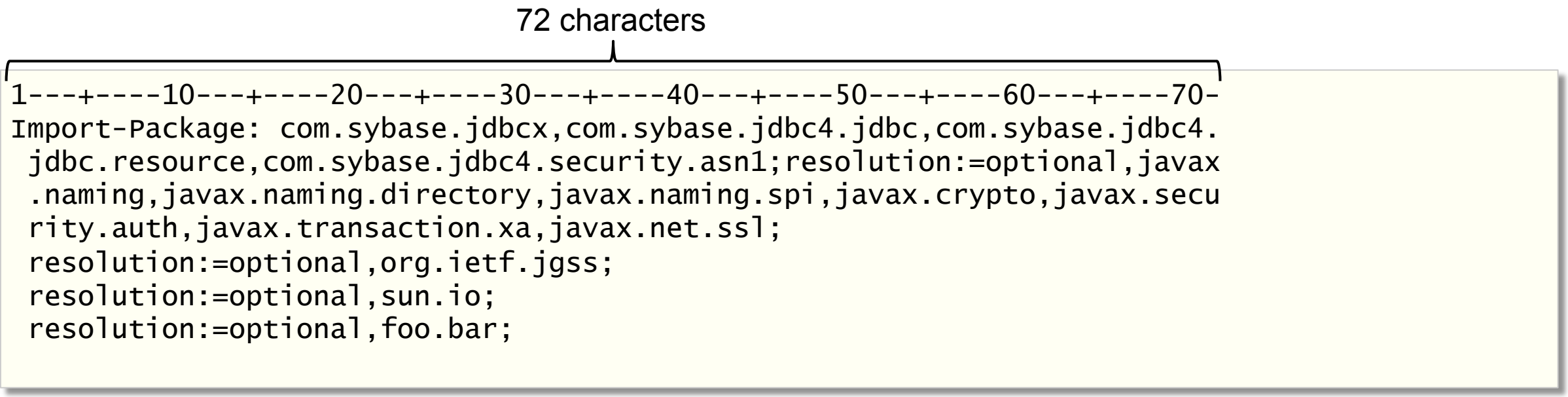
Bundle metadata properties are specified in plain text file in the form of <name>": "<value> pairs

- The property name is ***not*** case-sensitive
- The property name is separated from the property value by a colon ***and*** a space character!
- Property values containing whitespace should be delimited with double quote characters

```
Manifest-Version: 1.0
Created-By: 1.4 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.api
Bundle-Version: 1.0.0.SNAPSHOT
Bundle-Name: "Simple Paint API"
Export-Package: org.bar.foo.api
Import-Package: javax.swing
Bundle-License: http://www.opensource.org/licenses/apache2.0.php
```

MANIFEST.MF: File Syntax 2/4

A single line must not exceed 72 characters in length. If a property has a value longer than 72 characters, you can either:



MANIFEST.MF: File Syntax 2/4

A single line must not exceed 72 characters in length. If a property has a value longer than 72 characters, you can either:

- Simply split the property value at character 72 and continue on the next line, or

```
1---+----10---+----20---+----30---+----40---+----50---+----60---+----70-
Import-Package: com.sybase.jdbcx,com.sybase.jdbc4.jdbc,com.sybase.jdbc4.
 jdbc.resource,com.sybase.jdbc4.security.asn1;resolution:=optional,javax
 .naming,javax.naming.directory,javax.naming.spi,javax.crypto,javax.secu
 rity.auth,javax.transaction.xa,javax.net.ssl;
 resolution:=optional,org.ietf.jgss;
 resolution:=optional,sun.io;
 resolution:=optional,foo.bar;
```

MANIFEST.MF: File Syntax 2/4

A single line must not exceed 72 characters in length. If a property has a value longer than 72 characters, you can either:

- Simply split the property value at character 72 and continue on the next line, or
- Split the value at the natural break created by a property value separator

```
1---+----10---+----20---+----30---+----40---+----50---+----60---+----70-
Import-Package: com.sybase.jdbcx,com.sybase.jdbc4.jdbc,com.sybase.jdbc4.
 jdbc.resource,com.sybase.jdbc4.security.asn1;resolution:=optional,javax
 .naming,javax.naming.directory,javax.naming.spi,javax.crypto,javax.secu
 rity.auth,javax.transaction.xa,javax.net.ssl;
 resolution:=optional,org.ietf.jgss;
 resolution:=optional,sun.io;
 resolution:=optional,foo.bar;
```

MANIFEST.MF: File Syntax 2/4

A single line must not exceed 72 characters in length. If a property has a value longer than 72 characters, you can either:

- Simply split the property value at character 72 and continue on the next line, or
- Split the value at the natural break created by a property value separator

A continuation line ***must*** start with at least one space character

```
1---+----10---+----20---+----30---+----40---+----50---+----60---+----70-
Import-Package: com.sybase.jdbcx,com.sybase.jdbc4.jdbc,com.sybase.jdbc4.
 jdbc.resource,com.sybase.jdbc4.security.asn1;resolution:=optional,javax
 .naming,javax.naming.directory,javax.naming.spi,javax.crypto,javax.secu
 rity.auth,javax.transaction.xa,javax.net.ssl;
 resolution:=optional,org.ietf.jgss;
 resolution:=optional,sun.io;
 resolution:=optional,foo.bar;
```

MANIFEST.MF: File Syntax 3/4

OSGi property values can contain multiple clauses, each separated by a comma.

```
Property-Name: clause1, clause2, clause3
```

MANIFEST.MF: File Syntax 3/4

OSGi property values can contain multiple clauses, each separated by a comma.

Clauses can be further broken down into a value known as a ***target*** followed by parameter name/value pairs separated by semi-colons

```
Property-Name: clause1, clause2, clause3

Another-Property: target1; parameter1=value1; parameter2=value2,
 target2; parameter1=value1; parameter2=value2,
 target3; parameter1=value1; parameter2=value2
```

MANIFEST.MF: File Syntax 4/4

An OSGi property can be assigned any number of parameters. These parameters are of two types: attributes and directives

```
Attribute-Name: clause1, clause2, clause3

Another-Attribute: target1; dir1:=value1; attr1=value2,
 target2; dir1:=value1; attr1=value2,
 target3; dir1:=value1; attr1=value2
```

MANIFEST.MF: File Syntax 4/4

An OSGi property can be assigned any number of parameters. These parameters are of two types: attributes and directives

An ***attribute*** is an arbitrary value assigned to an arbitrary name

```
Attribute-Name: clause1, clause2, clause3

Another-Attribute: target1; dir1:=value1; attr1=value2,
 target2; dir1:=value1; attr1=value2,
 target3; dir1:=value1; attr1=value2
```

MANIFEST.MF: File Syntax 4/4

An OSGi property can be assigned any number of parameters. These parameters are of two types: attributes and directives

An ***attribute*** is an arbitrary value assigned to an arbitrary name

A ***directive*** is a specific keyword defined in the OSGi specification that alters the way the framework handles the associated information. The value is assigned using the "**:=**" syntax.

```
Attribute-Name: clause1, clause2, clause3

Another-Attribute: target1; dir1:=value1; attr1=value2,
 target2; dir1:=value1; attr1=value2,
 target3; dir1:=value1; attr1=value2
```

MANIFEST.MF: Properties Existing Only For Human Benefit

Some metadata properties exist for no other reason than to help humans understand what the bundle does and where it came from.

Since these properties have no meaning to the OSGi framework, you can store any values you wish in them

```
Bundle-Name: "Simple Paint Program"
Bundle-Description: "A simple paint program used to illustrate the use of OSGi"
Bundle-DocURL: http://www.somewebsite.com/osgi-docs/
Bundle-Category: example, library
Bundle-Vendor: "Some company"
Bundle-ContactAddress: "1234 Main Street, Dullsville, NE"
Bundle-Copyright: "My lawyer will call you"
```

MANIFEST.MF: Basic Required Properties

The OSGi framework requires a minimum set of properties in order to uniquely identify an OSGi bundle.

MANIFEST.MF: Basic Required Properties

The OSGi framework requires a minimum set of properties in order to uniquely identify an OSGi bundle.

From OSGi R4 (Oct 2005) onwards, bundles must firstly specify that they use the new manifest syntax, and secondly, all bundles must be uniquely identified.

```
Bundle-ManifestVersion: 2
```

The `Bundle-ManifestVersion` must have the value “2”

MANIFEST.MF: Basic Required Properties

The OSGi framework requires a minimum set of properties in order to uniquely identify an OSGi bundle.

From OSGi R4 (Oct 2005) onwards, bundles must firstly specify that they use the new manifest syntax, and secondly, all bundles must be uniquely identified.

However prior to R4, the human readable property `Bundle-Name` was already used to provide a human-readable label. So a new property called ***`Bundle-SymbolicName`*** was created.

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.myapp
```

Do not confuse `Bundle-Name` with `Bundle-SymbolicName`!

MANIFEST.MF: Basic Required Properties

The OSGi framework requires a minimum set of properties in order to uniquely identify an OSGi bundle.

From OSGi R4 (Oct 2005) onwards, bundles must firstly specify that they use the new manifest syntax, and secondly, all bundles must be uniquely identified.

However prior to R4, the human readable property `Bundle-Name` was already used to provide a human-readable label. So a new property called ***`Bundle-SymbolicName`*** was created.

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.myapp
```

The value of the `Bundle-SymbolicName` is a dot-separated character string that follows the Java reverse domain name convention.

MANIFEST.MF: Basic Required Properties

The OSGi framework requires a minimum set of properties in order to uniquely identify an OSGi bundle.

From OSGi R4 (Oct 2005) onwards, bundles must firstly specify that they use the new manifest syntax, and secondly, all bundles must be uniquely identified.

However prior to R4, the human readable property `Bundle-Name` was already used to provide a human-readable label. So a new property called ***`Bundle-SymbolicName`*** was created.

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.myapp
Bundle-Version: 1.0.0
```

The value of the `Bundle-SymbolicName` is a dot-separated character string that follows the Java reverse domain name convention.

The property `Bundle-Version` must also be used to specify the current version of the bundle.

MANIFEST.MF: Basic Required Properties

The OSGi framework requires a minimum set of properties in order to uniquely identify an OSGi bundle.

From OSGi R4 (Oct 2005) onwards, bundles must firstly specify that they use the new manifest syntax, and secondly, all bundles must be uniquely identified.

However prior to R4, the human readable property `Bundle-Name` was already used to provide a human-readable label. So a new property called ***`Bundle-SymbolicName`*** was created.

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.myapp
Bundle-Version: 1.0.0
```

The value of the `Bundle-SymbolicName` is a dot-separated character string that follows the Java reverse domain name convention.

The property `Bundle-Version` must also be used to specify the current version of the bundle.

An OSGi bundle is uniquely identified using both `Bundle-SymbolicName` ***and*** `Bundle-Version`.

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

```
Bundle-Version:
```

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

- The Major Number

```
Bundle-Version: 1
```

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

- The Major Number
- The Minor Number

```
Bundle-Version: 1.0
```

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

- The Major Number
- The Minor Number
- The Micro Number

```
Bundle-Version: 1.0.0
```

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

- The Major Number
- The Minor Number
- The Micro Number

```
Bundle-Version: 1.0.0.alpha
```

An optional fourth value can be added called a “qualifier”. This is an alphanumeric character string.

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

- The Major Number
- The Minor Number
- The Micro Number

```
Bundle-Version: 1.0.0.alpha
```

An optional fourth value can be added called a “qualifier”. This is an alphanumeric character string.

If the entire version number is omitted, it will default to `0.0.0`

MANIFEST.MF: Bundle Version Numbers

An OSGi bundle version number is composed of three, dot-separated numbers.

- The Major Number
- The Minor Number
- The Micro Number

```
Bundle-Version: 1.0.0.alpha
```

An optional fourth value can be added called a “qualifier”. This is an alphanumeric character string.

If the entire version number is omitted, it will default to `0.0.0`

Also, version numbers are compared using string comparison! So this makes version `0.5.1.fourth` older than version `0.5.1.third` because “f” comes before “t” in the alphabet.

MANIFEST.MF: Code Visibility – Internal to a Bundle

In a standard JAR file, all directories from the root downwards are implicitly searched during the class lookup process. OSGi however, uses a more explicit approach by defining a property called `Bundle-ClassPath`.

Just like a standard Java class path, the `Bundle-ClassPath` defines a list of locations to search, but it refers ***only*** to locations inside the bundle JAR file.

```
Bundle-ClassPath: search-here-first/, then-here/, finally_in_the_embedded.jar
```

MANIFEST.MF: Code Visibility – Internal to a Bundle

In a standard JAR file, all directories from the root downwards are implicitly searched during the class lookup process. OSGi however, uses a more explicit approach by defining a property called `Bundle-ClassPath`.

Just like a standard Java class path, the `Bundle-ClassPath` defines a list of locations to search, but it refers ***only*** to locations inside the bundle JAR file.

```
Bundle-ClassPath: search-here-first/, then-here/, finally_in_the_embedded.jar
```

The `Bundle-ClassPath` property is unusual for OSGi properties in that if you do not specify a value for this property, then it will default to “.” Making “.” the default ensures that both standard and bundle JAR files have the same internal search policy.

MANIFEST.MF: Code Visibility – Internal to a Bundle

In a standard JAR file, all directories from the root downwards are implicitly searched during the class lookup process. OSGi however, uses a more explicit approach by defining a property called `Bundle-ClassPath`.

Just like a standard Java class path, the `Bundle-ClassPath` defines a list of locations to search, but it refers ***only*** to locations inside the bundle JAR file.

```
Bundle-ClassPath: search-here-first/, then-here/, finally_in_the_embedded.jar
```

The `Bundle-ClassPath` property is unusual for OSGi properties in that if you do not specify a value for this property, then it will default to “.” Making “.” the default ensures that both standard and bundle JAR files have the same internal search policy.

IMPORTANT

The default value “.” is only used if ***no*** `Bundle-ClassPath` value is specified. If you specify your own value that does not include “.”, then the root relative directories will ***not*** be searched.

MANIFEST.MF: Code Visibility – External to a Bundle

Code is shared between bundles at the Java package level rather than at the level of an entire bundle.

By default, an OSGi bundle takes a ***shares nothing*** approach. Therefore, any package that needs to be visible outside the scope of the bundle must be explicitly exported.

If a package is exported from a bundle, then by definition, all the public classes of that package are visible outside the scope of the bundle.

```
Export-Package: org.bar.foo.myapp, org.bar.foo.utils
```

The value of the `Export-Package` property is a comma separated list of Java package names.

MANIFEST.MF: The Export-Package Property

It is possible that two OSGi bundles might export the same Java package; however, this is no guarantee that the coding in the two packages is functionally equivalent.

Therefore, extra attributes can be added to the `Export-Package` property to further qualify the exact identity of the exported package. These attributes can either be your own arbitrary name/value pairs, or predefined names found in the OSGi specification.

```
Export-Package: org.bar.foo.myapp; vendor="My company",
 org.bar.foo.utils; version=2.1.3
```

MANIFEST.MF: The Export-Package Property

It is possible that two OSGi bundles might export the same Java package; however, this is no guarantee that the coding in the two packages is functionally equivalent.

