eclipseRT

the eclipse series

OSGi and Equinox

Creating Highly Modular Java™ Systems

Jeff McAffer · Paul VanderLei · Simon Archer

Series Editors Jeff McAffer · Erich Gamma · John Weigand

Library of Congress Cataloging-in-Publication Data

OSGi and Equinox : creating highly modular Java systems / Jeff McAffer,
Paul VanderLei, Simon Archer.
p. cm.
Includes index.
ISBN 0-321-58571-2 (pbk. : alk. paper)
1. Java (Computer program language) 2. Computer software—Development.
I. VanderLei, Paul. II. Archer, Simon (Simon J.) III. Title.
QA76.73.J38M352593 2010
005.2'762—dc22
2009047201

ISBN-13: 978-0-321-58571-4
ISBN-10: 0-321-58571-2
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing February 2010

Foreword

My role as the Chief Technology Officer of SpringSource brings me into frequent contact with companies building enterprise applications: many familiar names from the Fortune 500, and a whole host of others besides. If there is one thing you quickly learn, it is that the world of enterprise applications is messy and complex. Even four to five years ago, customers adopting Spring were asking us for ways to help them manage the size and complexity of the applications they were building. Large team sizes and applications with hundreds or thousands of internal components (Spring beans) were not uncommon. The pressures on enterprises to deliver increasingly sophisticated applications, in shorter and shorter time frames, have only been growing since then. In many cases applications are now always live and are constantly evolving. The move to deliver software "as a service"—internally or externally—can only accelerate this trend.

In the enterprise Java landscape, the traditional unit of deployment for an enterprise application is a web application archive (WAR) file. A number of common themes arise in my discussions with enterprise development teams:

- The WAR file as a single large unit of packaging and deployment is slowing down development processes and making it more difficult to structure large development teams since everything must come together in a single packaging step before anything can be deployed.
- WAR files are getting too large and unwieldy—a typical enterprise application may have literally hundreds of third-party dependencies, all packaged inside the WAR file. This has an adverse effect on upload and deployment times.
- Attempting to tackle complexity by deploying multiple WAR files side by side in the same container leads to problems with heap usage in the JVM since each WAR file has its own copy of all the dependencies, even though many of them could in theory be shared.
- When deploying WAR files side by side, there is no easy way to share common services.

- The WAR file as the smallest unit of change means that changes in large enterprise applications cannot be easily isolated and contained.
- Attempts to introduce “self-policed” (i.e., unenforced) modularity constraints into a design typically fail, despite best intentions.

To help manage the large team sizes and complex requirements of modern enterprise applications, it is clear that we need a more principled way to “divide and conquer.” Something that lets us encapsulate well-defined parts of the system as modules with hidden internals and carefully managed externals. Something that enables those modules to be packaged and deployed individually without forcing us to revise the whole universe. Something that provides a principled mechanism for bringing those modules together in a running system, and that can cope with the changes introduced by continuous evolution.

Facing these requirements back in 2005, it was an easy decision at SpringSource (then Interface21) to turn to OSGi, the “dynamic module system for Java,” as the foundation technology for modular enterprise applications. Even then, the OSGi Service Platform was already mature and proven in industrial settings, as well as being lightweight through its heritage in embedded systems.

The modularity layer of OSGi provides a mechanism for dividing a system into independent modules, known as bundles, that are independently packaged and deployed and have independent lifecycles. This solved a part of the problem for us—helping to keep the implementation types of a module private, and exposing only types that form part of the public interface of a module. We wanted enterprise developers to continue developing their applications using Spring, of course, and through the Spring Dynamic Modules' open-source project created a simple model whereby each module had its own set of components (Spring beans). Some of those components are private to the module, but some should be made public so that components in other modules can use them. The OSGi service layer provides an answer to this problem, promoting an in-memory service-oriented design. Components from a module can be published in the OSGi service registry, and from there other modules can find and bind to those services. OSGi also provides the necessary primitives to track services that may come and go over time as modules are installed, uninstalled, and upgraded.

