

With examples using Apache Aries

Enterprise OSGi IN ACTION

Holly Cummins
Timothy Ward



 MANNING



MEAP Edition
Manning Early Access Program
Enterprise OSGi in Action version 5

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 1: Programming beyond hello world

- 1 OSGi and the enterprise—why now?
- 2 Hello World Wide Web
- 3 Persistence pays off
- 4 Packaging your Enterprise OSGi applications

Part 2: Building better Enterprise OSGi applications

- 5 Best practices for writing enterprise applications
- 6 Building dynamic applications with OSGi services
- 7 Provisioning and resolution
- 8 Tools for building and testing
- 9 IDE development tools

Part 3: Integrating Enterprise OSGi with everything else

- 10 Hooking up remote systems with distributed OSGi
- 11 Migration and integration
- 12 Coping with the non-OSGi world
- 13 Choosing a stack

Appendixes

- A OSGi basics
- B The OSGi ecosystem

OSGi and the Enterprise – why now?



Having just opened this book, we're sure that many of you are wondering what Enterprise OSGi is. Enterprise OSGi covers a number of topics, all of which are commonly used in Enterprise programming. That probably isn't the most helpful of descriptions, so for those of you who have used Java Enterprise Edition before Enterprise OSGi can be thought of as a new way of using the Enterprise Java programming model so that it incorporates OSGi features. For those of you who have more experience using OSGi than Java EE you can think of Enterprise OSGi as a new way of using OSGi so it incorporates Java EE and other useful Enterprise features. Whichever way you look at it, we hope you'll find Enterprise OSGi pretty exciting. In this chapter, we'll discuss:

- Why modularity is important, and how Java stacks up
- How OSGi enforces some simple rules to make Java better at modularity
- Why Enterprise Java and OSGi traditionally don't play well together
- How Enterprise OSGi fixes this, and what the Enterprise OSGi programming model looks like

We'll start by taking a look at what modularity is, and why it is so important in software engineering.

1.1 Why the world needs modularity

One of the stories of software engineering has been that of increasing abstraction and resulting improvements in modularity. The earliest programs were written in assembly language, which mapped directly to the instruction set of the machine executing the program, or coded directly as machine instructions. There was no higher level structure above the individual machine codes, very little abstraction, and limited scope for sharing and re-using programs. Because there was so little abstraction between the code and the hardware it ran on even slight hardware changes could mean that the application had to be extensively rewritten. Higher level languages introduced subroutines, allowing code to be grouped into named functions and re-used. The abstraction away from raw machine instructions also meant that code could be re-compiled rather than re-written for new hardware. Next came libraries, very large groupings of code with a separate interface and implementation. This abstraction allowed different pieces of code to be changed independently from one another without rebuilding entire applications, these libraries are therefore modular code. Finally, object orientation provided finer-grained modularity by grouping data and behaviour together into encapsulated objects. This abstraction substantially improved the level to which changes in a program could be isolated from the rest of the program, and the level to which code could be reused, marking a dramatic improvement in modularity.

One of the reasons modularity has become increasingly necessary is the scale of modern computer programs. They are developed by globally dispersed teams and can occupy several gigabytes of disk space. In this kind of environment, it is critical that code can be grouped into distinct modules, with clearly delineated areas of responsibility and well-defined interfaces between modules.

Another striking change to software engineering within the last decade is the emergence of open source. Almost every software need can now be satisfied with open source applications. There are open source application servers, IDEs, databases, and messaging engines. There is also a bewildering range of open source projects which address particular development needs from Java bytecode generation to web presentation layers. Because the projects are open source, they can easily be re-used by other software. As a result most programs now rely on some open source libraries. Even commercial software often uses open source componentry; numerous GUI applications, for example, are based on the Eclipse Rich Client Platform, and many application servers incorporate the Apache Web Server.

1.1.1 Java's missing modularity

The increasing scale of software engineering projects and the increasing availability of tempting open source libraries have made modularisation essential. Unfortunately, programming language design hasn't entirely kept up with the need for such modularity. Java does a great job of providing modularity at the class and package level. Methods and class variables can be declared public, or access can be restricted to the owning class, its descendants, or members of its package. Beyond this, however, there is little facility for modularity. Classes may be packaged together in a JAR but the JAR provides no encapsulation. Every class inside the JAR is externally accessible, no matter how internal its intended use.

SPAGHETTI CODE

We've all heard code which is too coupled and interdependent described as 'spaghetti code' (figure 1.1).

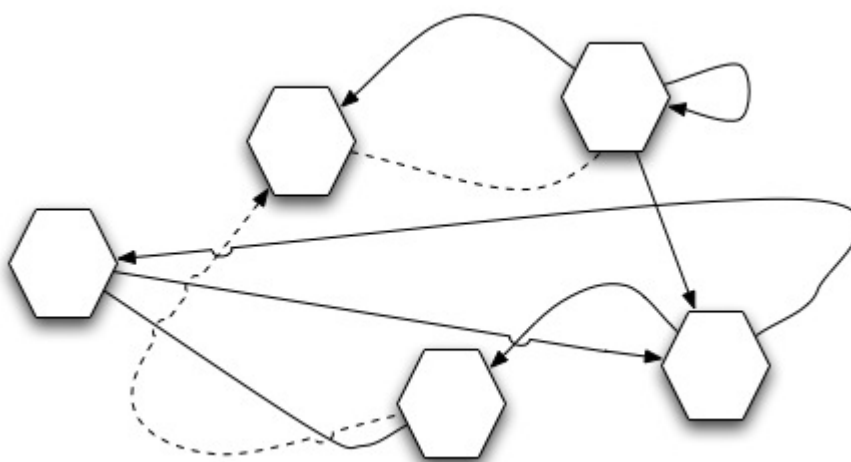


Figure 1.1 A highly interconnected 'spaghetti' application with little structure. The solid lines represent dependencies which are identifiable at both compile- and run-time, while the dotted lines are runtime-only dependencies. This sort of dependency graph is typical of procedural languages.

This sort of code is unfortunately common - both in open and closed source projects - and is universally despised. Not only is code like this hard to read and even harder to maintain, it is also very difficult to make even slight changes to its structure or move it to a new system. Even a slight breeze can be enough to cause problems! Given how strongly people dislike this sort of code it should be a lot less

common than it is, but sadly in a world where nothing stops you from calling any other function it is very easy to write spaghetti by accident. The other problem with spaghetti is that once you have some it tends to generate more very quickly...

Object orientation marked a big shift in the development of programming languages, providing a strong level of encapsulation in the language. Objects were responsible for maintaining their internal, private state, and could have internal, private methods. It was believed that this would mark an end to spaghetti code, and to an extent it did.