Therefore, extra attributes can be added to the `Export-Package` property to further qualify the exact identity of the exported package. These attributes can either be your own arbitrary name/value pairs, or predefined names found in the OSGi specification.

```
Export-Package: org.bar.foo.myapp; vendor="My company",
 org.bar.foo.utils; version=2.1.3
```

In this example, the package `org.bar.foo.myapp` is qualified with the attribute “vendor”. This is a custom attribute that has no meaning to the OSGi framework.

MANIFEST.MF: The Export-Package Property

It is possible that two OSGi bundles might export the same Java package; however, this is no guarantee that the coding in the two packages is functionally equivalent.

Therefore, extra attributes can be added to the `Export-Package` property to further qualify the exact identity of the exported package. These attributes can either be your own arbitrary name/value pairs, or predefined names found in the OSGi specification.

```
Export-Package: org.bar.foo.myapp; vendor="My company",
 org.bar.foo.utils; version=2.1.3
```

In this example, the package `org.bar.foo.myapp` is qualified with the attribute “vendor”. This is a custom attribute that has no meaning to the OSGi framework.

However, the package `org.bar.foo.utils` is also qualified with a specific version number that is meaningful to the OSGi framework.

MANIFEST.MF: Defining External Package Dependencies

An OSGi bundle exports some of the Java packages it contains because other bundles will need to use this functionality.

Therefore, the `Import-Package` property is used to declare the packages upon which the coding in your bundle is dependant.

```
Import-Package: org.bar.foo.utils, org.bar.foo.stuff
```

The `Import-Package` property value is a comma separated list of Java package names.

MANIFEST.MF: Defining External Package Dependencies

An OSGi bundle exports some of the Java packages it contains because other bundles will need to use this functionality.

Therefore, the `Import-Package` property is used to declare the packages upon which the coding in your bundle is dependant.

```
Import-Package: org.bar.foo.utils, org.bar.foo.stuff
```

The `Import-Package` property value is a comma separated list of Java package names.

It is the responsibility of the OSGi framework to resolve package dependencies before starting to execute any application code.

MANIFEST.MF: The Import-Package Property

We have already seen that extra attributes can be used to qualify the packages exported from a bundle.

The same attributes can then be used by the `Import-Package` property as extra qualifiers that must match before a package is chosen for import.

```
Import-Package: org.bar.foo.utils; version=2.1.3,
 org.bar.foo.stuff; vendor="My company"
```

MANIFEST.MF: The Import-Package Property

We have already seen that extra attributes can be used to qualify the packages exported from a bundle.

The same attributes can then be used by the `Import-Package` property as extra qualifiers that must match before a package is chosen for import.

```
Import-Package: org.bar.foo.utils; version=2.1.3,
 org.bar.foo.stuff; vendor="My company"
```

For arbitrary attributes such as “vendor”, the value is simply tested for exact equality.

MANIFEST.MF: The Import-Package Property

We have already seen that extra attributes can be used to qualify the packages exported from a bundle.

The same attributes can then be used by the `Import-Package` property as extra qualifiers that must match before a package is chosen for import.

```
Import-Package: org.bar.foo.utils; version=2.1.3,
 org.bar.foo.stuff; vendor="My company"
```

For arbitrary attributes such as “vendor”, the value is simply tested for exact equality.

However, for OSGi specified attributes such as “version”, more flexibility is available for value comparison.

MANIFEST.MF: Importing the Correct Package Version

In the example below, we want to import version 2.1.3 of package `org.bar.foo.utils`.

However for historical reasons, the OSGi framework has always assumed that future versions of a package will be backward compatible and therefore considers the version number (as shown here) to be the ***minimum required*** version. So OSGi interprets this import clause to mean:

"*Import any version of package `org.bar.foo.utils` **greater than or equal** to 2.1.3*"

```
Import-Package: org.bar.foo.utils; version=2.1.3,
 org.bar.foo.stuff; vendor="My company"
```

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets “[ ]” for an inclusive match and parentheses “( )” for an exclusive match.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets “[ ]” for an inclusive match and parentheses “( )” for an exclusive match.

The value of the version attribute now defines that any package between versions 2.1.0 and 2.1.3 ***inclusive*** is a suitable import candidate.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets "[ ]" for an inclusive match and parentheses "( )" for an exclusive match.

The value of the version attribute now defines that any package between versions 2.1.0 and 2.1.3 ***inclusive*** is a suitable import candidate.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

For any particular version `vers`, version ranges are defined as follows:

`(min,max)` means `min < vers < max`

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets "`[ ]`" for an inclusive match and parentheses "`( )`" for an exclusive match.

The value of the version attribute now defines that any package between versions 2.1.0 and 2.1.3 ***inclusive*** is a suitable import candidate.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

For any particular version `vers`, version ranges are defined as follows:

```
(min,max)    means   min <  vers < max
(min, max]   means   min <  vers <= max
```

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets "[ ]" for an inclusive match and parentheses "( )" for an exclusive match.

The value of the version attribute now defines that any package between versions 2.1.0 and 2.1.3 ***inclusive*** is a suitable import candidate.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

For any particular version `vers`, version ranges are defined as follows:

```
(min,max)    means   min <  vers < max
(min, max]   means   min <  vers <= max
[min, max)   means   min <= vers < max
```

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets "[ ]" for an inclusive match and parentheses "( )" for an exclusive match.

The value of the version attribute now defines that any package between versions 2.1.0 and 2.1.3 ***inclusive*** is a suitable import candidate.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

For any particular version `vers`, version ranges are defined as follows:

```
(min,max)    means   min <  vers < max
(min, max]   means   min <  vers <= max
[min, max)   means   min <= vers < max
[min,max]    means   min <= vers <= max
```

MANIFEST.MF: Specifying Package Version Ranges

Since this assumption might not always be valid, OSGi allows version ranges to be specified using square brackets "[ ]" for an inclusive match and parentheses "( )" for an exclusive match.

The value of the version attribute now defines that any package between versions 2.1.0 and 2.1.3 ***inclusive*** is a suitable import candidate.

```
Import-Package: org.bar.foo.utils; version=[2.1.0,2.1.3],
 org.bar.foo.stuff; vendor="My company"
```

For any particular version `vers`, version ranges are defined as follows:

```
(min,max)    means   min <  vers < max
(min, max]   means   min <  vers <= max
[min, max)   means   min <= vers < max
[min,max]    means   min <= vers <= max
```

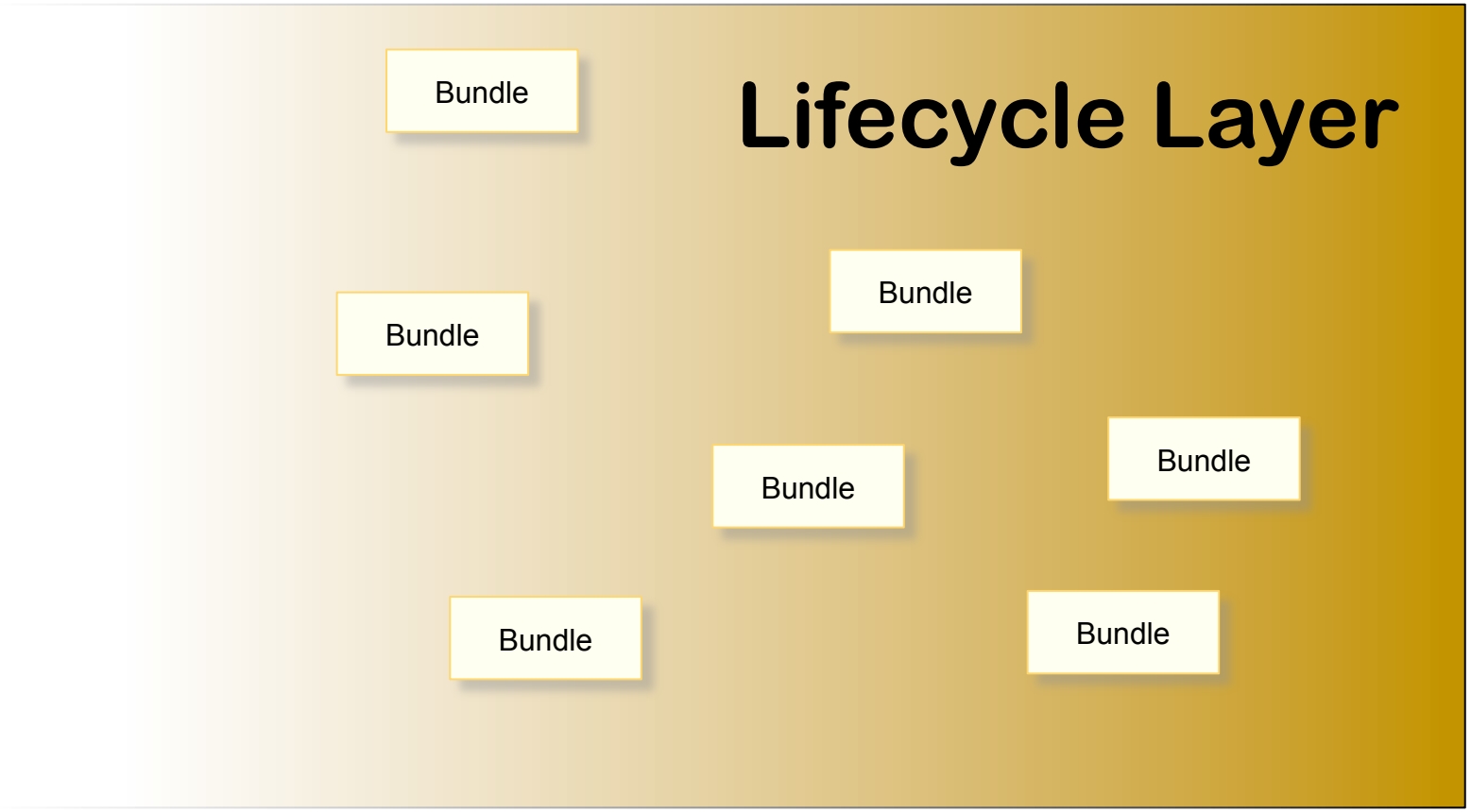
Notice that a square bracket ***and*** a parenthesis can be used in the same version range definition.

OSGi

The Lifecycle Layer

OSGi Lifecycle Layer: Overview

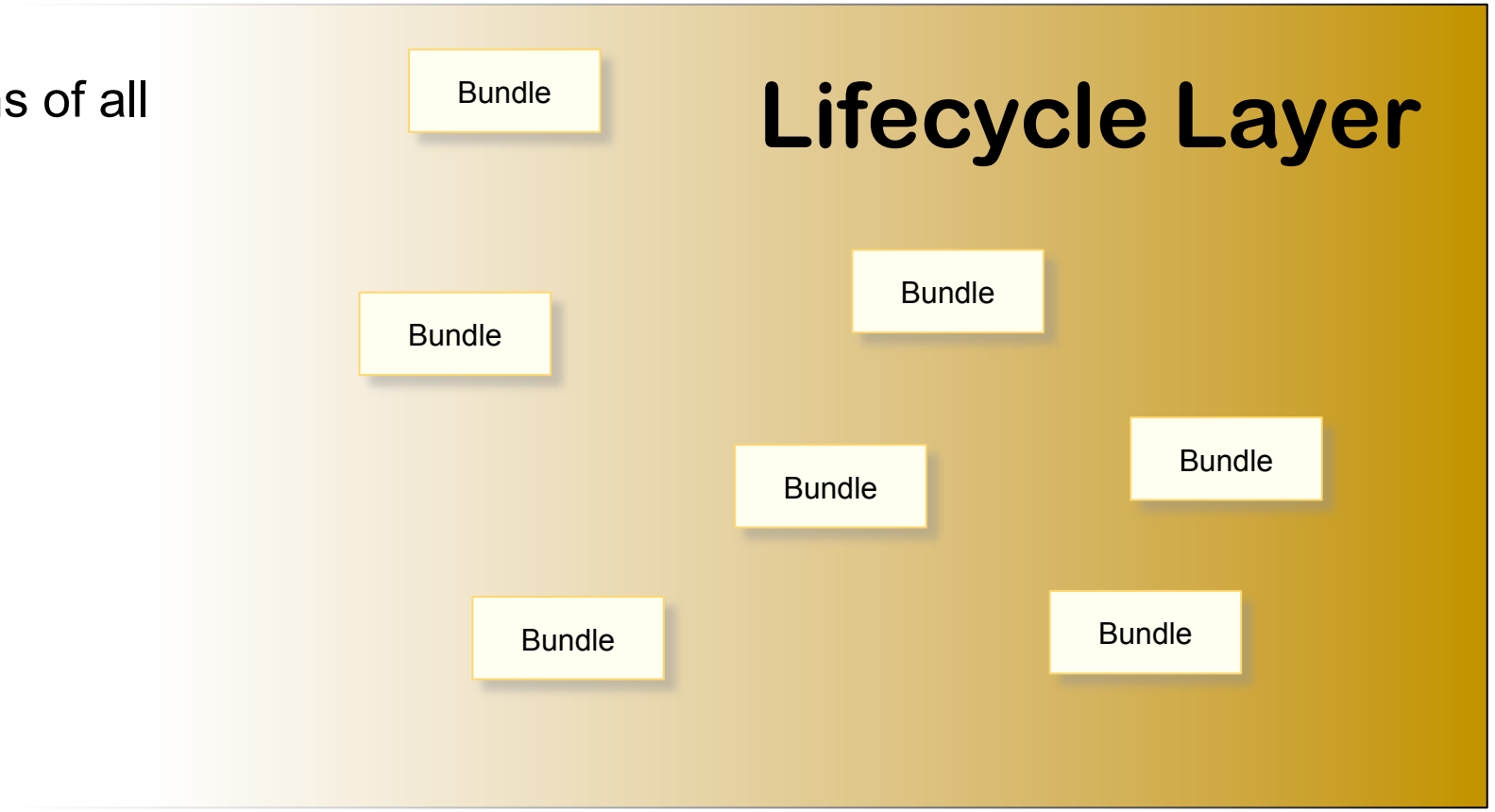
The Lifecycle Layer in the OSGi Framework manages the lifecycle of bundles. From the perspective of the functionality within a single bundle, this layer serves both an external and internal purpose:



OSGi Lifecycle Layer: Overview

The Lifecycle Layer in the OSGi Framework manages the lifecycle of bundles. From the perspective of the functionality within a single bundle, this layer serves both an external and internal purpose:

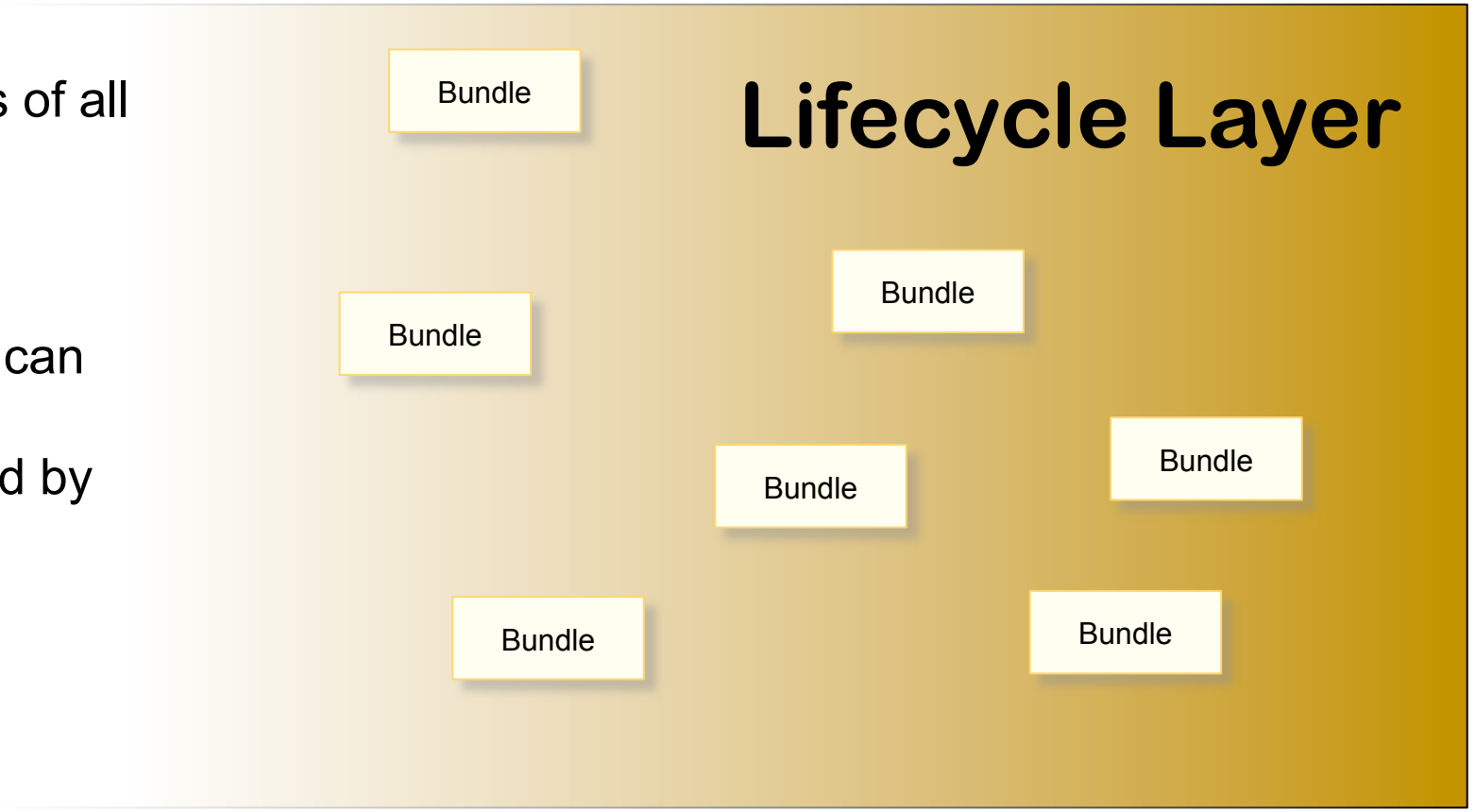
- **External**
  Manages the lifecycle operations of all bundles in the framework



OSGi Lifecycle Layer: Overview

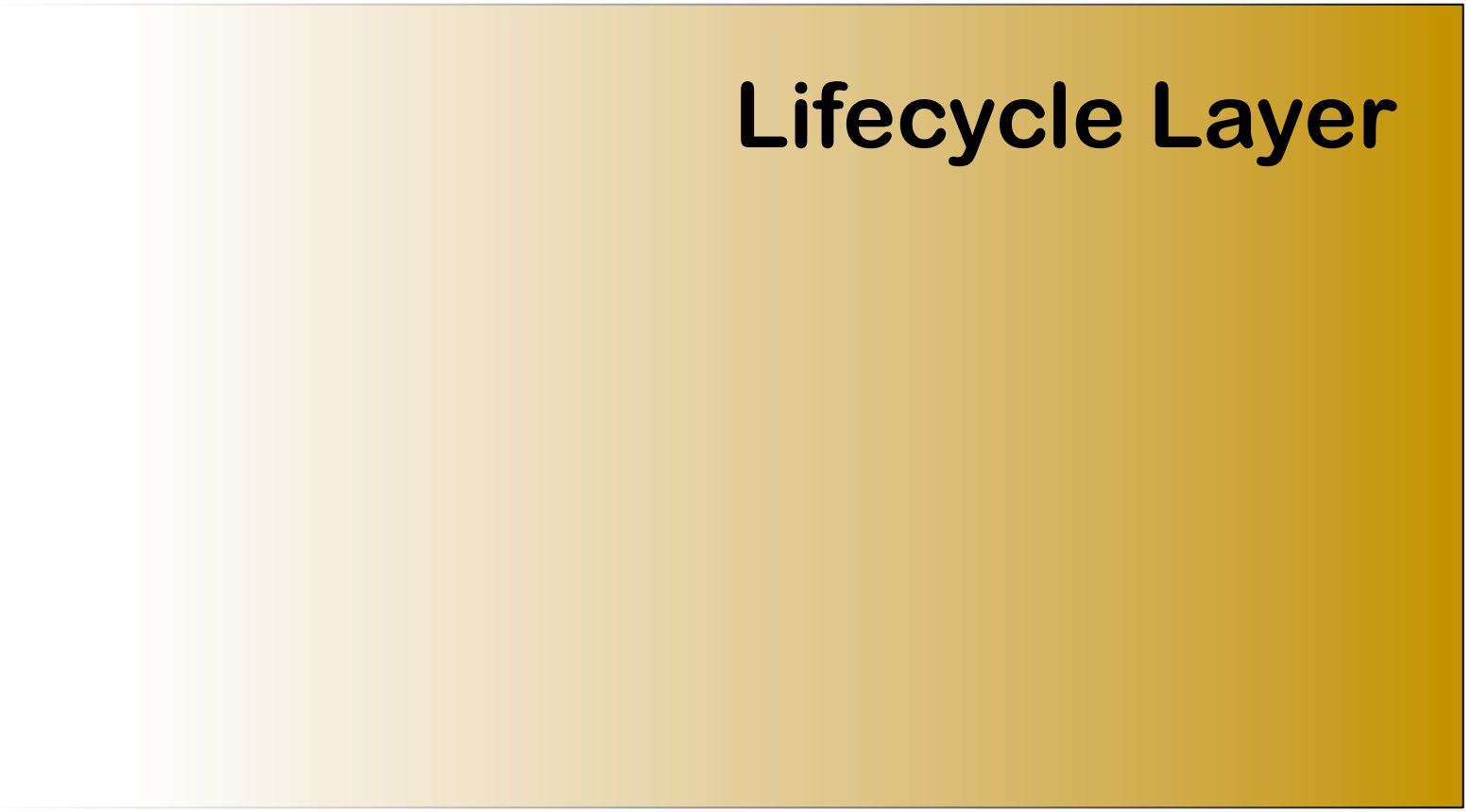
The Lifecycle Layer in the OSGi Framework manages the lifecycle of bundles. From the perspective of the functionality within a single bundle, this layer serves both an external and internal purpose:

- **External**
  Manages the lifecycle operations of all bundles in the framework
- **Internal**
  Provides the API through which application code within a bundle can gain access to both its execution context and the services provided by the OSGi Framework



OSGi Lifecycle Layer: Lifecycle Operations

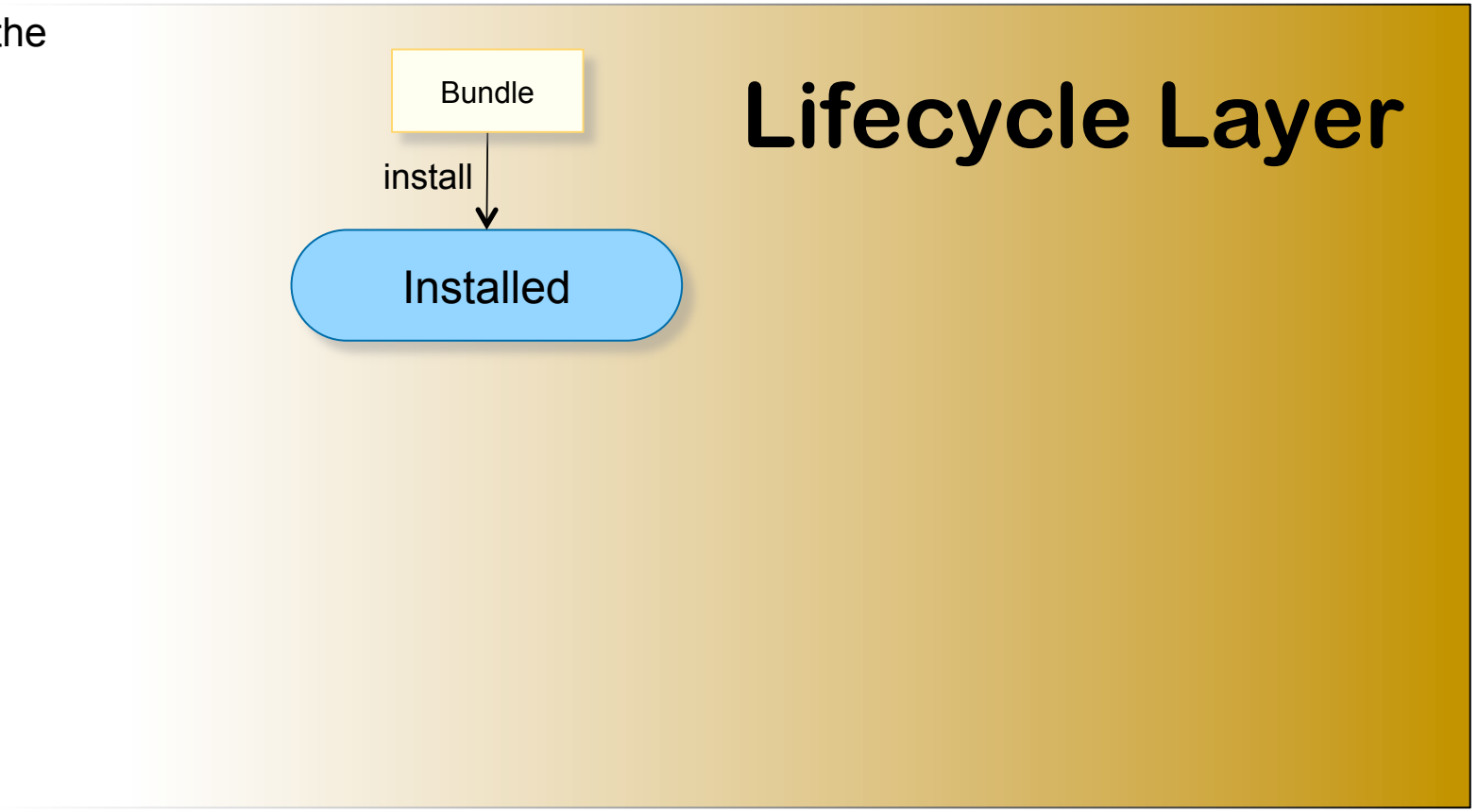
Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

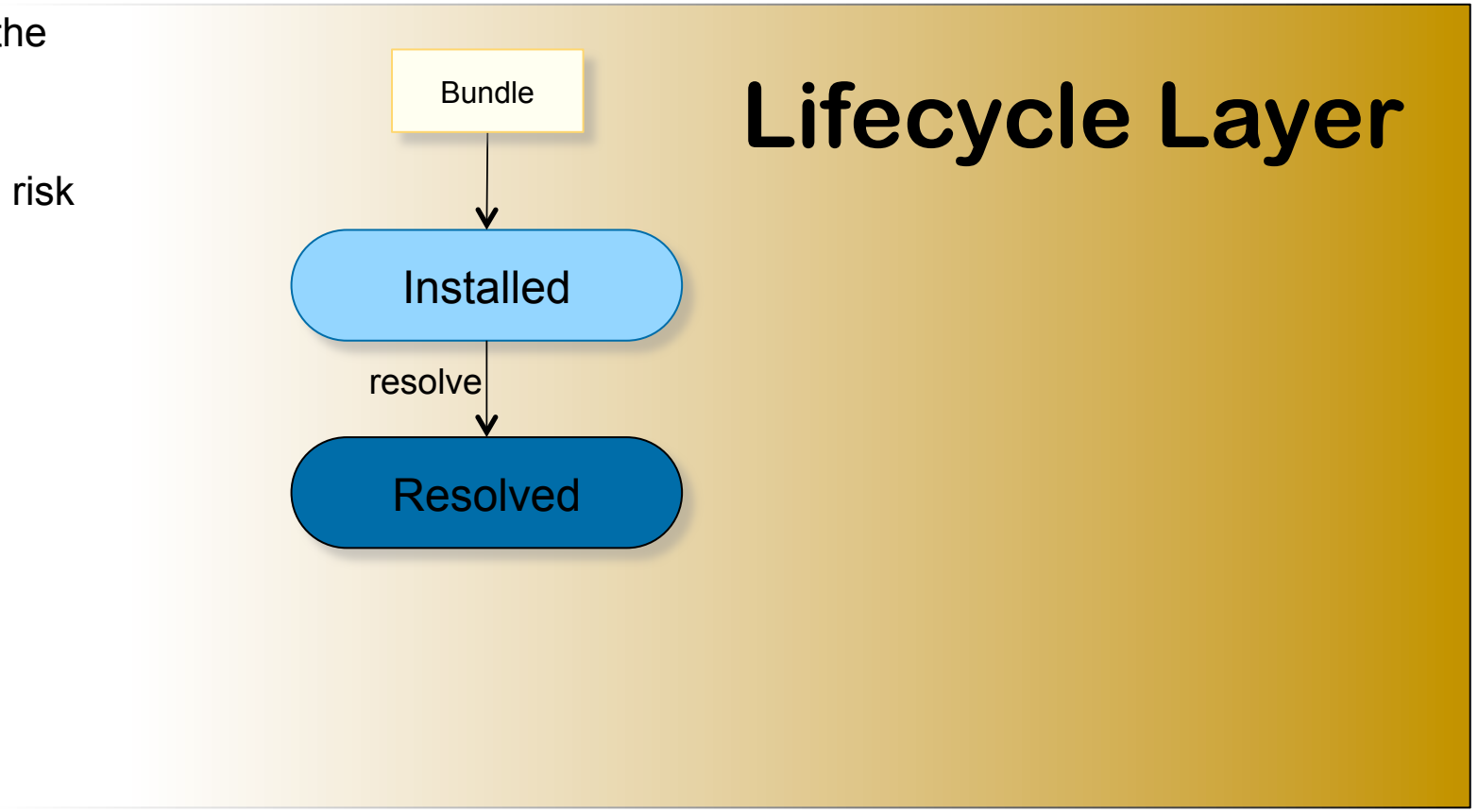
- **Installed** The bundle is made known to the OSGi Framework