The next stage in our journey with OSGi was the introduction of the SpringSource dm Server: an enterprise application server that is not only built on top of OSGi, but critically also supports the deployment of applications developed as a set of OSGi bundles. Spring Dynamic Modules works with any compliant OSGi Service Platform implementation, but for the dm Server we had to choose an OSGi Service Platform as the base on which to build. We chose to build on Equinox, the Eclipse implementation of the OSGi Service Platform, and also the reference implementation for the core OSGi specification. The open-source nature of Equinox fit well with our own open-source philosophy and has been invaluable in enabling us to work closely with the developers of Equinox and submit patches and change requests over time. The widespread adoption of Equinox (as the underpinnings of Eclipse, to name but one example) gave us confidence that it would be battle-hardened and ready for enterprise usage.

I am seeing a strong and growing serious interest in OSGi among companies large and small. Building on OSGi will provide a firm foundation for dividing your application into modules, which in turn will help you structure the team(s) working on it more effectively. "Organization follows architecture" in the sense that your ability to divide a complex application into independent pieces also facilitates the structuring of team responsibilities along the same lines. In other scenarios, your teams may be fixed, and you need an architecture that enables those teams to work together most effectively. Again, a principled basis for dividing a system into modules can facilitate that. With OSGi as a basis, your unit of packaging and deployment can become a single module, removing bottlenecks in the process and helping to minimize the impact of change. OSGi is also incredibly well suited to product-line engineering, and to situations where you need to provide an extension or plug-in mechanism to enable third parties to extend your software.

The future for OSGi looks bright. Version 4.2 of the specification has just been released, and the OSGi Core Platform and Enterprise Expert Groups are very active. A glance at the membership of the OSGi Alliance and the composition of the expert groups tells you just how seriously enterprise vendors are taking it. I am confident that the investment of your time in reading and studying this book will be well rewarded. It is my belief that OSGi is here to stay. A firm grasp of the strengths—and the weaknesses—of the OSGi Service Platform will prove invaluable to you on your journey toward creating agile, modular software.

—Adrian Colyer
CTO, SpringSource
October 2009

This page intentionally left blank

Preface

OSGi is a hot topic these days; all the major Java application server vendors have adopted OSGi as their base runtime, Eclipse has been using OSGi as the basis of its modularity story and runtime for at least the past five years, and countless others have been using it in embedded and “under the covers” scenarios. All with good reason.

The success of Eclipse as a tooling platform is a direct result of the strong modularity enshrined in OSGi. This isolates developers from change, empowers teams to be more agile, allows organizations to change the way that they develop software, and lubricates the formation and running of ecosystems. These same benefits can be realized in any software domain.

The main OSGi specification is remarkably concise—just 27 Java types. It is well designed, and specified to be implemented and used in real life. Adoption of OSGi is not without challenges, however. Make no mistake: Implementing highly modular and dynamic systems is hard. There is, as they say, no free lunch. Some have criticized OSGi as being complicated or obtuse. In most cases it is the problem that is complex—the desire to be modular or dynamic surfaces the issues but is not the cause. Modularizing existing monolithic systems is particularly challenging.

This book is designed to both highlight such topics and provide knowledge, guidance, and best practices to mitigate them. We talk heavily of modularity, components, and dynamism and show you techniques for enhancing your system’s flexibility and agility.

Despite using OSGi for many years, participating in writing the OSGi specifications, and implementing Equinox (the OSGi framework specification reference implementation), during the writing of this book we learned an incredible amount about OSGi, Equinox, and highly modular dynamic systems. We trust that in reading it you will, too.

About This Book

This book guides up-and-coming and established OSGi developers through all stages of developing and delivering an example OSGi-based telematics and fleet management system called *Toast*.

We develop Toast from a blank workspace into a full-featured client and server system. The domain is familiar to most everyone who has driven a car or shipped a package. Telematics is, loosely speaking, all the car electronics—radio, navigation, climate control, and so on. Fleet management is all about tracking and coordinating packages and vehicles as they move from one place to another.

The set of problems and opportunities raised allows us to plausibly touch a wide range of issues from modularity and component collaboration to server-side programming and packaging and delivery of highly modular systems. We create stand-alone client applications, embedded and stand-alone server configurations, and dynamic enhancements to both. This book enables you to do the same in your domain.