Extending the spaghetti metaphor, conventional Java programs (or any other OO language for that matter) can be thought of as 'object minestrone' (figure 1.2)- although there is a distinct object structure (the chunks of vegetable and pasta), there is no structure beyond the individual objects. The objects are thrown together in a soup and every vegetable can see every other vegetable.

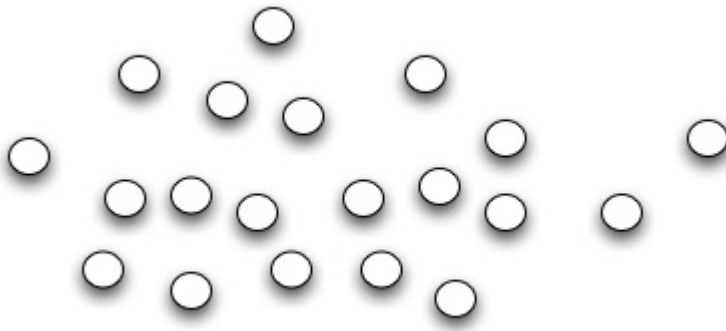


Figure 1.2 An application with no structure beyond individual well-encapsulated objects (connections between objects are not shown). This sort of structure is typical of object-oriented languages. While the objects themselves are highly modular, there is no more granular modularity.

CLASSPATH HELL

Insufficient encapsulation is not the only problem with Java's existing modularity. Few Java JARs are entirely freestanding; most will have dependencies on some other libraries or frameworks. Unfortunately, determining what these dependencies are is often a matter of trial and error. Inevitably, some dependencies may get left off the classpath when it is run. In the best case, this omission will be discovered early when a `ClassNotFoundException` is thrown. In the worst case, the codepath will be rarely travelled and the problem won't be discovered until weeks in when a `ClassNotFoundException` interrupts some particularly business-critical operation. Good documentation of dependencies can help here, but the only reliable way of ensuring every dependency is present is to package them all up in a single archive with the original JAR. This is inefficient and it's extra frustrating to have to do it for really common dependencies.

What's worse, even packaging JARs with all the other JARs they depend on isn't guaranteed to make running an application a happy experience. What if a dependency is in fact one of the common ones - so common that other applications running in the same JVM depend on it? This is fine, as long as the required versions are the same. One copy will come first on the classpath and be loaded, and the other copy will be ignored. What happens when the required versions are different? One copy will still be loaded, and the other will still be ignored. One application will run with the version it expects, and the other will not. In some cases, the 'losing' application may terminate with a `NoSuchMethodException` because it invokes methods which no longer exist. In other, worse, cases, there will be no obvious exceptions but the application will not behave correctly. These issues are incredibly unpleasant and in Java have been given the rather self-explanatory name 'Classpath Hell'.

1.1.2 Enterprise Java and modularity – even worse!

Enterprise Java and the Java EE programming model are used by a large number of developers, however there are many Java developers who will have no experience with either. Before we can explain why the enterprise suffers even more greatly than standard Java we need to make sure that we have a common understanding of what the enterprise is.

WHAT DISTINGUISHES ENTERPRISE JAVA FROM NORMAL EVERYDAY JAVA?

Part of the distinction, naturally, is the involvement of the enterprise – enterprise Java is used to produce applications used by businesses. But then businesses use many other applications, like word processors and spreadsheets. You certainly wouldn't say that a word processor, no matter how business-oriented, had been produced to a enterprise programming model. Similarly, many 'enterprise programmers' don't work for particularly large corporations.

So what's different about the enterprise applications? In general, they're designed to support multiple simultaneous users. With multiple users, some sort of remote access is usually required – having fifty users crammed into a single room isn't going to make anyone happy! Nowadays, remote access almost always means a web front end.

In order to store the information associated with these users, they usually persist data. Writing database access code isn't much fun, so persistence providers supply a nicer set of interfaces to manage the interaction between the application code and the database.

This is a business application, and so transactions are usually involved – either buying and selling of goods and services, or some other business agreements. In order to ensure these 'real' transactions proceed smoothly and consistently even in the event of a communications problem, software transactions are used.

With all this going on, these enterprise applications are starting to get pretty complex. They're not going to fit into a single Java class, or a single JAR file. It may not even be practical to run every part on a single server. Distribution allows the code, and therefore the work, to be spread across multiple servers on a network. Some people argue that distribution is in fact *the* key feature of what's known as enterprise computing, and the other elements, like transactions and the web, are simply there to facilitate distribution (like the web) or to handle some of the consequences of distribution on networks which aren't necessarily reliable (like transactions).

Java EE provides a fairly comprehensive set of standards designed to fit the scaling and distribution requirements of these enterprise applications, and is widely used throughout enterprise application development.

MODULAR JAVA EE - BIGGER ISN'T BETTER

Our enterprise application is now running across multiple servers, with a web front end, a persistence component, and a transaction component. Quite how all the pieces fit together may not be known by individual developers when they're writing their code. Which persistence provider will be used? What about the transaction provider? What if we change vendor next year? Java EE needs modularity for its applications even more than base Java does. Running on different servers means that the classpath, available dependencies and technology implementations are likely to diverge. This becomes even more likely as the application is spread over more and more systems.

With these interconnected applications, it's much better for developers to avoid specifying where all their dependencies come from and how they're constructed. Otherwise the parts of the application become so closely coupled to one another that changing any of them becomes very difficult. In the case of a little program, this close coupling would be called spaghetti code (see figure 1.1 again). In large applications it is sometimes known as the "Big Ball of Mud". In any case the pattern is equally awkward and the consequences can be just as severe.

Unfortunately for Java EE there is no basic Java modularity to fall back upon, the modules within a Java application often spaghetti between one another, and inevitably their open source library dependencies have to be packaged within the applications. To improve cost effectiveness each server in a Java EE environment will typically host multiple applications, each of which packages its own dependencies, and potentially requires a different implementation of a particular enterprise service. This is a clear recipe for Classpath Hell, however the situation is in fact even worse than it first appears. The Java EE application servers themselves are large, complicated pieces of software, and even the best of them contain a little spaghetti. In order to reliably provide basic functions at low development cost they also depend on open source libraries, many of the same libraries used by the applications that run on the application server! This is a really serious problem, as now developers and systems administrators have no way to avoid the conflict. Even if all applications are written to use the same version of an open source library they can still be broken by the different version (typically undocumented) in the underlying application server.

1.2 OSGi to the rescue

OSGi is a big subject, in fact there are entire books dedicated to it - including this one! This section will go over the basics of OSGi at a high level showing how OSGi solves some of the fundamental modularity problems in Java. We'll also delve into greater detail into some aspects of OSGi which may not be familiar to most readers, but which will be important to understand when we start writing enterprise OSGi applications. We'll also be explaining the syntax we use for the diagrams later in the book. This section will cover all the really important facts for writing Enterprise OSGi applications, but you're new to OSGi, or if after reading it you're bursting to know even more about the core OSGi platform you should read appendices XREF `osgi_appendix` and XREF `osgi_ecosystem_appendix`. We can't cover all of OSGi in two appendices, so we'd also definitely recommend you get hold of *OSGi in Action* by Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage.