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

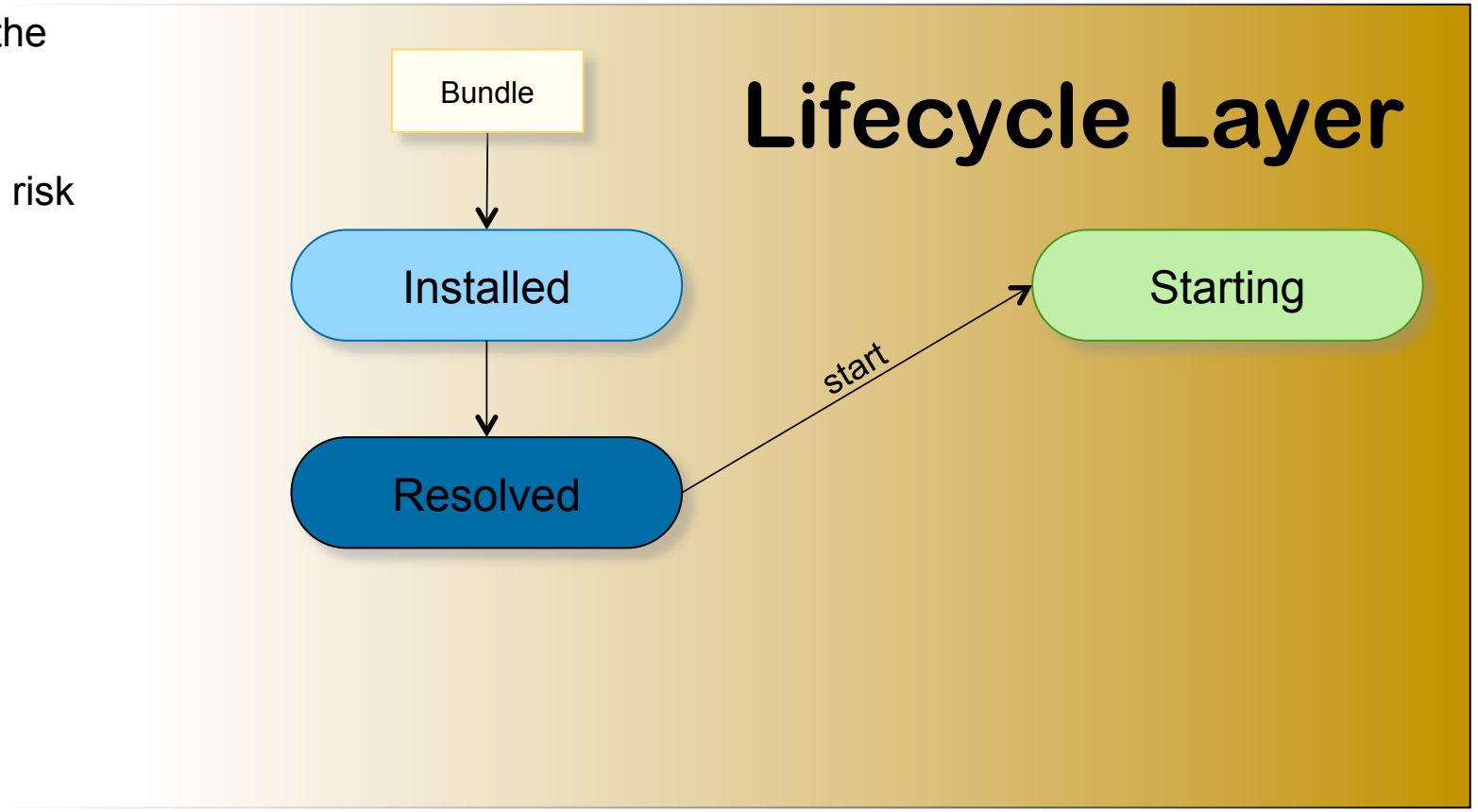
- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

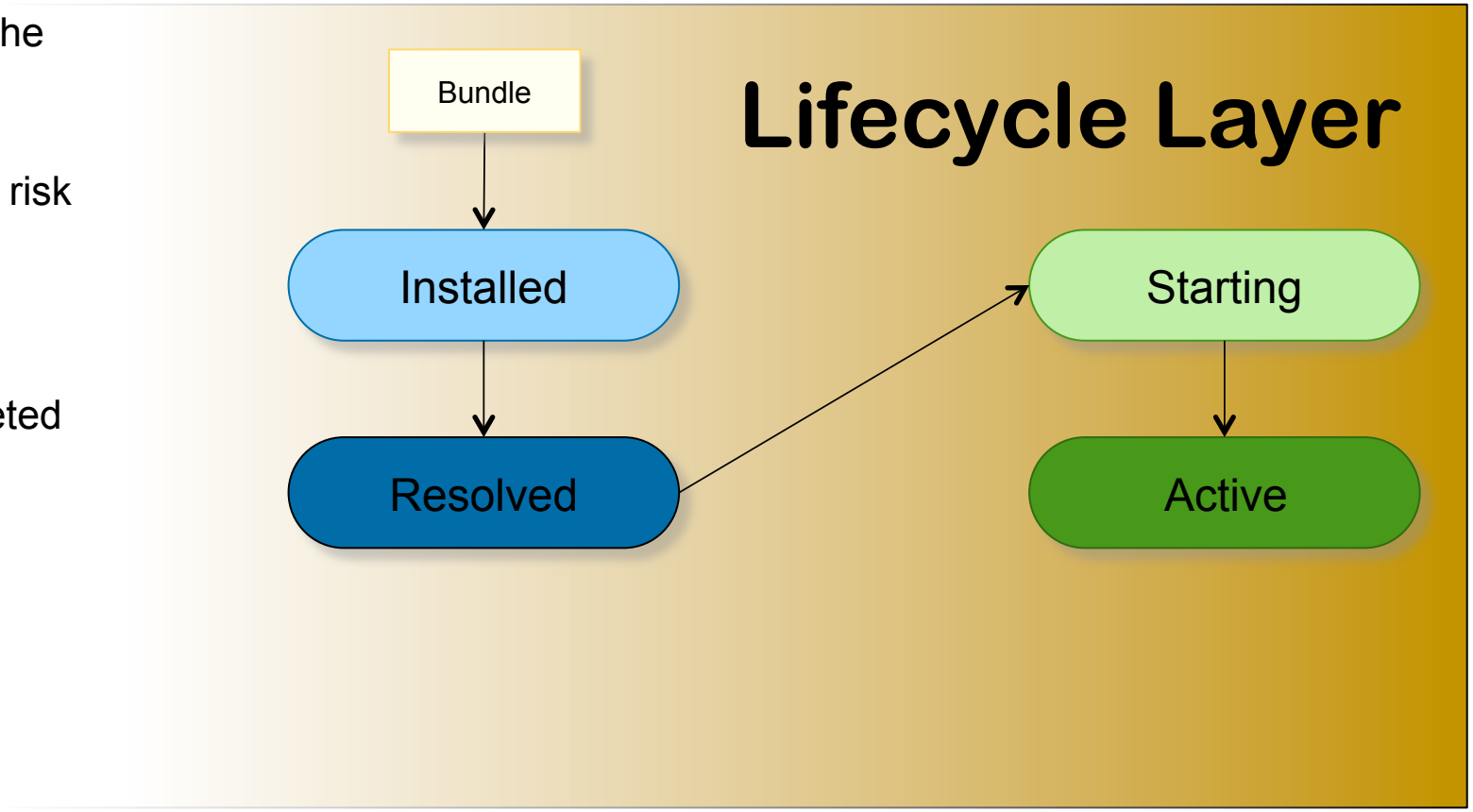
- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.
- **Starting** The bundle is in the process of starting up



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

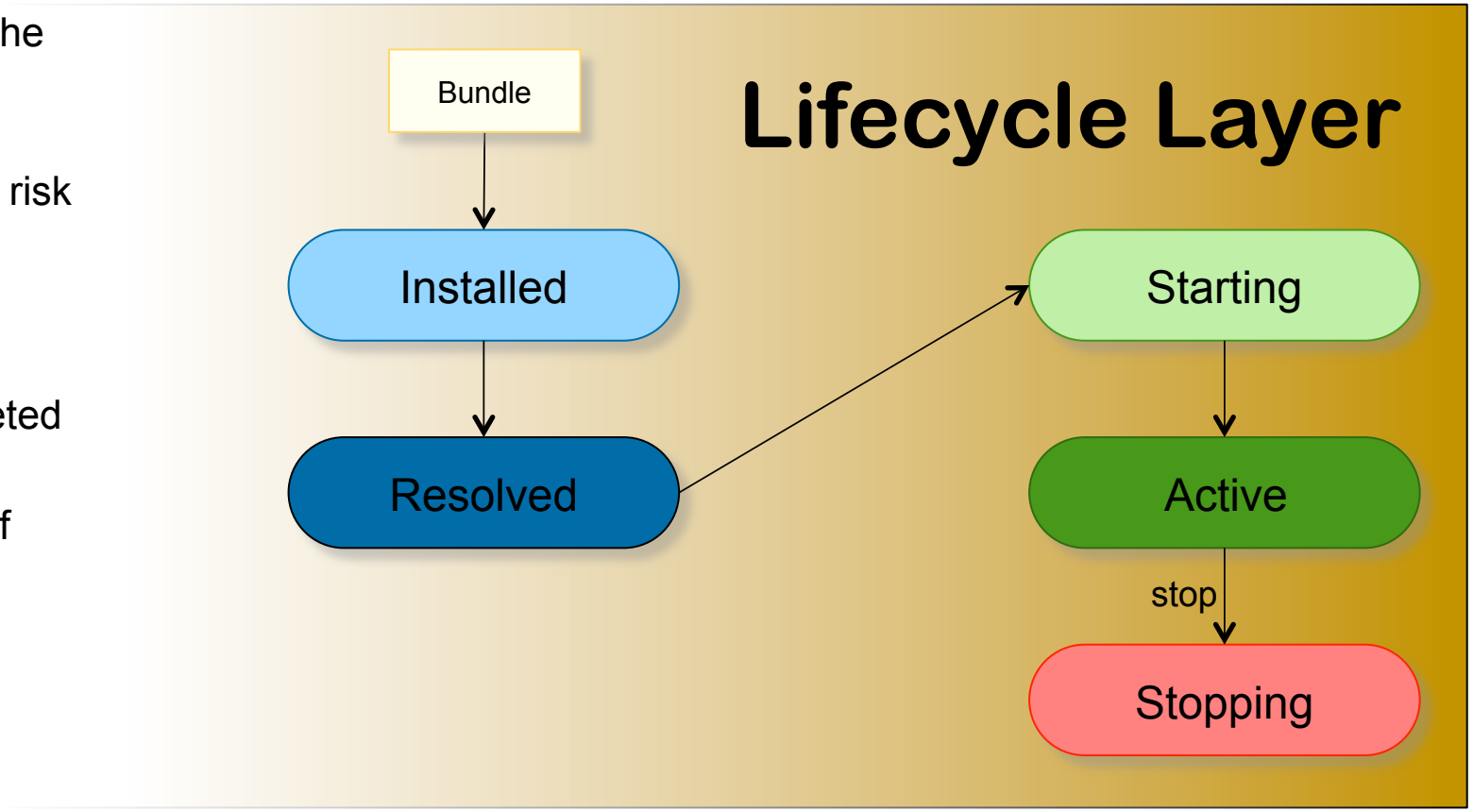
- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.
- **Starting** The bundle is in the process of starting up
- **Active** The activation coding has completed and the bundle is now ready for use



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

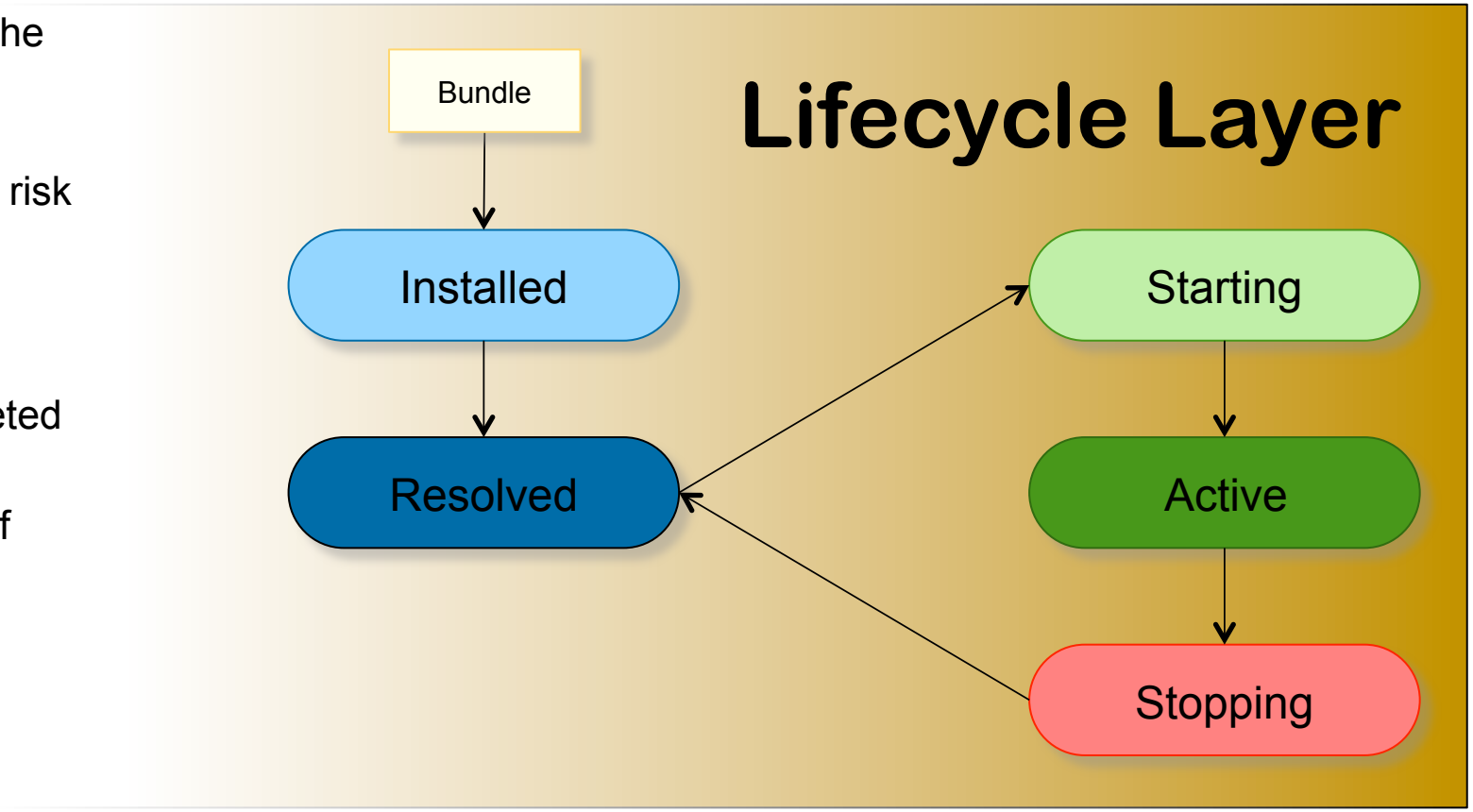
- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.
- **Starting** The bundle is in the process of starting up
- **Active** The activation coding has completed and the bundle is now ready for use
- **Stopping** The bundle is in the process of shutting down