Roughly speaking, the book is split into two sections. The first half, Parts I and II, sets the scene for OSGi and Equinox and presents a tutorial-style guide to building Toast. The tutorial incrementally builds Toast into a functioning fleet management system with a number of advanced capabilities. The tutorial is written somewhat informally to evoke the feeling that we are there with you, working through the examples and problems. We share some of the pitfalls and mishaps that we experienced while developing the application and writing the tutorial.

The second half of the book looks at what it takes to "make it real." It's one thing to write a prototype and quite another to ship a product. Rather than leaving you hanging at the prototype stage, Part III is composed of chapters that dive into the details required to finish the job—namely, the refining and refactoring of the first prototype, customizing the user interface, and building and delivering products to your customers. This part is written as a reference, but it still includes a liberal sprinkling of step-by-step examples and code samples. The goal is both to dive deep and cover most of the major stumbling blocks reported in the community and seen in our own development of professional products.

A final part, Part IV, is pure reference. It covers the essential aspects of OSGi and Equinox and touches on various capabilities not covered earlier in the book. We also talk about best practices and advanced topics such as integrating third-party code libraries and being dynamic.

OSGi, despite being relatively small, is very comprehensive. As such, a single book could never cover all possible topics. We have focused on the functions and services that we use in the systems we develop day to day under the assumption that they will be useful to you as well.

OSGi, Equinox, and EclipseRT

The OSGi community is quite vibrant. There are at least three active open-source framework implementation communities and a wide array of adopters and extenders. The vast majority of this book covers generic OSGi topics applicable to any OSGi system or implementation. Throughout the book we consistently use Equinox, the OSGi framework specification reference implementation, as the base for our examples and discussions. From time to time we cover features and facilities available only in Equinox. In general, these capabilities have been added to Equinox to address real-world problems—things that you will encounter. As such, it is prudent that we discuss them here.

Throughout the book we also cover the Eclipse Plug-in Development Environment (PDE) tooling for writing and building OSGi bundles. PDE is comprehensive, robust, and sophisticated tooling that has been used in the OSGi context for many years. If you are not using PDE to create your OSGi-based systems, perhaps you should take this opportunity to find out what you are missing.

Finally, Eclipse is a powerhouse in the tooling domain. Increasingly it is being used in pure runtime, server-side, and embedded environments. This movement has come to be known as *EclipseRT*. EclipseRT encompasses a number of technologies developed at Eclipse that are aimed at or useful in typical runtime contexts. The Toast application developed here has been donated to the Eclipse Examples project and is evolving as a showcase for EclipseRT technologies. We encourage you to check out http://wiki.eclipse.org/Toast to see what people have done to and with Toast.

Audience

This book is targeted at several groups of Java developers. Some Java programming experience is assumed, and no attempt is made to introduce Java concepts or syntax.

For developers new to OSGi and Equinox, there is information about the origins of the technology, how to get started with the Eclipse OSGi bundle tooling, and how to create your first OSGi-based system. Prior experience with Eclipse as a development tool is helpful but not necessary.

For developers experienced with writing OSGi bundles and systems, the book formalizes a wide range of techniques and practices that are useful in creating highly modular systems using OSGi—from service collaboration approaches to server-side integration and system building as part of a release engineering process, deployment, and installation.

For experienced OSGi developers, this book includes details of special features available in Equinox and comprehensive coverage of useful facilities such as

Declarative Services, buddy class loading, Google Earth integration, and the Eclipse bundle tooling that make designing, coding, and packaging OSGi-based systems easier than ever before.

Sample Code

Reading this book can be a very hands-on experience. There are ample opportunities for following along and doing the steps yourself as well as writing your own code. The companion download for the book includes code samples for each chapter. Instructions for getting and managing these samples are given in Chapter 3, "Tutorial Introduction," and as needed in the text. In general, all required materials are available online at either http://eclipse.org or http://equinoxosgi.org. As mentioned previously, a snapshot of Toast also lives and evolves as an open-source project at Eclipse. See http://wiki.eclipse.org/Toast.