So what is OSGi and what is it that OSGi provides which is so valuable? Put simply OSGi is a modularity framework for Java. In a sense, OSGi takes the Java programming model closer to an 'ideal' programming model - one which has robustness, power, and elegance. The way it does this is by encouraging good software engineering practice through higher levels of modularity. These, along with versioning, are the driving *principles* behind OSGi. OSGi enables abstraction, encapsulation, decomposition, loose coupling, and re-use.

1.2.1 Modularity, versioning and compatibility

OSGi solves the problems of Classpath Hell and Spaghetti code in one fell swoop using an incredibly simple, but equally powerful, approach centred around declarative dependency management and strict versioning.

OSGI BUNDLES - MODULAR BUILDING BLOCKS

Bundles are Java modules. On one level, a bundle is just an ordinary Java ARchive (JAR) file, with some extra headers and metadata in its JAR manifest. The OSGi runtime is usually referred to as the "OSGi framework", or sometimes just "the framework", and is a container that manages the lifecycle and operation of OSGi bundles. Outside of an OSGi framework a bundle behaves just like any other JAR, with all the same disadvantages, and no improvement to modularity. Inside an OSGi framework, however, a bundle behaves very differently. The classes inside an OSGi bundle are able to use one another just like any other JAR in standard Java, but the OSGi framework prevents classes inside a bundle from being able access classes inside any other bundle unless it is explicitly allowed to do so. One way of thinking about this is that it acts a bit like a new visibility modifier in between protected and public, allowing only code from the same JAR to see it.

Obviously if JAR files were not able to load any classes from one another they would be fairly useless, which is why in OSGi a bundle has the ability to deliberately expose packages outside itself for use by other bundles. The other half of the modularity statement is that in order to make use of an 'exported' package a bundle must define an 'import' for it. In combination these imports and exports provide a very strict definition of the classes that can be shared between OSGi bundles, but express it in a very simple way.