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

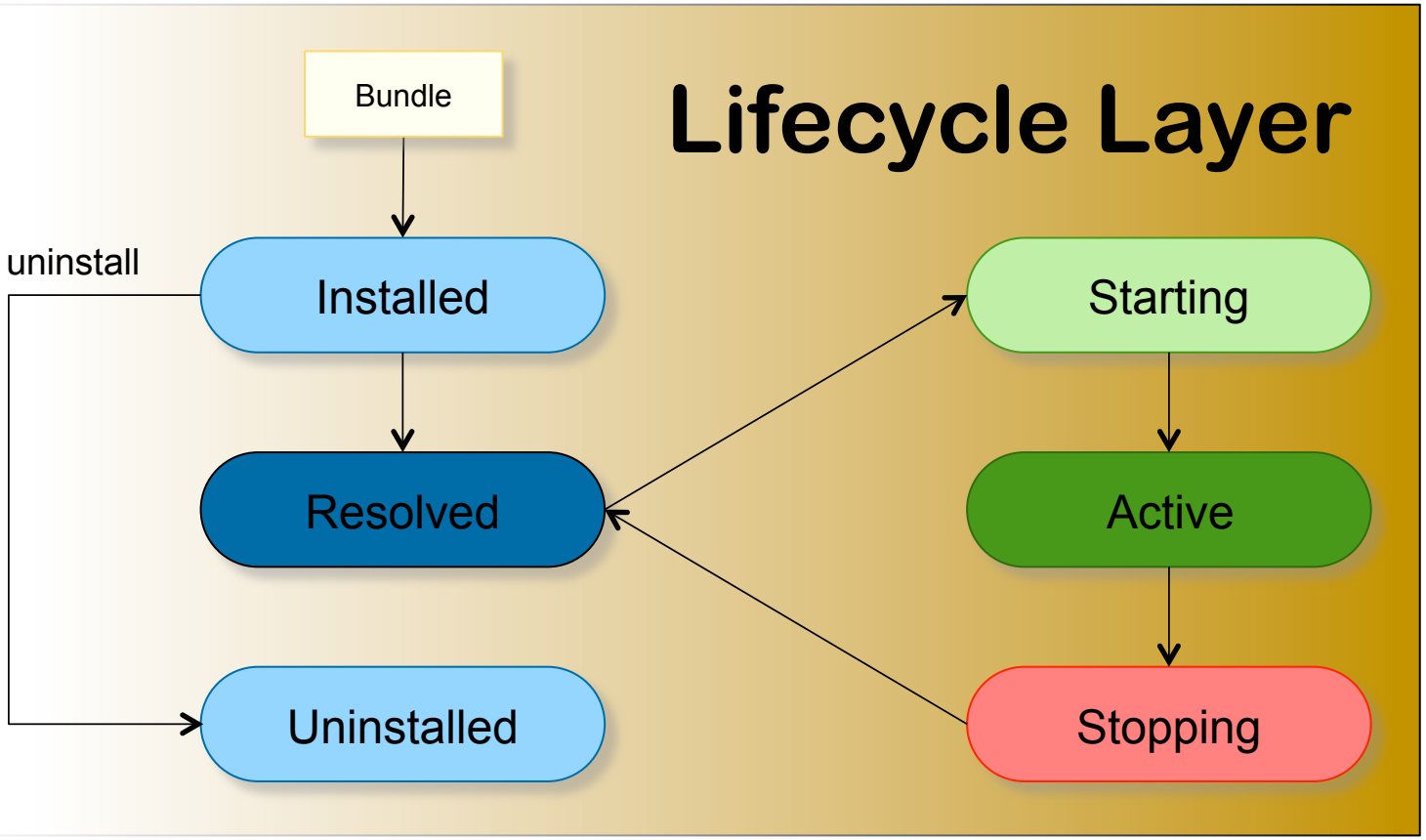
- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.
- **Starting** The bundle is in the process of starting up
- **Active** The activation coding has completed and the bundle is now ready for use
- **Stopping** The bundle is in the process of shutting down & returns to the resolved state



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

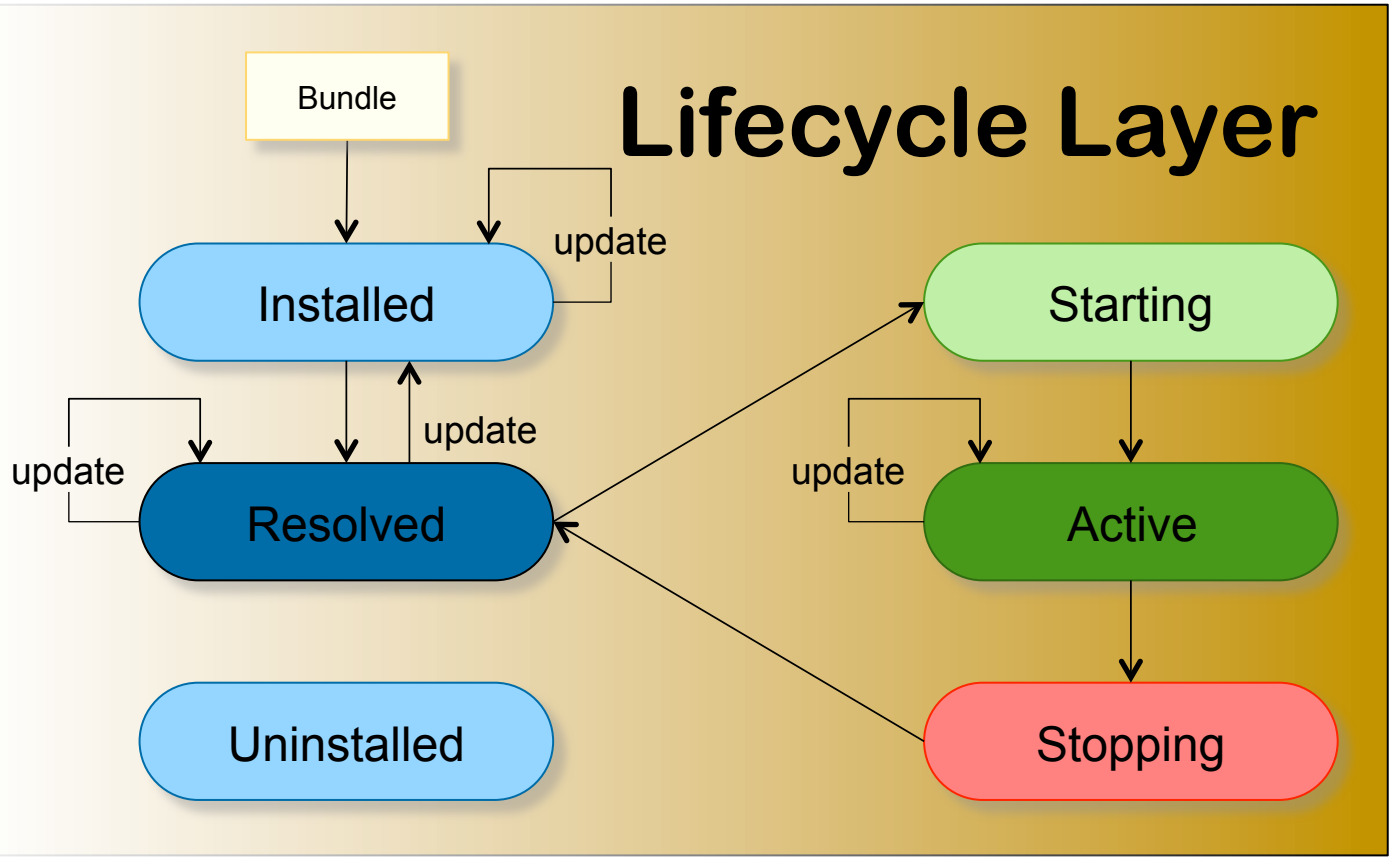
- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.
- **Starting** The bundle is in the process of starting up
- **Active** The activation coding has completed and the bundle is now ready for use
- **Stopping** The bundle is in the process of shutting down & returns to the resolved state
- **Uninstalled** The bundle is no longer useable by the OSGi Framework



OSGi Lifecycle Layer: Lifecycle Operations

Any bundle that needs either to interact with the OSGi Framework or perform some special startup or shutdown processing, will pass through a fixed sequence of steps during its lifecycle.

- **Installed** The bundle is made known to the OSGi Framework
- **Resolved** All bundle dependencies are calculated and satisfied. This avoids the risk of class cast exceptions at runtime.
- **Starting** The bundle is in the process of starting up
- **Active** The activation coding has completed and the bundle is now ready for use
- **Stopping** The bundle is in the process of shutting down & returns to the resolved state
- **Uninstalled** The bundle is no longer useable by the OSGi Framework
- **Update** A new bundle version is available



OSGi Lifecycle Layer: Basic Metadata

In order for the Lifecycle Layer to be able to start and stop your bundle, you must declare in the `MANIFEST.MF` file which class handles bundle activation.

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.bar.foo.myapp
Bundle-Version: 1.0.0
Bundle-Activator: org.bar.foo.myapp.Activator
Import-Package: org.osgi.framework; version="[1.3,2.0]"
```

The `Bundle-Activator` property informs the OSGi Framework which class implements the interface `org.osgi.framework.BundleActivator`.

OSGi Lifecycle Layer: Starting & Stopping Bundles

`org.osgi.framework.BundleActivator` provides 2 methods: `start()` and `stop()`

```
package org.bar.foo.myapp;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
  public void start(BundleContext ctx) {
    // Perform startup tasks here
  }
  public void stop(BundleContext ctx) {
    // Perform shutdown tasks here
  }
}
```

Typical tasks performed during `BundleActivator.start()` are:

- Event callback registration
- Starting threads

OSGi Lifecycle Layer: `BundleContext`

Both the `start()` and `stop()` methods receive a parameter of type `BundleContext`.

```
package org.bar.foo.myapp;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
  public void start(BundleContext ctx) {
    // Perform startup tasks here
  }
  public void stop(BundleContext ctx) {
    // Perform shutdown tasks here
  }
}
```

OSGi Lifecycle Layer: BundleContext

Both the `start()` and `stop()` methods receive a parameter of type `BundleContext`.

```
package org.bar.foo.myapp;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
  public void start(BundleContext ctx) {
    // Perform startup tasks here
  }
  public void stop(BundleContext ctx) {
    // Perform shutdown tasks here
  }
}
```

The `BundleContext` parameter gives you access to the OSGi execution context

OSGi Lifecycle Layer: BundleContext

Both the `start()` and `stop()` methods receive a parameter of type `BundleContext`.