Conventions

The following formatting conventions are used throughout the book:

Bold—Used for UI elements such as menu paths (e.g., **File > New > Project**) and wizard and editor elements

Italics—Used for emphasis and to highlight terminology

`Lucida`—Used for Java code, property names, file paths, bundle IDs, and the like that are embedded in the text

`Lucida Bold`—Used to highlight important lines in code samples

Notes and sidebars are used often to highlight information that readers may find interesting or helpful for using or understanding the function being described in the main text. We tried to achieve an effect similar to that of an informal pair-programming experience where you sit down with somebody and get impromptu tips and tricks here and there.

Feedback

The official web site for this book is http://equinoxosgi.org. Additional information and errata are available at informit.com/title/0321585712. You can report problems or errors found in the book or code samples to the authors at book@equinoxosgi.org. Suggestions for improvements and feedback are also very welcome.

CHAPTER 2

OSGi Concepts

The OSGi Alliance[1] (http://osgi.org) is an independent consortium with the mission "to create a market for universal middleware." This manifests itself as a set of specifications, reference implementations, and test suites around a dynamic module system for Java. The module system forms the basis for a "service platform" that in turn supports the creation and execution of loosely coupled, dynamic modular systems. Originating in the embedded space, OSGi retains its minimalist approach by producing a core specification of just 27 Java types. This ethic of simplicity and consistency is pervasive in the OSGi specifications.

In this chapter we explore the basic concepts around OSGi and look at how they fit together. You will learn about

- The OSGi framework, its key parts and operation
- Bundles, their structure, and their lifecycle
- Services, extensions, and component collaboration

2.1 A Community of Bundles

An OSGi system is a community of components known as *bundles*. Bundles executing within an OSGi service platform are independent of each other, yet they collaborate in well-defined ways. Bundles are fully self-describing, declaring their public API, defining their runtime dependencies on other bundles, and hiding their internal implementation.

1. The OSGi Alliance was founded as the Open Services Gateway initiative. They have since rebranded as the "OSGi Alliance."

Bundle writers, *producers*, create bundles and make them available for others to use. System integrators or application writers, *consumers*, use these bundles and write still more bundles using the available API. This continues until there is enough functionality available to solve a given problem. Bundles are then composed and configured to create the desired system.

As shown in Figure 2-1, an OSGi application has no top and no bottom—it is simply a collection of bundles. There is also no *main* program; some bundles contribute code libraries; others start threads, communicate over the network, access databases, or collaborate with still others to gain access to hardware devices and system resources. While there are often dependencies between bundles, in many cases bundles are peers in a collaborative system.

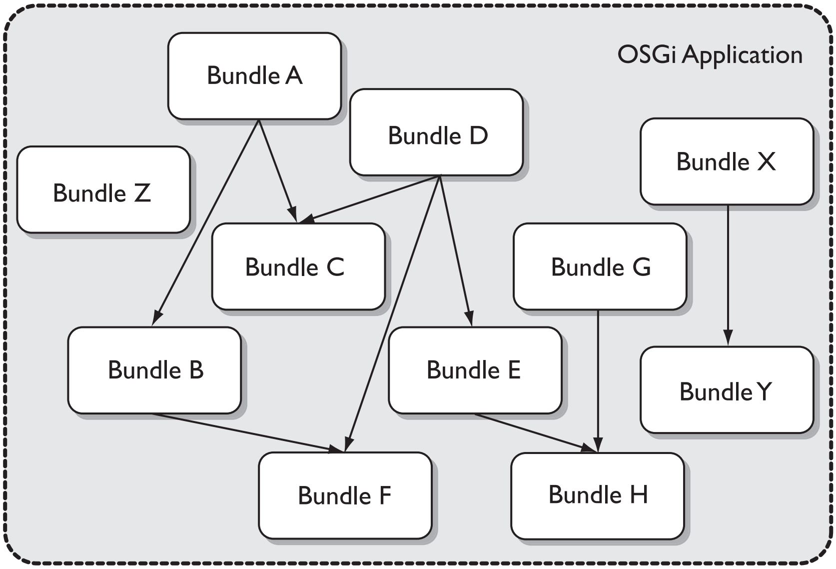

Figure 2–1 An OSGi application as a collection of interdependent bundles

OSGi-based systems are dynamic in that the bundles in the community can change over the lifetime of the application. A bundle can be installed, uninstalled, and updated at any time. To facilitate this, bundles must be implemented to gracefully handle being uninstalled, as well as to respond to the addition, removal, and possible replacement of collaborating bundles.