Listing 1.1 A simple bundle manifest showing how packages can be imported into and exported from a bundle.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: fancyfoods.example
Bundle-Version: 1.0.0
Bundle-Name: Fancy Foods example manifest
Import-Package: fancyfoods.api.pkg;version="[1.0.0,2.0.0)"
Export-Package: fancyfoods.example.pkg;version="1.0.0"
```

There are many more possible headers that are used in OSGi, a number of which will be described in later chapters.

By strictly describing the links between modules OSGi allows Java programs to be less like minestrone and more like a tray of cupcakes (figure 1.3). Each cupcake has an internal structure (cake, paper case, icing, and perhaps decorations), but is completely separate from the other cupcakes. Importantly a chocolate cupcake can

be removed and replaced with a lemon cupcake without affecting the whole tray. As you build relationships between OSGi bundles this is a bit like stacking the cupcakes on top of one another. Exporting a package provides a platform onto which an Import can be added. As you build up a stack of cupcakes then cupcakes in the higher layers will be resting on other cupcakes in lower levels, but these dependencies can be easily identified. This prevents you from accidentally removing the cupcake on the bottom and causing an avalanche!

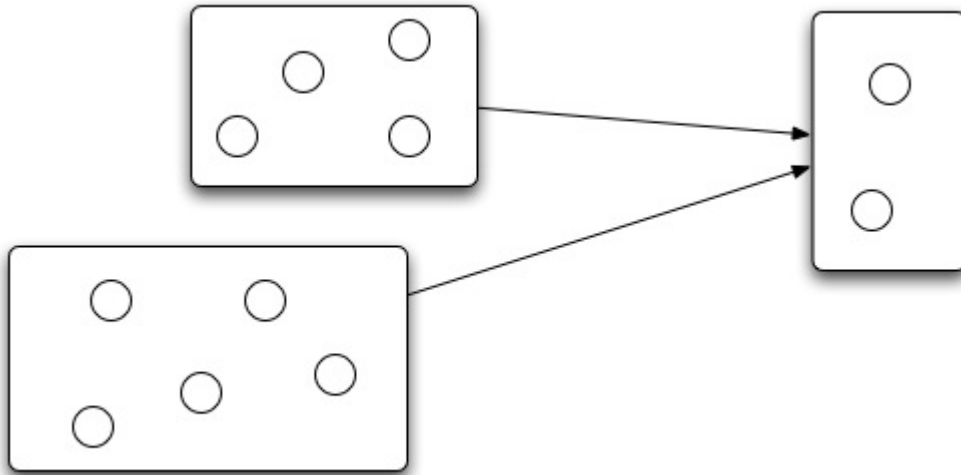


Figure 1.3 A well-structured application with objects grouped inside modules. Dependencies between modules are clearly identified. This is typical of the application structure which can be achieved with OSGi.

By enforcing a higher level granular structure on Java application code OSGi bundles strongly encourage good software engineering practice. Rather than spaghetti code being easy to produce accidentally it is only possible to load and use other classes that are explicitly intended for you to use. The only way to write spaghetti in OSGi is to deliberately expose the guts of your OSGi bundle to the world, and even then the other bundles still have to choose to use that package. In addition to making it harder to write spaghetti OSGi also makes it easier to spot spaghetti. A bundle that exports a hundred packages and imports a thousand is obviously not very cohesive or modular!

In addition to defining the API that they expose, OSGi bundles also completely define the packages that are needed for them to be used. By enforcing this constraint OSGi makes it abundantly clear what dependencies are needed for a given bundle to run, and also transparent as to which bundles can supply those dependencies. Importing and exporting packages goes a long way to solving Classpath Hell as you no longer have to guess which JAR file is missing from your

classpath, however in order to completely eradicate Classpath Hell OSGi has another trick up its sleeve. Versioning.

VERSIONING IN OSGI

Versioning is a necessary complement to modularity. It doesn't sound as enticing as modularity - in fact, if we're being perfectly honest, it sounds a bit dull - but it is essential if modularity is to work at all in anything but the simplest scenarios. Why?

Let's imagine we've achieved perfect modularity in our software project. All our components are broken out into modules, which are being developed by different teams, perhaps even different organisations. They're being widely re-used in different contexts. What happens when a module implements a new piece of function which breaks existing behaviour, either by design or as an unhappy accident? Some consuming modules will want to pick up the new function, but others will need to stick with the old behaviours. Coordinating this needs the module changes to be accompanied by a version change.

Let's go a step further. What if the updated module is consumed by several modules within the same system, some of which want the new version, and some the old version? This kind of co-existence of versions is very important in a complex environment, and it can only be achieved by having versions being first-class properties of modules, and compartmentalising the class space.

SIDEBAR Versions, versions everywhere!

Versioning is incredibly important in OSGi, in fact it is so important that if you don't supply a version in your metadata then you will still have the version 0.0.0! Another important point is that versioning doesn't just apply to packages, OSGi bundles are also versioned. This means that in a running framework you might not just have multiple versions of the same package, but multiple versions of the same bundle as well!

THE SEMANTIC VERSIONING SCHEME

Versioning is a way of communicating about what's changing (or not changing) in software, and so it's essential that the language used be shared. How should modules and packages be versioned? When should the version number change? What's most important is being able to distinguish between changes which will break consumers of a class by changing an API, and changes which are internal only.

The OSGi alliance recommend a scheme called semantic versioning, the details

are available at <http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf>. Semantic versioning is essentially a very simple scheme, but it conveys much more meaning about what's changing than normal versions. Every version consists of four parts, major, minor, micro, and qualifier. A change to the major part of a version number - for example, changing 2.0.0 to 3.0.0 - indicates that the code change is not backwards compatible. Removing a method or changing its argument types is an example of this kind of breaking change. A change to the minor part indicates a change which is backwards compatible for consumer of an API, but not for implementation providers. For example, the minor version should be incremented if a method is added to an interface in the API, as this will require changes to implementations. If a change does not affect the externals at all, it should be indicated by a change to the micro version. Such a change could be a bug fix, or a performance improvement, or even some internal changes that remove a private method from an API class. Having a strong division between bundle internals and bundle externals means the internals can be changed quite dramatically without anything other than the micro version of the bundle needing to change. Finally, the qualifier is used to add extra information like a build date.