```
package org.bar.foo.myapp;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
  public void start(BundleContext ctx) {
    // Perform startup tasks here
  }
  public void stop(BundleContext ctx) {
    // Perform shutdown tasks here
  }
}
```

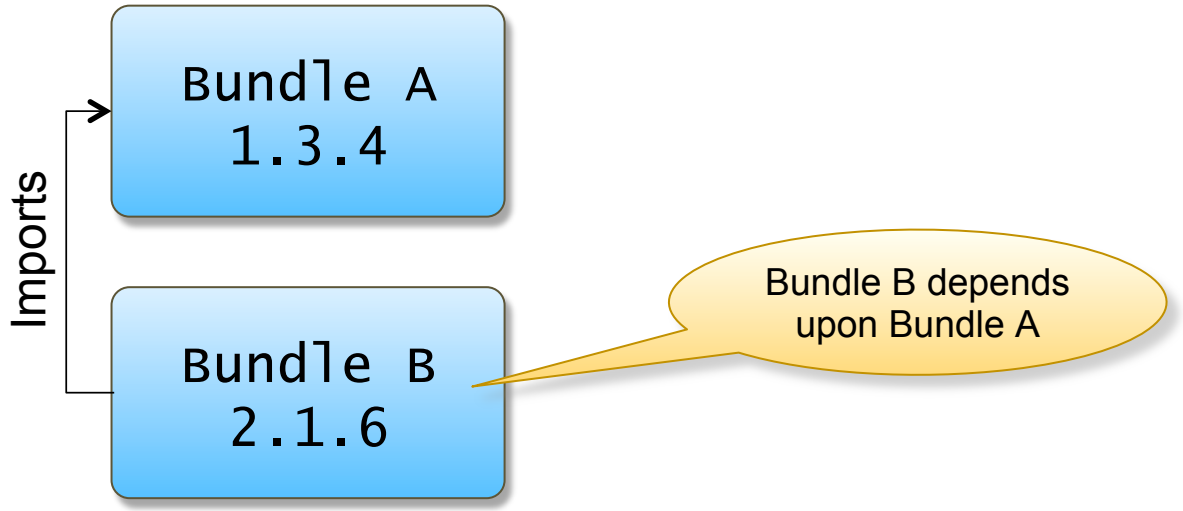
The methods belonging to the `BundleContext` parameter can be divided into two broad categories:

- Deployment and Lifecycle Management
- Bundle interaction using Services

OSGi Lifecycle Layer: Update & Refresh 1/3

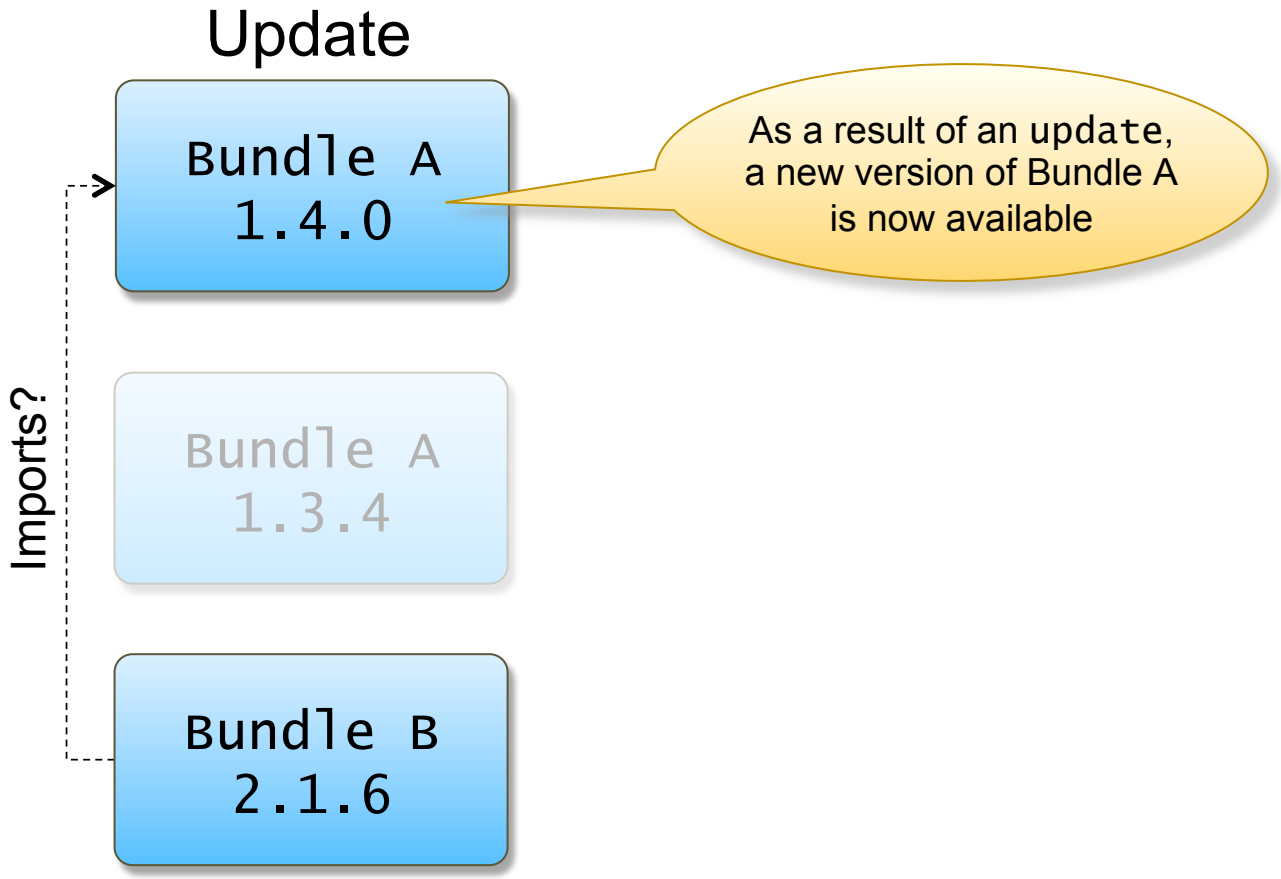
Bundles can be updated to new versions whilst the system is running; however, this action could potentially be disruptive to active bundle dependencies.

Therefore, OSGi performs updates as a two-step process: an ***update*** followed by a ***refresh***.



OSGi Lifecycle Layer: Update & Refresh 2/3

Using the ***update*** command, a new version of bundle A is made available, so the OSGi Framework recalculates a new dependency graph to ensure that all dependencies can still be satisfied.



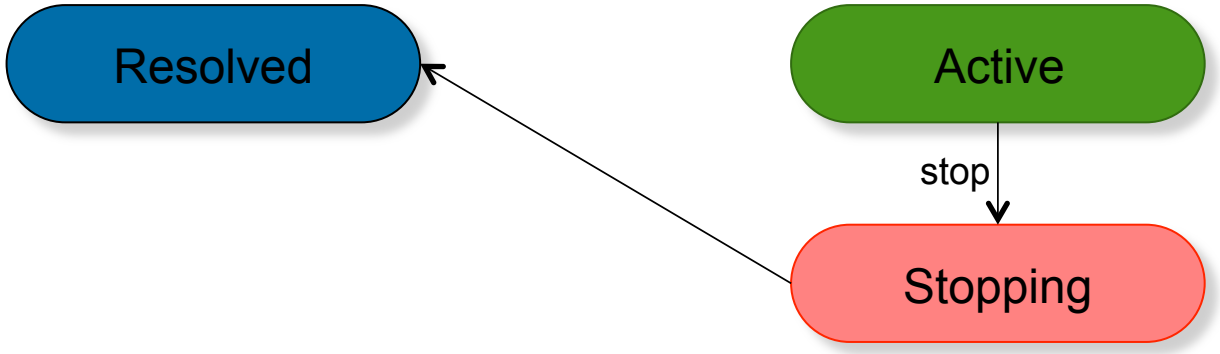
OSGi Lifecycle Layer: Update & Refresh 3/3

Once the new dependency graph is fully calculated and all the dependencies have been satisfied, the administrator then issues a ***refresh***. This process works according to the following steps:

OSGi Lifecycle Layer: Update & Refresh 3/3

Once the new dependency graph is fully calculated and all the dependencies have been satisfied, the administrator then issues a ***refresh***. This process works according to the following steps:

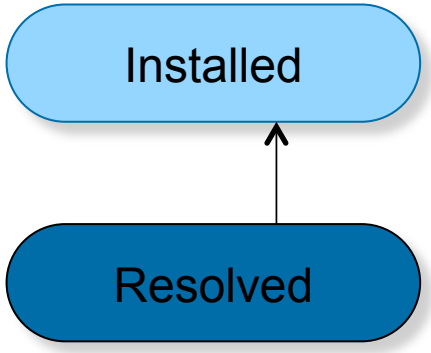
1) All **Active** bundles in the dependency graph are stopped. This returns them to the **Resolved** state



OSGi Lifecycle Layer: Update & Refresh 3/3

Once the new dependency graph is fully calculated and all the dependencies have been satisfied, the administrator then issues a ***refresh***. This process works according to the following steps:

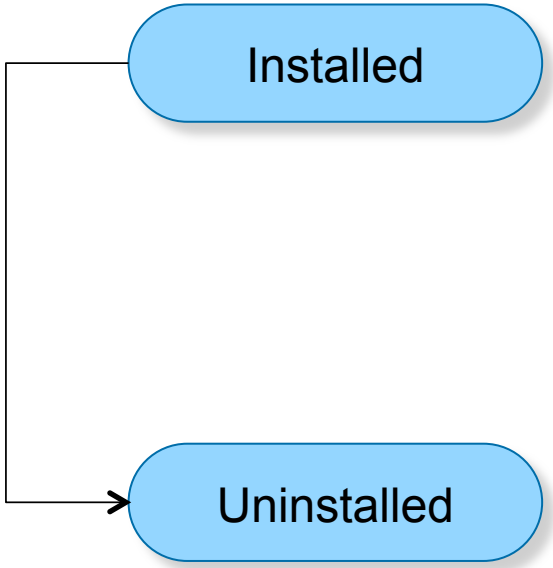
1) All **Active** bundles in the dependency graph are stopped. This returns them to the **Resolved** state
2) All bundles in the **Resolved** state have their dependencies broken, returning them to the **Installed** state



OSGi Lifecycle Layer: Update & Refresh 3/3

Once the new dependency graph is fully calculated and all the dependencies have been satisfied, the administrator then issues a ***refresh***. This process works according to the following steps:

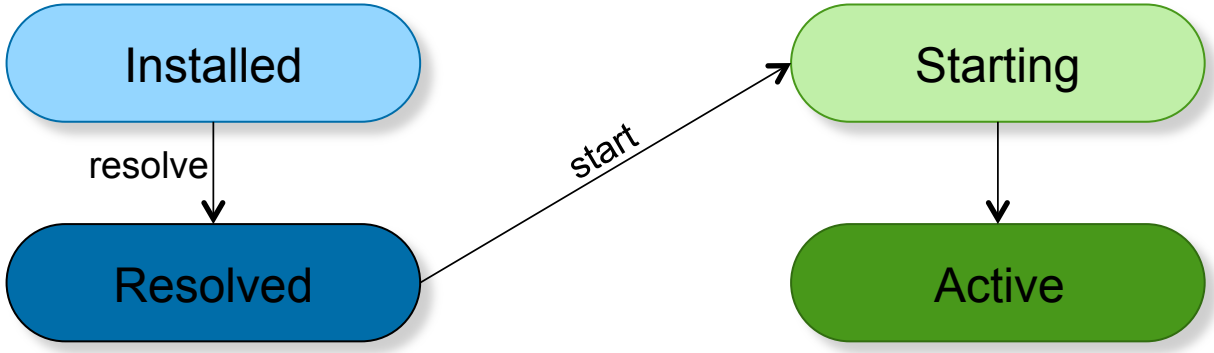
1) All **Active** bundles in the dependency graph are stopped. This returns them to the **Resolved** state
2) All bundles in the **Resolved** state have their dependencies broken, returning them to the **Installed** state
3) All bundles in the **Uninstalled** state are removed from the dependency graph



OSGi Lifecycle Layer: Update & Refresh 3/3

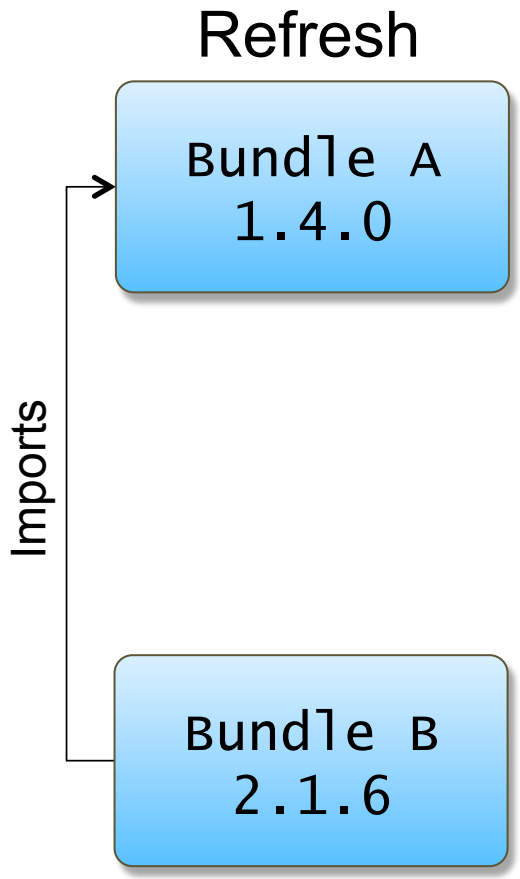
Once the new dependency graph is fully calculated and all the dependencies have been satisfied, the administrator then issues a ***`refresh`***. This process works according to the following steps:

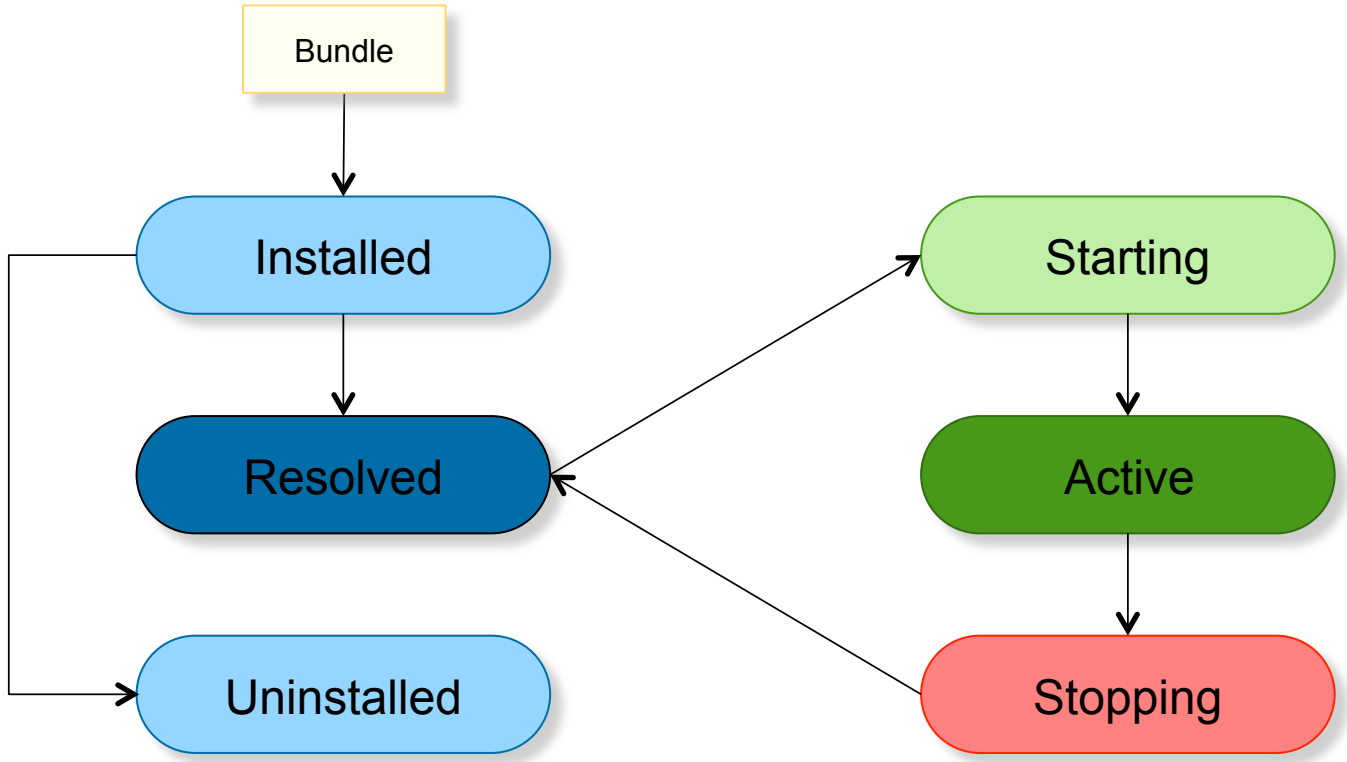
1) All **Active** bundles in the dependency graph are stopped. This returns them to the **Resolved** state
2) All bundles in the **Resolved** state have their dependencies broken, returning them to the **Installed** state
3) All bundles in the **Uninstalled** state are removed from the dependency graph
4) All previously **Active** bundles are restarted, which automatically causes them to move through the **Resolved** state back to the **Active** state



OSGi Lifecycle Layer: Update & Refresh 4/4

After the ***refresh***, Bundle B now imports the updated version of Bundle A





OSGi

The Service Layer

OSGi Service Layer: When To Use?

The OSGi Service Layer is useful in situations where you require:

- Loose coupling between service providers and consumers
- Interface based development
- Metadata description of dependencies
- Multiple implementations based on the same interface

OSGi Service Layer: When To Use?

The OSGi Service Layer is useful in situations where you require:

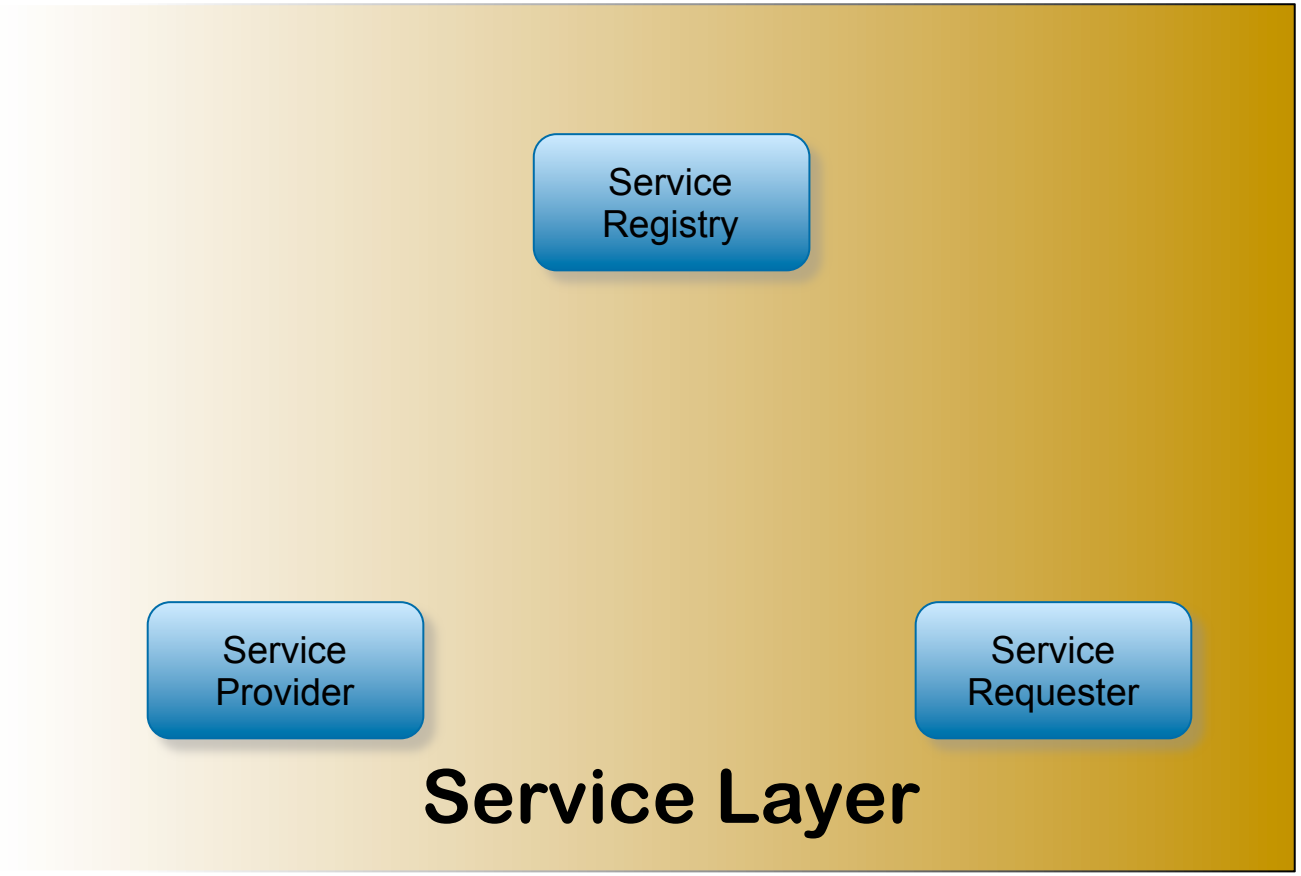
- Loose coupling between service providers and consumers
- Interface based development
- Metadata description of dependencies
- Multiple implementations based on the same interface

Alternatively, the Service Layer will probably not be the correct solution in situations where either:

- A tight coupling exists between two units of code that are developed and updated together
- There will only ever be one implementation of an interface.

OSGi Service Layer: Overview

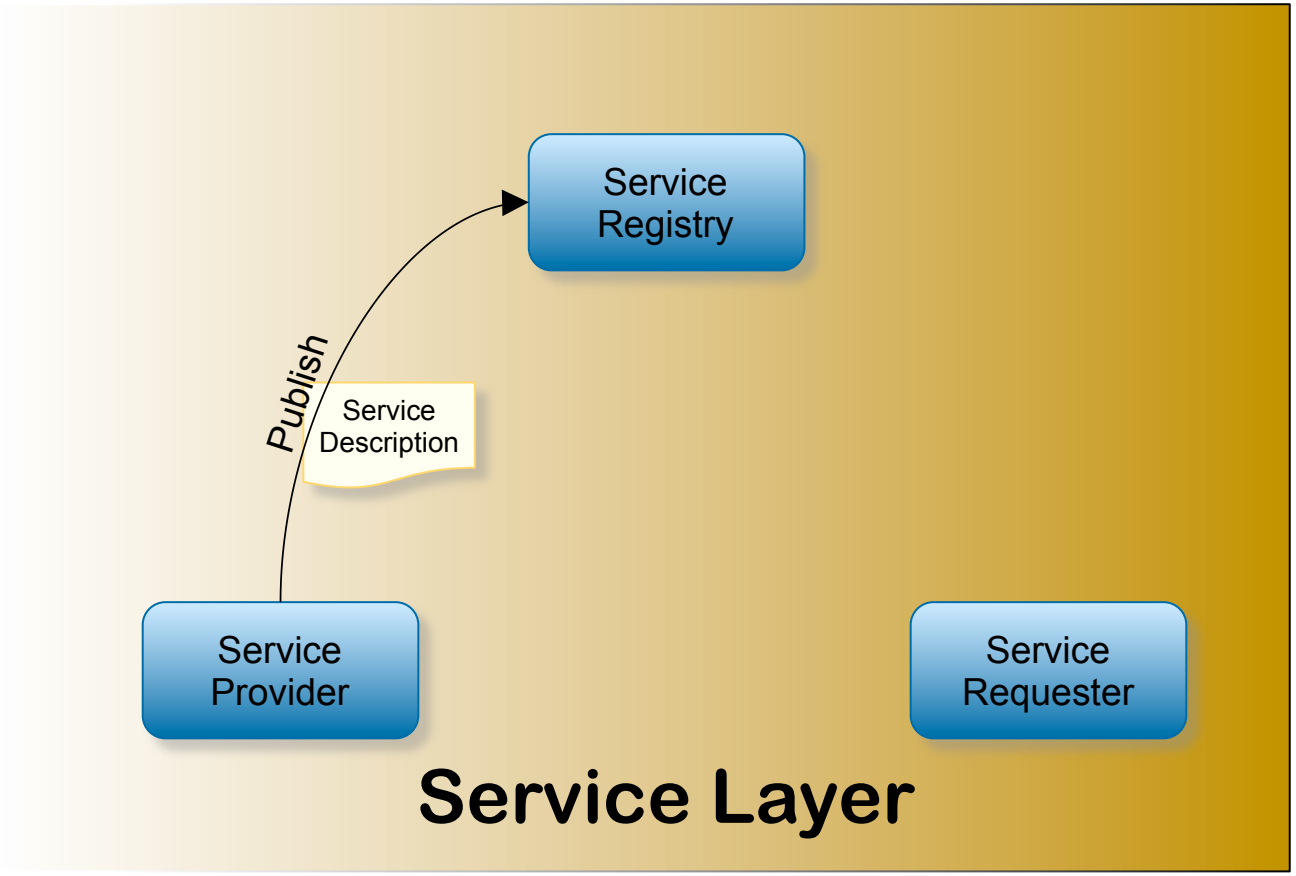
The Service Layer makes bundles available as services through the ***Publish-Find-Bind*** pattern.



OSGi Service Layer: Overview

The Service Layer makes bundles available as services through the ***Publish-Find-Bind*** pattern.

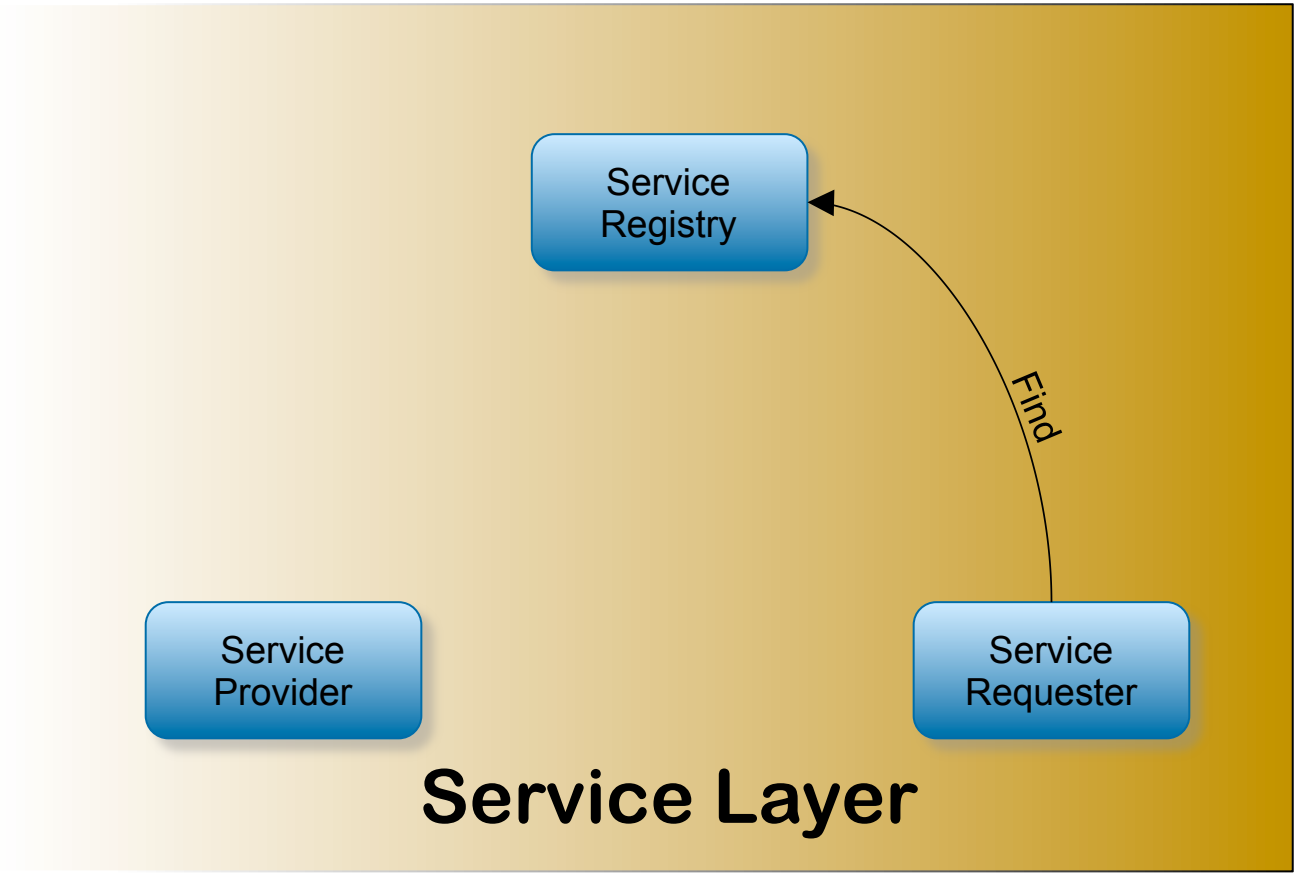
1. **Publish**
   The provider publishes a new service description into the Service Registry



OSGi Service Layer: Overview

The Service Layer makes bundles available as services through the ***Publish-Find-Bind*** pattern.

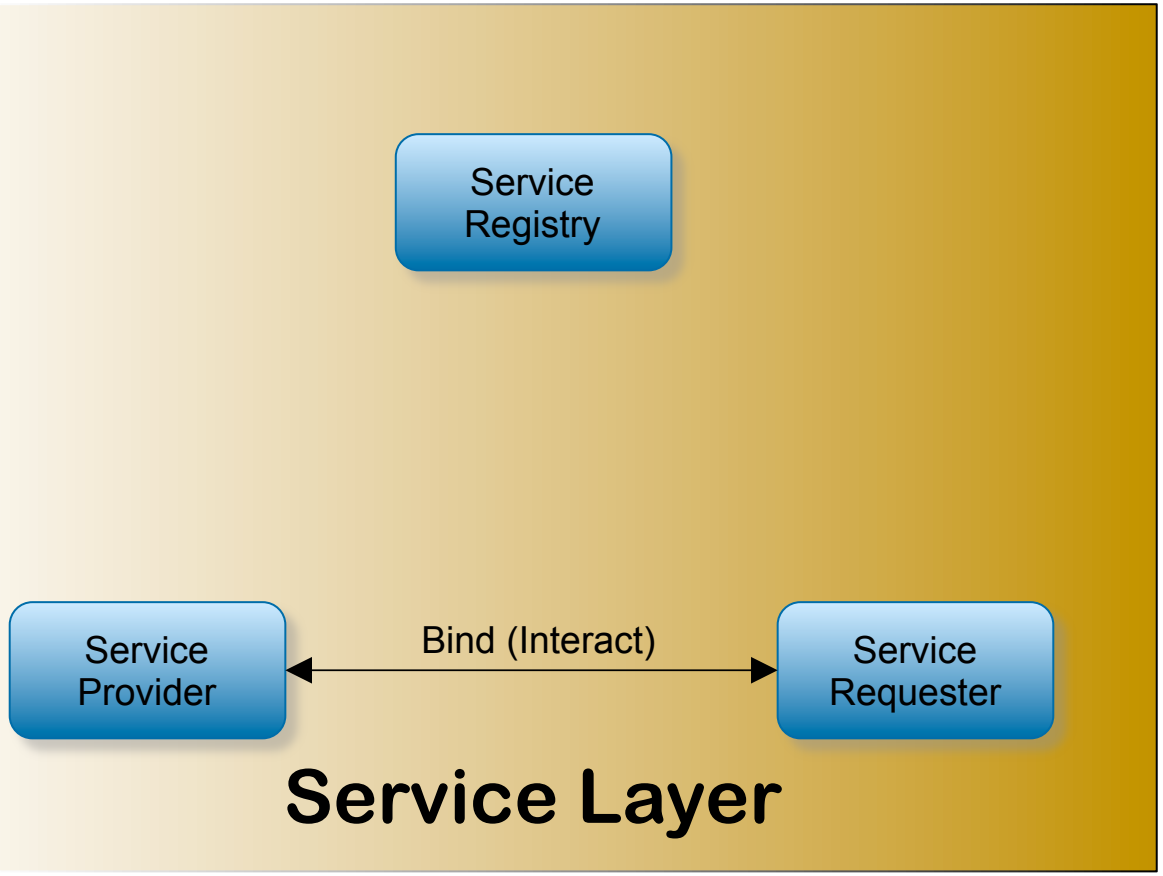
1. **Publish**
   The provider publishes a new service description into the Service Registry
2. **Find**
   The Service Requester then searches the repository for the required service



OSGi Service Layer: Overview

The Service Layer makes bundles available as services through the ***Publish-Find-Bind*** pattern.

1. **Publish**
   The provider publishes a new service description into the Service Registry
2. **Find**
   The Service Requester then searches the repository for the required service
3. **Bind**
   If the required service exists in the repository, then the service requester interacts with it directly



OSGi Service Layer: The `BundleContext` Interface

The `BundleContext` object is passed to the `start()` and `stop()` methods of the class implementing the `BundleActivator` interface. The interface methods listed here are related to services.