These characteristics lead to a fundamentally simple but powerful module system upon which other systems can be built. Indeed, modularity and OSGi bundles are among the secrets to the success of Eclipse as a platform and as an ecosystem. In any suitably large system it is increasingly unlikely that all components will be written by the same producer. In fact, in an OSGi system such as an Eclipse application, it is common for bundles to come from a variety of producers, such as open-source projects, corporations, and individuals. The strong modularity promoted and supported by OSGi dramatically increases the opportunity for code reuse and accelerates the delivery of applications.

2.2 Why OSGi?

If OSGi is so small and simple, what makes it so special? To understand more, let's first look at a traditional Java application. A Java system is composed of *types—classes* and *interfaces*. Each type has a set of members—*methods* and *fields*—and is organized into *packages*. The set of Java packages defines a global type namespace, and the Java language defines the visibility rules used to manage the interactions between types and members. As shown in Figure 2-2, types and packages are typically built and shipped as Java Archives (JARs). JARs are then collected together on one *classpath* that is linearly searched by the Java virtual machine (JVM) to discover and load classes.

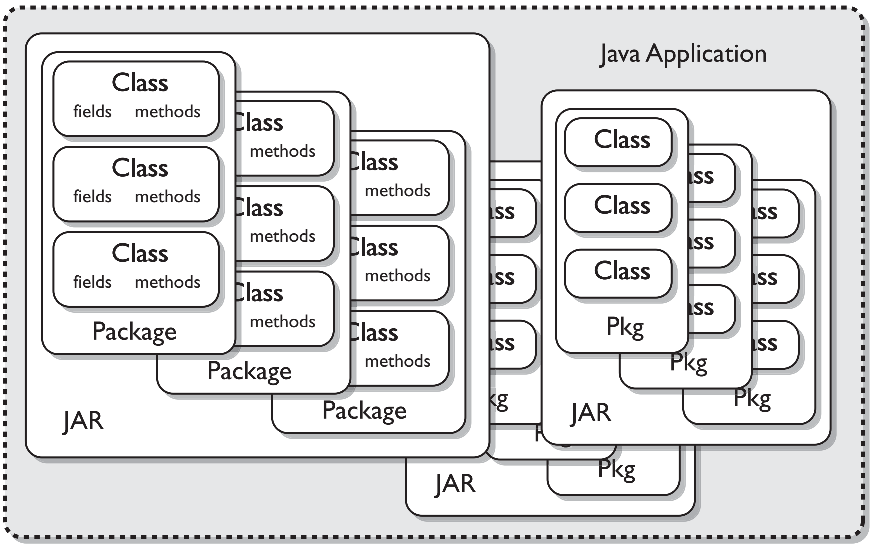

Figure 2–2 A Java application

So far it sounds pretty good—packages feel modular and there are visibility rules to enable information hiding. At the low level the story is reasonable, but things break down at the system and collaboration level. There are two main issues: Packages are too granular to be modules, and JARs are simply a delivery mechanism with no runtime semantics.

The Java type and member visibility rules allow developers to hide elements within a package, so it feels natural to say that packages == modules. In practice this forces either packages to be too large or modules to be too numerous. Experience tells us that modules are often themselves composed of code from various sources and that it is a best practice to use fine-grained package naming to enable later refactoring. Mixing packages with modularity is counter to both experiences.

The JAR concept is very useful. It could be argued that the JAR as a delivery vehicle was one of the drivers of the original success of Java. Producers create JARs of useful function, and consumers use these JARs to build systems. Unfortunately, JARs really are just a delivery vehicle and have minimal impact on the running of the system. Delivered JARs simply go on a flat classpath with no control over the accessibility of their contents.

Combined, these characteristics mean that Java has no support for defining or enforcing dependencies. Without dependencies, modularity is not possible. You end up with systems where JARs fight for position on the classpath, JAR content has more to do with who wrote the code rather than its functionality, APIs are unclear, and the relationships between JARs are at best managed by weak conventions. As shown in Figure 2-3, the net result is monolithic applications composed of tightly coupled JARs with multidirectional and even cyclical dependencies. Collaboration and sharing between teams is impacted and application evolution hindered.