Whilst our explanation focuses on API, it isn't just packages that should be semantically versioned. The versions of bundles also represent a promise of functional and API compatibility. It's particularly important to remember that semantic versions are quite different from marketing versions. Even if a great deal of work has gone into a new release of a product, if it is backwards compatible the version would only change from 2.3 to 2.4 (say), rather than from version 5 to version 6 (say). This can be a bit depressing for the release team, but it's very helpful for users of the product who need to understand the nature of the changes. Also, think of it this way - a low major version means you don't make a habit of breaking your customers!

GUARANTEES OF COMPATIBILITY

One of the benefits provided by the semantic versioning scheme is a guarantee of compatibility. A module will be bytecode compatible with any versions of its dependencies where the major version is the same, and the minor version is the same or higher. One warning about importing packages is that modules should not try to import and run with dependencies with lower minor versions than the ones they were compiled against.

SIDEBAR Forward compatibility

Version ranges are very important when importing packages in OSGi, as they define what the expected future compatibility of your bundle is. If you don't specify a range then your import runs to infinity, meaning that your bundle expects to be able to use any version of the package, regardless of how it changes! It is good practice to *always* specify a range, using square brackets for inclusive or parentheses for exclusive versions. For example `[1.1, 2)` for an API client compiled against a package at version 1.1 that would be compatible up to, but not including, version 2.

CO-EXISTENCE OF IMPLEMENTATIONS

The most significant benefit provided by versioning is that it allows different versions of the same module or package to co-exist in the same system. If the modules weren't versioned, there would be no way of knowing that they are different and isolating them from one another. With versioned modules (and some classloading magic courtesy of OSGi), each module can use the version of its dependencies which is most appropriate (figure 1.4).

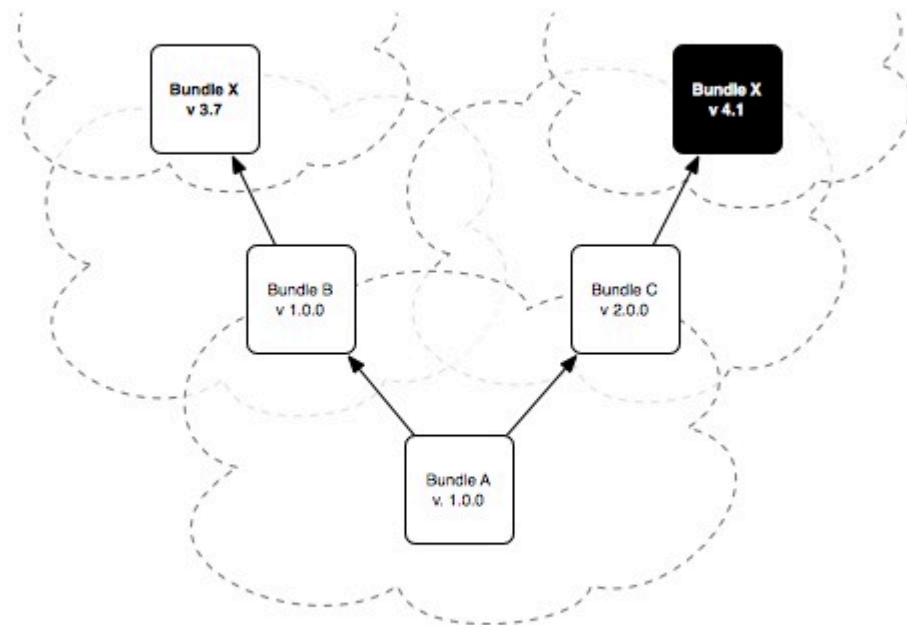


Figure 1.4 The transitive dependencies of a module (the dependencies of its dependencies) may have incompatible versions. In a flat classpath, this can be disastrous, but OSGi allows the implementations to co-exist by isolating them.

So as you can see, being explicit about dependencies, API and versioning allows OSGi to completely obliterate Classpath Hell, but of course, OSGi on its own doesn't guarantee well-structured applications. What it does do is give developers the tools they need to be able to define a proper application structure. It also makes it easier to identify when application structures have slid in the direction of highly-coupled soupishness. This is a pretty big improvement over standard Java, and OSGi is worth considering on the basis of these functions alone. OSGi, however, has a few more tricks up its sleeve. Curiously enough, modularity was only one of the aims when creating OSGi, another focus was dynamic runtimes.

1.2.2 Dynamism and lifecycle management

Dynamism is not new to software engineering. However, it is fundamental to OSGi. Just as versioning is part of OSGi to support proper modularity, modularity is arguably an OSGi feature because it's required to support full dynamism. Many people are unaware that OSGi was originally designed to operate in small, embedded systems where the system could actually physically change. A static classpath really wasn't good enough in this kind of environment!

Why did OSGi need a new model for dynamism? After all, in some ways, Java is pretty dynamic. For example, reflection allows fields to be accessed and methods to be invoked on any class by name. A related feature, proxies, allows classes to be generated on the fly which implement a set of interfaces. These can be used to stub out classes, or create wrappers dynamically. Arguably another even more powerful dynamic feature of Java is URL classloaders. Classes may be loaded from a given URL at any point in time, rather all being loaded at JVM initialisation from a static classpath. Furthermore, anyone can write a classloader.

Java's ability to write custom classloaders and add classes dynamically to a running system is not to be sniffed at. In fact, it is this feature which makes much of OSGi possible. However, Java's classloading APIs are too low-level to be widely useful on their own. What OSGi provides is a layer which harnesses this dynamism and makes it generally available to an audience who aren't interested in writing their own classloaders or hand-loading all the classes they need.

BUNDLE LIFECYCLES

Unlike most JAR files on the standard Java classpath, OSGi bundles are not static entities which live on the classpath indefinitely. Dividing classloading responsibility among multiple classloaders enables the entire system to be highly dynamic. Bundles can be stopped and started on demand, with their classloaders and classes simply appearing and disappearing from the system as required. Bundles which have been started are guaranteed to have their requirements met; if a bundle's dependencies cannot be satisfied, it won't be able start. The complete state machine for bundle lifecycles is sufficiently simple to display in a single picture.

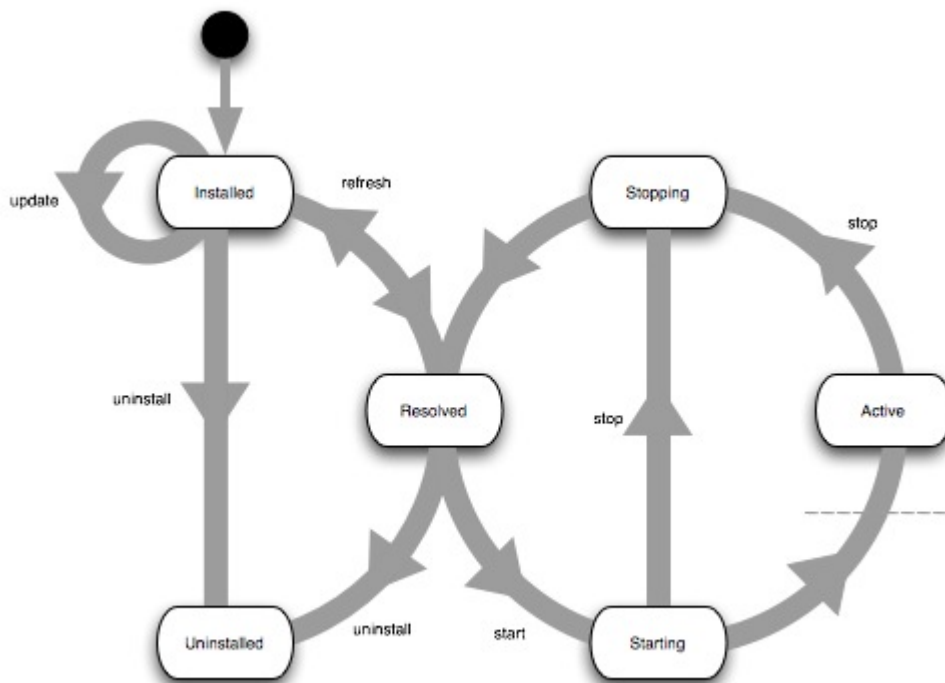


Figure 1.5 Bundles may move between the installed, resolved, starting, active, and stopping states. A bundle is resolved if it is installed and all its dependencies are also resolved or started. When a bundle is uninstalled it is no longer able to start, nor can it provide packages to any new bundles.

INSTALLATION AND RESOLUTION

Installation is, most simply, the process by which new bundles are added to an existing OSGi framework at runtime. This can be done in a number of framework specific ways, but there is a commonly used, framework independent, mechanism for installing a bundle from a remote location.