```
public interface BundleContext {









}
```

OSGi Service Layer: The BundleContext Interface

The BundleContext object is passed to the start() and stop() methods of the class implementing the BundleActivator interface. The interface methods listed here are related to services.

```
public interface BundleContext {
  void addServiceListener(ServiceListener listener, String filter) throws InvalidSyntaxException;
  void addServiceListener(ServiceListener listener);
  void removeServiceListener(ServiceListener listener);









}
```

These service related methods handle:

- Adding and removing service listeners

OSGi Service Layer: The BundleContext Interface

The BundleContext object is passed to the start() and stop() methods of the class implementing the BundleActivator interface. The interface methods listed here are related to services.

```
public interface BundleContext {
  void addServiceListener(ServiceListener listener, String filter) throws InvalidSyntaxException;
  void addServiceListener(ServiceListener listener);
  void removeServiceListener(ServiceListener listener);

  ServiceRegistration registerService(String[] clazzes, Object service, Dictionary properties);
  ServiceRegistration registerService(String clazz, Object service, Dictionary properties);









}
```

These service related methods handle:

- Adding and removing service listeners
- Registering services

OSGi Service Layer: The BundleContext Interface

The BundleContext object is passed to the start() and stop() methods of the class implementing the BundleActivator interface. The interface methods listed here are related to services.