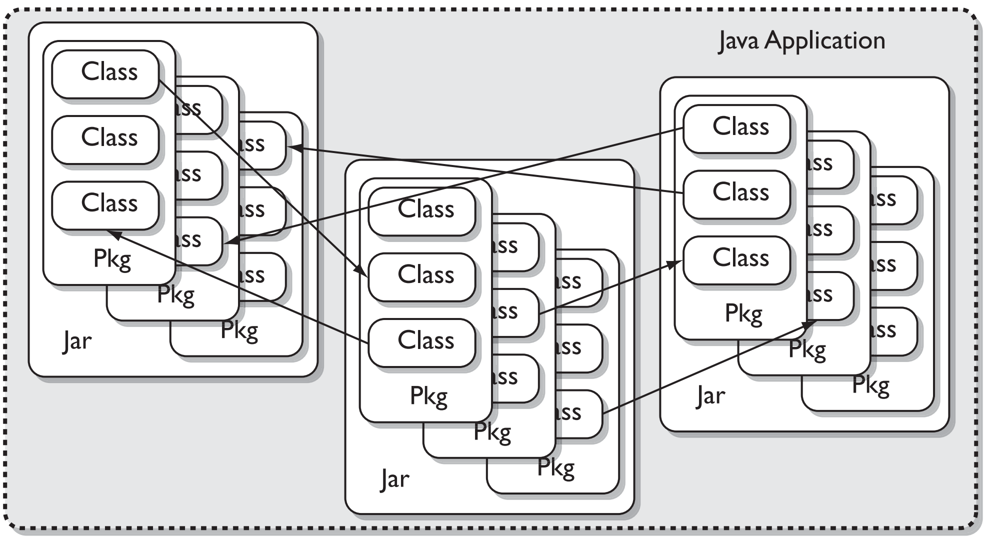

Figure 2–3 A monolithic application

OK, so what makes OSGi better? It's still Java, right? True. OSGi builds on the basic Java mechanisms just outlined but adds a few key elements. In particular, rather than talking about JARs, OSGi talks about bundles. A bundle is typically implemented as a JAR, but with added identity and dependency information; that is, bundles are self-describing JARs. This simple idea has two effects: Producers and consumers have an opportunity to express their side of the contract, and the runtime has the information it needs to enforce these expectations.

By default the packages in a bundle are hidden from other bundles. Packages containing API must, by definition, be available to other bundles and so must be explicitly *exported*. Bundles including code that uses this API must then have a matching *import*. This visibility management is similar in concept to Java's package visibility but at a much more manageable and flexible level.

The OSGi runtime enforces these visibility constraints, thus forming the basis of a strong but loosely coupled module system. Importing a package simply states that the consuming bundle depends on the specified package, regardless of the bundles that provide it. At runtime a bundle's package dependencies are resolved and bundles are wired together, based on rules that include package names, versions, and matching attributes. This approach effectively eliminates the *classpath hell* problem while simultaneously providing significant class loading performance improvements and decreased coupling.

No code is an island. All this loose coupling comes at a price. In a traditional Java system, if you wanted to use some functionality, you would simply reference the required types. The tightly coupled approach is simple but limiting. In a scenario that demands more flexibility this is not possible. The Java community is littered with ad hoc and partial solutions to this: Context class loaders, `Class.forName`, "services" lookup, log appenders—all are examples of mechanisms put in place to enable collaboration between loosely coupled elements.

While the importing and exporting of packages express static contracts, *services* are used to facilitate dynamic collaboration. A service is simply an object that implements a contract, a type, and is registered with the OSGi service registry. Bundles looking to use a service need only import the package defining the contract and discover the service implementation in the service registry. Note that the consuming bundle does not know the implementation type or producing bundle since the service interface and implementation may come from different bundles—the system is collaborative yet remains loosely coupled.

Services are dynamic in nature: A bundle dynamically registers and unregisters services that it provides, and it dynamically acquires and releases the services that it consumes. Some bundles are service providers, some are service consumers, and others are both providers and consumers.