```
org.osgi.framework.BundleContext.installBundle(String location)
```

After installation into a framework an OSGi bundle is in the 'INSTALLED' state, and is a pretty boring, inert object. An INSTALLED bundle does not have a classloader and cannot provide code or packages to anyone. What happens to the bundle next is the real magic of OSGi, and is known as the resolution process. Once all of a bundle's dependencies are available in the OSGi framework (that is to say there are exported packages available for all of its imports), then the framework resolver will attempt to 'resolve' the bundle. This process creates fixed wires between package imports and exports, obeying the versioning criteria declared in the metadata. If a consistent set of wires can be created for a bundle then that bundle is said to be 'RESOLVED'. A RESOLVED bundle is much more interesting than an INSTALLED bundle - it has a classloader, and so classes and resources can be loaded from the moment the bundle resolves. The wires created by the resolver have a huge impact on how classes are loaded in OSGi, but also guarantee that dependencies are available at runtime and eliminate the risk of a `NoClassDefFoundError`.

CLASSLOADING

OSGi's classloading is at the heart of what makes it different from standard Java. It's a very elegant and scalable system. Unfortunately it's also one of the greatest sources of problems when adapting applications which weren't designed with modularity in mind to run in an OSGi environment.

Instead of every class in the virtual machine being loaded by a single monolithic classloader, classloading responsibilities are divided among a number of classloaders (figure 1.6). Each bundle has an associated classloader, which loads classes contained within the bundle itself. If a bundle has a package import that is wired to a second bundle by the framework resolver, then its classloader will delegate to the other bundle's classloader when attempting to load any class or resource in that package. In addition to the bundle classloaders, there are environment classloaders which handle core JVM classes.

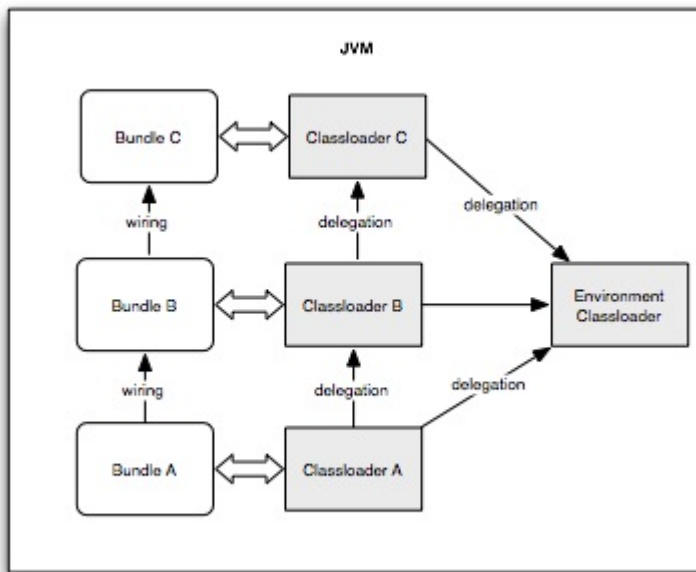


Figure 1.6 The JVM contains many active classloaders in an OSGi environment. Each bundle has its own classloader. These classloaders delegate to the classloaders of other bundles for imported packages, and to the environment's classloader for core classes.

Each classloader has well-defined responsibilities. If a classload request is not delegated to another bundle, it will pass it up the delegation chain. Somewhat surprisingly, this means that being included in a bundle does not guarantee that a package will be loaded by that bundle. If that bundle also has an import for the package which is wired by the framework resolver then all class loads for that package will be delegated elsewhere! This is a principle known as substitutability. It allows bundles to maintain a consistent class space between them by standardising on just one variant of a package, even when multiple variants are exported. Figure 1.7 shows the class space for a bundle which exports a substitutable package.

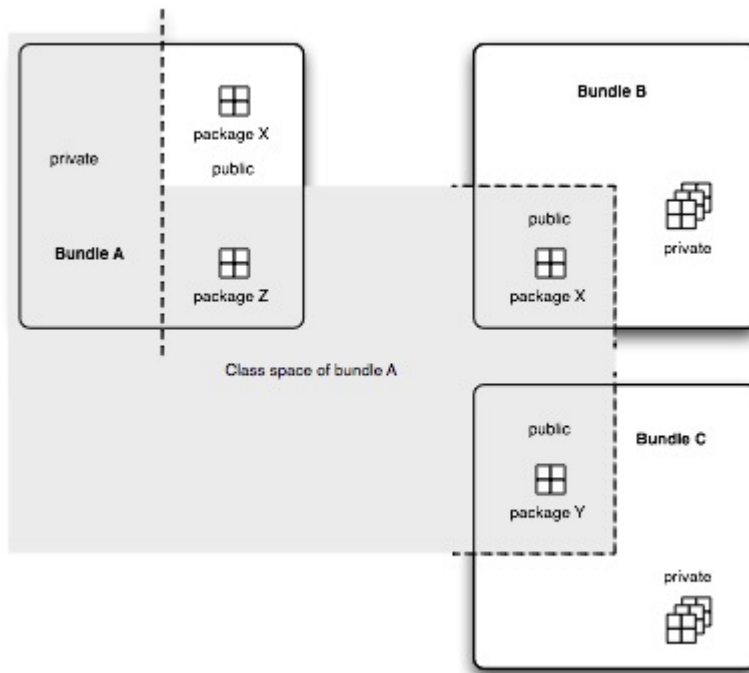


Figure 1.7 The class space for a bundle includes all of its private classes, and the public classes of any bundle it is wired to. It does not necessarily include all the bundle's public classes, since some might be imported from other bundles instead.

SERVICES AND THE SERVICE REGISTRY

Bundles and bundle lifecycles are as far as many OSGi developers go with OSGi. Enterprise OSGi, however, makes heavy use of another fundamental OSGi feature, services. OSGi services are much more dynamic than their JEE alternatives. OSGi services are a bit like META-INF/services without all the messy files, or like JNDI with more power and less ... JNDI. Although OSGi services fill the same basic requirement as these two technologies, they have important extra features such as dynamism, versioning and property-based filtering. They are a simple and powerful way for bundles to transparently share object instances without having to expose any internal implementation - even the name of the class implementing the API. By hiding the service implementation and promoting truly decoupled modules OSGi services effectively enable a single-JVM service-oriented-architecture. Services also enable a number of other useful architectural patterns.

Figure 1.8 shows a simple OSGi service, represented by a triangle. The pointy end faces towards the provider of the service. One way of thinking of this is that the arrow points in the direction of invocation when a client calls the service. Another way to think of it is that the provider of a particular service is unique,

whereas there may be many clients for it, as a result the triangle must point at the only "special" bundle. Alternatively, if you squint really hard the service might look a bit to you like the spout of an old-fashioned watering can, spreading water - or a service - out from a single source to many potential recipients.

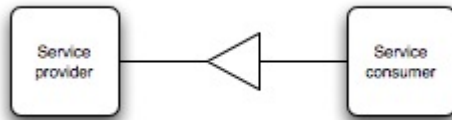


Figure 1.8 A service which is provided by one bundle and used by another bundle. The narrow end of the triangle points towards the service provider.

PROVIDING SERVICES

Services are registered by a bundle using one or more class names which mark the API of the service, and the service object itself. Optional properties can provide extra information about the service and can be used by clients to filter which services get returned when they are looking for one. Service properties aren't intended for use by the service itself.