```
public interface BundleContext {
  void addServiceListener(ServiceListener listener, String filter) throws InvalidSyntaxException;
  void addServiceListener(ServiceListener listener);
  void removeServiceListener(ServiceListener listener);

  ServiceRegistration registerService(String[] clazzes, Object service, Dictionary properties);
  ServiceRegistration registerService(String clazz, Object service, Dictionary properties);

  ServiceReference[] getServiceReferences(String clazz, String filter) throws InvalidSyntaxException;
  ServiceReference[] getAllServiceReferences(String clazz, String filter) throws InvalidSyntaxException;
  ServiceReference   getServiceReference(String clazz);



}
```

These service related methods handle:

- Adding and removing service listeners
- Registering services
- Finding service references

OSGi Service Layer: The BundleContext Interface

The BundleContext object is passed to the start() and stop() methods of the class implementing the BundleActivator interface. The interface methods listed here are related to services.

```
public interface BundleContext {
  void addServiceListener(ServiceListener listener, String filter) throws InvalidSyntaxException;
  void addServiceListener(ServiceListener listener);
  void removeServiceListener(ServiceListener listener);

  ServiceRegistration registerService(String[] clazzes, Object service, Dictionary properties);
  ServiceRegistration registerService(String clazz, Object service, Dictionary properties);

  ServiceReference[] getServiceReferences(String clazz, String filter) throws InvalidSyntaxException;
  ServiceReference[] getAllServiceReferences(String clazz, String filter) throws InvalidSyntaxException;
  ServiceReference   getServiceReference(String clazz);

  Object getService(ServiceReference reference);
  boolean ungetService(ServiceReference reference);
}
```

These service related methods handle:

- Adding and removing service listeners
- Registering services
- Finding service references
- Binding to a service. De-registering usage

OSGi Service Layer: Publishing a Service 1/2

Publishing an OSGi service first requires a service description to be built. After that, it can be published.

```
public void start(BundleContext ctx) {









}
```

OSGi Service Layer: Publishing a Service 1/2

Publishing an OSGi service first requires a service description to be built. After that, it can be published.

```
public void start(BundleContext ctx) {
  String[] interfaces = new String[] { StockListing.class.getName(), StockChart.class.getName() };









}
```

The steps involved are

- Provide an (array of) interface name(s)

OSGi Service Layer: Publishing a Service 1/2

Publishing an OSGi service first requires a service description to be built. After that, it can be published.

```
public void start(BundleContext ctx) {
  String[] interfaces = new String[] { StockListing.class.getName(), StockChart.class.getName() };

  Dictionary metadata = new Properties();
  metadata.setProperty("name", "LSE");
  metadata.setProperty("currency", Currency.getInstance("GBP"));
  metadata.setProperty("country", "GB");


}
```

The steps involved are

- Provide an (array of) interface name(s)
- Create an optional dictionary of metadata properties

OSGi Service Layer: Publishing a Service 1/2

Publishing an OSGi service first requires a service description to be built. After that, it can be published.

```
public void start(BundleContext ctx) {
  String[] interfaces = new String[] { StockListing.class.getName(), StockChart.class.getName() };

  Dictionary metadata = new Properties();
  metadata.setProperty("name", "LSE");
  metadata.setProperty("currency", Currency.getInstance("GBP"));
  metadata.setProperty("country", "GB");

  ServiceRegistration registration = ctx.registerService(interfaces, new LSE(), metadata);
}
```

The steps involved are

- Provide an (array of) interface name(s)
- Create an optional dictionary of metadata properties
- Register the service

OSGi Service Layer: Publishing a Service 1/2

Publishing an OSGi service first requires a service description to be built. After that, it can be published.

```
public void start(BundleContext ctx) {
  String[] interfaces = new String[] { StockListing.class.getName(), StockChart.class.getName() };

  Dictionary metadata = new Properties();
  metadata.setProperty("name", "LSE");
  metadata.setProperty("currency", Currency.getInstance("GBP"));
  metadata.setProperty("country", "GB");

  ServiceRegistration registration = ctx.registerService(interfaces, new LSE(), metadata);
}
```

The steps involved are

- Provide an (array of) interface name(s)
- Create an optional dictionary of metadata properties
- Register the service

In the context above, we are registering a stock listing service that could supply data from any stock exchange. In this case, we are opting for the London Stock Exchange.

OSGi Service Layer: Publishing a Service 2/2

In addition to the arbitrary metadata service properties, there is a set of standard OSGi service properties.

Key	Type	Defined by	Description
`objectClass`	`String[]`	Developer	The class names by which the service was registered – cannot be changed after registration
`service.id`	`Long`	OSGi Framework	Unique registration sequence number assigned when the service is registered
`service.pid`	`String`	Developer	Persistent (unique) service identifier
`service.ranking`	`Integer`	Developer	Used to specify service priority during the discovery process. Services are sorted first in descending order by their ranking and then in ascending order by their id.
`service.description`	`String`	Developer	An arbitrary description of the service
`service.vendor`	`String`	Developer	An arbitrary vendor name providing the service

OSGi Service Layer: Finding a Service 1/3

The simplest way to find an OSGi service is to make a query based on just the interface name

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName());
```

If this query is successful, an indirect reference to the service will be returned.

OSGi Service Layer: Finding a Service 1/3

The simplest way to find an OSGi service is to make a query based on just the interface name

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName());
```

If this query is successful, an indirect reference to the service will be returned.

The reason the service reference is indirect is because the bundles used to implement services should always be considered dynamic objects.

OSGi Service Layer: Finding a Service 1/3

The simplest way to find an OSGi service is to make a query based on just the interface name

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName());
```

If this query is successful, an indirect reference to the service will be returned.

The reason the service reference is indirect is because the bundles used to implement services should always be considered dynamic objects.

Therefore, the actual bundle implementing the service could vary over the course of time; yet because of the contract provided by the service's interface, the OSGi Framework will always attempt to locate a suitable service.

OSGi Service Layer: Finding a Service 1/3

The simplest way to find an OSGi service is to make a query based on just the interface name

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName());
```

If this query is successful, an indirect reference to the service will be returned.

The reason the service reference is indirect is because the bundles used to implement services should always be considered dynamic objects.

Therefore, the actual bundle implementing the service could vary over the course of time; yet because of the contract provided by the service's interface, the OSGi Framework will always attempt to locate a suitable service.

The basic idea here is that you don't care about the actual implementation of a service, as long as it satisfies the contract.

OSGi Service Layer: Finding a Service 2/3

However, an OSGi service can be described to any arbitrary level of detail by means of the metadata in its service description. This metadata can then be queried when finding a service.

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName(), "(name=LSE)");
```

The extra parameter is a standard LDAP filter string as defined by RFC 1960 (`http://www.ietf.org/rfc/rfc1960.txt`).

A Quick Guide To LDAP Filter Strings

Attribute matching:	`(name=John Smith)`
Fuzzy matching:	`(name~=johnsmith)`
Wild-card matching:	`(name=Jo*n*Smith*)`
Does attribute exist:	`(name=*)`
Match ALL the contained clauses:	`(&(name=John Smith)(occupation=doctor))`
Match at least ONE of the contained clauses:	`(|(name~=John Smith)(name~=Smith John))`
Negate the contained clause:	`(!(name=John Smith))`

OSGi Service Layer: Finding a Service 3/3

The standard OSGi properties can also be included in the LDAP filter string.

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName(), "(&(name=LSE)(objectClass=org.example.StockChart))");
```

OSGi Service Layer: Binding to a Service

Once you have the indirect reference to the service in the registry, you must then bind to it. The process of binding via an indirect reference ensures you are never tied to a specific implementation.

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName(), "(&(name=LSE)(objectClass=org.example.StockChart))");

StockListing listing = (StockListing)cxt.getService(indirectRef);
```

Each time the `getService()` method is called, the OSGi Framework increments a usage count for that particular bundle.

OSGi Service Layer: Binding to a Service

Once you have the indirect reference to the service in the registry, you must then bind to it. The process of binding via an indirect reference ensures you are never tied to a specific implementation.

```
ServiceReference indirectRef =
  ctx.getServiceReference(StockListing.class.getName(), "(&(name=LSE)(objectClass=org.example.StockChart))");

StockListing listing = (StockListing)cxt.getService(indirectRef);

// Now we're done with the service
cxt.ungetService(indirectRef);
indirectRef = null;
```

Each time the `getService()` method is called, the OSGi Framework increments a usage count for that particular bundle.

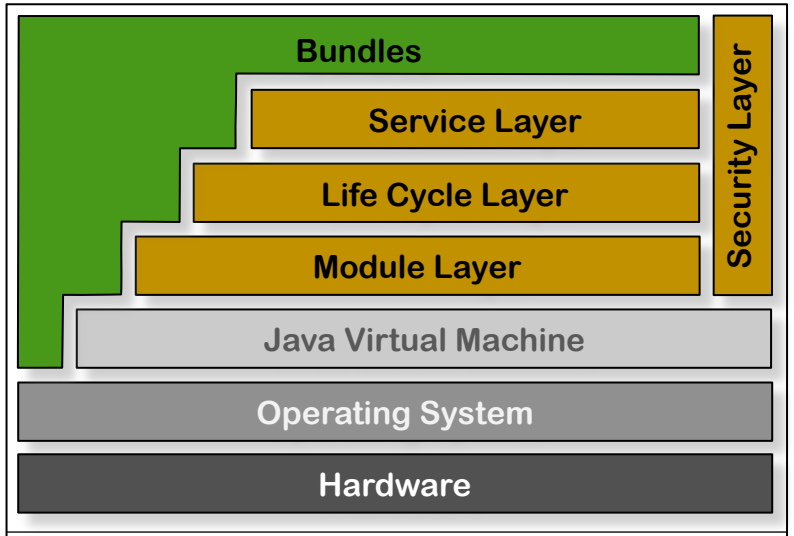
Therefore, when we have finished with a service, we must inform the OSGi Framework that our usage of this bundle has now come to an end.

Summary

OSGi: Summary 1/3

OSGi is a dynamic module system for Java providing:

- Modules (implemented as “Bundles”)
- Visibility (extends the Java concept of “public” to be “locally” or “globally” public within a bundle)
- Dependencies between bundles
- Versioning of bundles

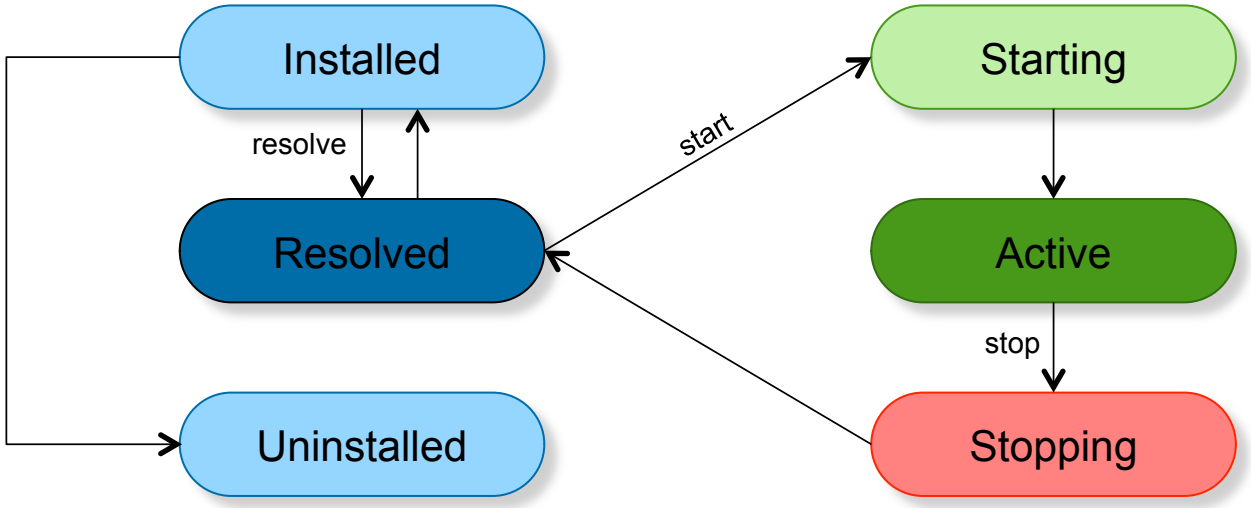


OSGi Architecture

OSGi: Summary 2/3

OSGi allows bundles to be dynamically:

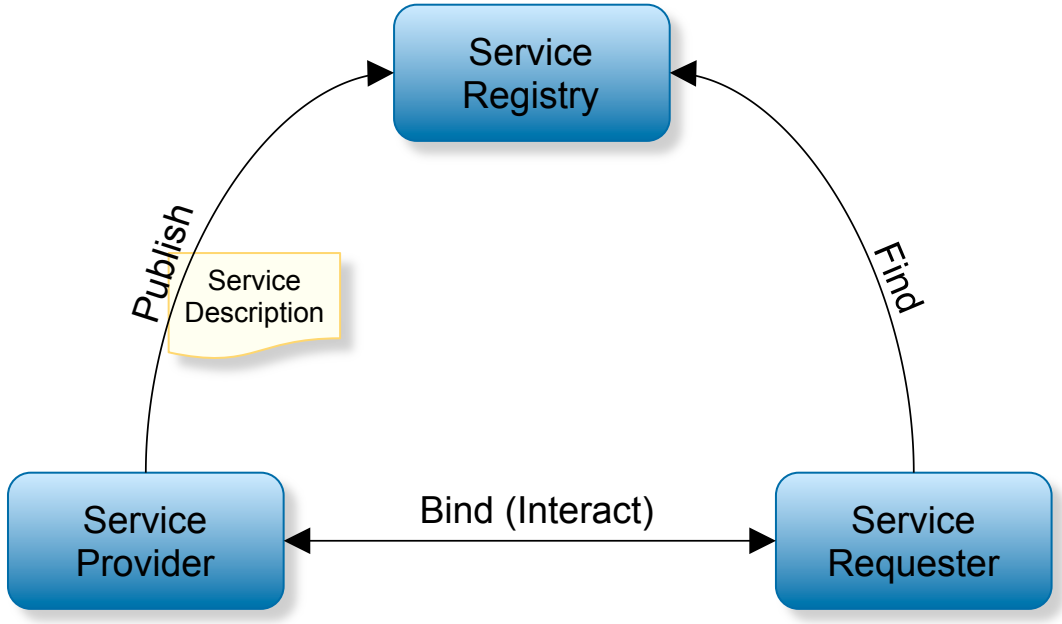
- Installed
- Started
- Stopped
- Upgraded
- Uninstalled



OSGi: Summary 3/3

OSGi is service oriented.

This allows the modules to interact using the ***Publish-Find-Bind*** pattern.



© 2013 SAP AG. All rights reserved

© 2013 SAP AG. Alle Rechte vorbehalten.