```

Dictionary<String, String> props = new Hashtable<String, String>();
props.put("check.type", "slow");
ctx.registerService(InventoryLister.class.getName(), props);
  
```

As you can see, providing a service is very easy. Now providing a service isn't very useful unless people have a way of finding and using it.

ACCESSING SERVICES

Services can be looked up using a simple API. Enterprise OSGi also allows services to be accessed declaratively and injected as a dependency. We'll be making use of service dependency injection throughout this book, starting in section XREF blueprint_section. Before we get there, let's have a peek at what a service lookup looks like without dependency injection:

```

String interfaceName = InventoryLister.class.getName();
ServiceReference ref = ctx.getServiceReference(interfaceName);
InventoryLister lister = (InventoryLister) ctx.getService(ref);
  
```

What happens when multiple providers of the service have been registered?

Service consumers have a choice between getting just one, or getting a list containing all of them. If the service is something like a credit card processing service, it's only necessary to take a payment once. In this situation one service provider is sufficient, and it probably doesn't matter too much which provider is chosen. In the case of a logging service, on the other hand, logged messages should probably be sent to all the available loggers, rather than just one of them. Fortunately OSGi also allows us to find all of the services that match a particular request.

```
ServiceReference[] refs = ctx.getServiceReferences(Logger.class
    .getName());
if (refs != null) {
    for (ServiceReference ref : refs) {
        Logger logger = (Logger) ctx.getService(ref);
        logger.doSomeLogging();
    }
}
```

1 Properties can be used to refine lookups

So as you can see, in addition to its modular, versioned runtime, OSGi provides a host of lifecycle management to provide a lightweight dynamic framework. All of these encourage good engineering practice, however, as with most things, using OSGi doesn't guarantee that your application will be well-structured. What OSGi does is to give developers the tools they need to be able to define a proper application structure. It also makes it easier to identify when application structures have slid in the direction of highly-coupled soupishness. So given its obvious advantages, why isn't everyone using OSGi already?

1.2.3 *Why isn't everyone using OSGi?*

As we've mentioned previously OSGi isn't a new technology; the ideas have been around for more than a decade now. But OSGi adoption within the Java community is not as ubiquitous as you would expect given its obvious advantages. There are several reasons for this.

THE ORIGINS OF OSGI

The OSGi Alliance's original mission was to allow Java to be used in embedded and networked devices. In addition to Sun, IBM, and Oracle, its original members were, for the most part, mobile phone manufacturers like Motorola and Ericsson, and networking companies like Lucent and Nortel. There were also energy companies involved, such as Electricité de France and the late Enron Communications. Within a few years, OSGi was being used inside set-top boxes, Siemens medical devices, Bombardier locomotives, and the entertainment system of the BMW 5-series. The main reason for this is that the advantages of OSGi are particularly useful in constrained devices, or in applications where the system must remain running for long periods, including through maintenance updates.

SIDEBAR **The meaning of OSGi**

Because of its embedded origins the letters 'OSGi' used to stand for Open Services Gateway initiative. Now, if you didn't already know what OSGi was about, the phrase 'Open Services Gateway initiative' doesn't really shout 'dynamic module system for Java'. In fact, the name is so divorced from what OSGi is used for today that the original expansion of the acronym has been abandoned, and OSGi now stands for, well, OSGi.

The next wave of OSGi adoption happened in large-scale software projects, particularly IDEs, application servers and other middleware. It's initially surprising that a technology designed for the tiniest Java installations should be such a good fit for the largest ones. Do the software running a car stereo and an enterprise application server really have much in common? As it happens, yes. What embedded devices needed was modularity and dynamism; software with very large codebases has the same requirements. Despite the huge increase in processing power and memory available to modern devices, OSGi is, if anything, even more useful in these big systems. The increasing complexity of software projects is a key driver for OSGi adoption in Java applications.

Until recently, desktop applications suffered from neither the level of constraint of embedded systems, nor the complexity of hefty middleware systems. In these environments the advantages of OSGi are sometimes viewed as insufficient when compared to its perceived drawbacks.

OSGI - BAD PRESS AND POOR UNDERSTANDING

Over its life OSGi has suffered from some misconceptions and rather poor press, one of the big hurdles initially for OSGi was the perception that it was purely a technology for the embedded market, and that it was not needed, not useful, or worst a hindrance in the desktop and server space. This perception is clearly wrong, and was proved so when the Eclipse runtime chose to use an OSGi framework as its basis. It should be noted that Eclipse initially did not believe an OSGi platform would be suitable, but were later convinced by the huge increases in startup speed, reduced footprint, and complexity management offered by OSGi.

Despite OSGi's success in Eclipse, OSGi is still perceived by many as being 'too complex' for use in normal Java applications. Whilst OSGi metadata is quite simple, it is not part of base Java. This means that even experts in Java are usually novices in the use of OSGi. Further to this, the OSGi classloading model is significantly different to base Java, and requires developers to understand the dependencies that their code has. This is, in general, very good practice but it is also something that developers are not used to. This change in mindset does not always come easily, and means that many people write OSGi off as too hard to be useful. Whilst this may seem a little short-sighted, there are, in fact, good reasons why OSGi seems excessively complex, particularly when applications try to make use of existing open source libraries.

UTILITY LIBRARIES AND OSGI

One of the big successes for Java has been that a large open source community has grown around it, providing a cornucopia of libraries, frameworks and utilities. These libraries are so commonly used that there are almost no applications that do not use any and most applications use several. Some libraries are so commonly used that many developers consider them to be part of the Java API.

Most open source libraries in Java were not originally written with OSGi in mind, although an ever increasing number are re-designing and re-packaging themselves in an OSGi friendly way. Historically this means that these libraries, for the large part, had to be converted manually by the developers using them. As the developers have to guess about which packages are API and which package dependencies exist this process is time consuming and error prone - a key reason that OSGi is perceived to be hard.

Unfortunately for developers new to OSGi the problems with open source libraries do not end with packaging. Many libraries require users to provide some

configuration, and sometimes implementations of classes. In standard Java, with its flat classpath, these resources and classes can be loaded easily using the same classloader that loaded the library class. In OSGi these resources and classes are private to the application bundle, and cannot be loaded by the library at all! Problems like this are widespread in libraries that were not written with OSGi in mind and cannot usually be solved without re-engineering the library significantly.

Fortunately for OSGi open source libraries are increasing available with OSGi friendly packaging and API, meaning that OSGi is becoming easier to use than it ever has been before. This effort is not purely being made by the open source community, but also in new OSGi specifications where the OSGi alliance is providing new, OSGi aware, mechanisms to make use of existing technologies.

1.2.4 Why OSGi and Java EE don't work together

Unfortunately, utility libraries aren't the only barrier to OSGi adoption. Many Java applications run on application servers, and use some form of the enterprise Java, even if it's only servlets or dependency injections. Sadly, the Java EE programming model has historically been pretty incompatible with OSGi. Enterprise Java exists because the Java community recognised common needs and practices across a variety of business applications. Java EE provides a common way to make use of enterprise services. Unfortunately Java EE aggravates the modularity problems present in standard Java, and is more resistant to OSGi solutions.

FRAMEWORKS AND CLASS LOADING

Enterprise Java Application Servers are large beasts, and typically host multiple applications, each of which may contain many modules. In order to provide some level of isolation between these applications there is a strict classloading hierarchy, one which separates the applications and modules from each other, and from the Application Server. This hierarchy is strongly based on the hierarchy of the classloaders in standard Java. Classes from the Application Server are shared between the applications through a common parent classloader, however this means that application classes cannot easily be loaded from the Application Server classes. This may not seem like a big problem, but Java EE contains an awful lot of containers and frameworks that provide hook and plug points to application code. In order to load these classes the frameworks, which are part of the base Application Server runtime, have to have access to the application.

This problem is bypassed in Enterprise Java using the concept of the Thread Context Classloader. This classloader has visibility to the classes inside the

application module and is attached to the thread whenever a managed object (like a servlet or EJB) is executing. This classloader can then be retrieved and used by the framework code to access classes and resources in the application module that called into it. This solution works, but it means that many useful frameworks rely heavily on the Thread Context ClassLoader being set appropriately. In OSGi the Thread Context Classloader is rarely set, and furthermore it completely violates the modularity of your system. There is no guarantee that classes loaded by the Thread Context ClassLoader will match your class space, particularly there is no assurance that you will share a common view of the interface that needs to be implemented. This causes a big problem for Java EE technologies in OSGi.

META-INF SERVICES AND THE FACTORY PATTERN

Although reflection, dynamic classloading and the Thread Context ClassLoader are useful, they are of limited practical use in writing loosely coupled systems. For example, reflection doesn't allow an implementation of an interface to be discovered, unless the implementation's class name is already known. Having to specify implementation names in advance pretty much defeats the point of using interfaces. This problem crops up again and again in Java EE and unsurprisingly there is a common pattern for solving it.

For many years, the best solution to the problem of obtaining interface implementations was to isolate the problem to one area of code, known as a factory. The system wasn't really loosely coupled, but at least only one area was tightly coupled. The factory would use reflection to instantiate a class whose name had been hardcoded into the factory. This didn't do much to eliminate the logical dependency between the factory and the implementation, but at least the compile-time dependency went away.

A better pattern was to externalise the implementation's class name out to a file on disk or in a JAR, which was then read in by the factory. The implementation still had to be specified, but at least it could be changed without recompiling the factory. This pattern was formalised into what's known as META-INF services. Any JAR can register an implementation for any interface by providing the implementation name in a file named after the interface it implements, found in the META-INF/services folder of the JAR. Factories can look up interface implementations using a ServiceLoader and all registered implementations will be returned.

This mechanism sounds similar to OSGi services. So why isn't this good enough for OSGi? One practical reason is that the service registry for META-INF

services wasn't available when OSGi was being put together. The other, more relevant, issues are that while META-INF services avoids tight coupling in code it doesn't give any dynamism beyond that, and it still relies on one JAR being able to load the internal implementation class of another. Furthermore, META-INF services can't be registered programmatically, and they certainly can't be unregistered.

Whilst the META-INF services pattern isn't ideal, it is extremely widely used, as a result the Enterprise OSGi 4.3 specification is set to include support for dynamically exposing META-INF services as OSGi services.

1.3 Programming with Enterprise OSGi

From what you've read so far it probably sounds like OSGi is a bit of a lost cause for enterprise programming. Sure it has some really cool ideas and lets you do some things that you couldn't easily do before, but who could give up the smorgasbord of enterprise services that just don't work in OSGi? Obviously this isn't the case, or this would be a very short book! Recently the gap between Java EE and OSGi has grown much smaller, with the introduction of *Enterprise OSGi*.

One of the fascinating things about OSGi's development from a platform for home gateways to a platform for trains and cars to to a platform for IDEs and application servers is that OSGi itself hasn't actually had to change that much. Even though the domains were totally different, the capabilities provided by OSGi solved problems in all in all of them. Enterprise Java is a bit different because basic OSGi by itself isn't quite enough. In order to address the needs of the Enterprise, the OSGi alliance has branched out and produced an Enterprise Specification with Enterprise-specific extensions to core OSGi.

1.3.1 Enterprise OSGi and OSGi in the Enterprise

Like the term “Enterprise Java”, “Enterprise OSGi” can mean different things to different people. Some people simply refer to the use of core OSGi concepts, or an OSGi framework, to provide business value for one or more applications. This definition is, however, a little looser than is normally accepted, and many people suggest that merely using an OSGi framework to host business applications is not sufficient to justify the description “Enterprise OSGi”. A parallel can be drawn between the definition of Enterprise Java programming, a term which is intimately linked to the use of the Java Enterprise Edition programming model, and usually an Application Server or Servlet Container. Simply using a Java Virtual machine using the standard Java SE APIs to write a business application would not normally be considered an “Enterprise Java Application”. Similarly when business applications are merely using an OSGi framework and features from the core OSGi specifications they're not really “*Enterprise OSGi*”. This scenario is best described by the term “*OSGi in the Enterprise*”.

The term Enterprise OSGi has evolved and is used by some to refer to the set of specifications defined by the OSGi Enterprise Expert Group in the OSGi Enterprise 4.2 and 4.3 releases. This strict definition would exclude some of the features in Open Source which are not in the Enterprise OSGi specification but which clearly have an enterprise OSGi 'feel'. Probably the most common and accurate definition of “Enterprise OSGi” is actually a blend of what's in the specification with the more general usage of OSGi in the enterprise we just discussed. What we mean by “Enterprise OSGi” in this book is an OSGi-based application which makes use of one or more enterprise services, as described in the Enterprise OSGi specifications, to provide business value. This links both the use of OSGi concepts, and an OSGi framework, with the the OSGi Enterprise Specification which defines how enterprise services can be used from inside an OSGi framework. So what does this enterprise OSGi programming model look like?

1.3.2 Dependency Injection

Dependency injection, sometimes called “Inversion of Control”, defines both an enterprise technology, and an architectural model. Dependency injection has only recently become part of the official JEE standard, but it has been a de facto part of enterprise programming for many years through the use of frameworks like Spring. Dependency injection is, if anything, even more valuable in an OSGi environment. As the OSGi service registry is such a dynamic environment, it is actually very difficult to correctly write a bundle that makes use of a service in a safe way, monitoring its lifecycle and finding an appropriate replacement.

Using and providing services becomes even more difficult when an implementation depends upon more than one service, with a user having to write some very complex thread safe code. Using dependency injection eliminates this complexity by managing the lifecycle of both the services exposed, and consumed, by a bundle.

Because of its ability to dramatically simplify programming in OSGi, dependency injection is at the heart of the Enterprise OSGi programming model. Without it business logic is difficult to separate from dependency management logic, and development is slow. The other key advantage of dependency injection is that it allows application code to avoid dependencies on core OSGi APIs, dramatically reducing the amount that developers need to learn before they can start making use of OSGi and making applications easier to unit test.

1.3.3 Java EE integration

As we mentioned above, Enterprise OSGi wouldn't be very useful if it didn't provide at least some of the functions available in Java EE. Furthermore, Java EE already has a lot of experienced developers, and it wouldn't be very helpful if the OSGi way of doing things was completely different to the Java EE way. These two requirements are fundamental in the OSGi Enterprise Specifications, which aim to provide core enterprise services, re-using Java EE idioms and structure where possible.

The OSGi Enterprise specification is quite large, and covers many of the Java EE services that developers know and love. For example, it includes an OSGi-friendly version of a WAR, declarative JPA database access and access to JTA transactions. Rather than listing all the available services here this book will introduce them with worked examples, most of which will be instantly familiar to anyone with a Java EE background.

1.4 Summary

As I'm sure you'll have noticed by now this chapter doesn't contain a “Hello World” sample. We hope that rather than rushing to get your money back, you understand the reason for this. The Enterprise is a very big place, containing lots of platforms, technologies and some very difficult problems. In the Enterprise “Hello World” just doesn't cut the mustard. There will be plenty of sample code throughout the rest of the book, and we promise that in the next chapter our first sample will be rather more than five lines of code and a call to `System.out!`

What this chapter does contain is a discussion of what the problems with the current enterprise technologies are. For some of you, with years of OSGi experience, this information will probably just have been confirming what you already know. For others, we hope you now have a sense of how easily applications can become complex particularly when they are “Enterprise scale”.

With its long history, and the breadth of support now offered by OSGi for enterprise technologies, it should be clear that OSGi is ready for the Enterprise. It should also be clear that whether you're an OSGi developer whose application needs are getting too big, or you're a Java EE developer tired of object minestrone and debugging your application's classpath for the thousandth time, that you are ready for Enterprise OSGi.