In many ways OSGi can be thought of as an extension to the Java programming language that allows package visibility and package dependency constraints to be specified at development time and enforced at runtime. Through these constraints it is easier to build applications that are composed of loosely coupled and highly cohesive components.

2.3 The Anatomy of a Bundle

A bundle is a self-describing collection of files, as shown in Figure 2-4.

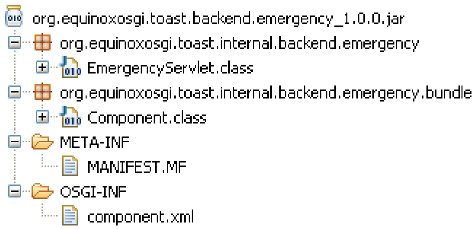

Figure 2–4 Bundle anatomy

The specification of a bundle's contents and requirements is given in its manifest file, `META-INF/MANIFEST.MF`. The manifest follows the standard JAR manifest syntax but adds a number of OSGi-specific headers. The manifest for the `org.equinoxosgi.toast.backend.emergency` bundle from the figure looks like this:

org.equinoxosgi.toast.backend.emergency/MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.equinoxosgi.toast.backend.emergency
Bundle-Version: 1.0.0
Import-Package: javax.servlet;version="2.4.0",
 javax.servlet.http;version="2.4.0",
 org.equinoxosgi.toast.core;version="1.0.0",
 org.equinoxosgi.toast.core.emergency;version="1.0.0",
 org.osgi.service.component;version="1.0.0",
 org.osgi.service.http;version="1.2.0"
Export-Package: org.equinoxosgi.toast.backend.emergency.internal;
 version="1.0.0";x-internal:=true,
 org.equinoxosgi.toast.backend.emergency.internal.bundle;
 version="1.0.0";x-internal:=true
Bundle-RequiredExecutionEnvironment: J2SE-1.4
Bundle-Copyright: Copyright (c) 2009 equinoxosgi.org
Bundle-Name: Toast Back End Emergency
Bundle-Vendor: equinoxosgi.org
```

All bundle manifests must have the headers `Bundle-SymbolicName` and `Bundle-Version`. The combination of these headers uniquely identifies the bundle to

OSGi frameworks, developers, and provisioning systems. A bundle also expresses its modularity through headers such as `Export-Package`, `Import-Package`, and `Require-Bundle`. Additional headers such as `Bundle-Copyright`, `Bundle-Name`, and `Bundle-Vendor` are purely documentation. Throughout the book we'll introduce additional headers as they arise in the tutorial.

A bundle can contain Java types, native libraries, or other, nonexecutable files. The content and structure of a bundle depend entirely on what it is delivering and how it is being used. Most bundles deliver Java code to be executed by a Java runtime. These are structured as JARs with the Java code in a package-related folder structure (e.g., `org/equinoxosgi/toast/core/Delay.class`).

Bundles that deliver non-Java content (e.g., source, documentation, or static web content) are structured to suit the mechanism consuming their content. For example, native executables and files being accessed from other programs must reside directly on disk rather than nested inside JAR files. OSGi framework implementations such as Equinox facilitate this by supporting *folder-based* bundles. Folder-based bundles are essentially just JAR bundles that have been extracted.

2.4 Modularity

An OSGi bundle provides a clear definition of its modularity—this includes its identity, its requirements, and its capabilities. The `Bundle-SymbolicName` and `Bundle-Version` manifest headers take care of defining identity. A bundle can have a number of different capabilities and requirements. The most common pattern is to express these dependencies in terms of Java packages. Bundle developers can also specify dependencies on whole bundles.

2.4.1 Exporting a Package

To give access to Java types in a bundle, the bundle must export the package containing the types; that is, OSGi's unit of Java dependency is the Java package. Bundles can export any number of packages. By exporting a package, the bundle is saying that it is able and willing to supply that package to other bundles. Exported packages form the public API of the bundle. Packages that are not exported are considered to be private implementation details of the bundle and are not accessible to others. This is a powerful concept and one of the reasons that OSGi's component model is so appealing.

A bundle that uses the `Export-Package` header to export several packages is shown in the following manifest snippet. Notice that the packages are specified in a comma-separated list and that a version number can be specified for each package. Each package is versioned independently.

```
org.equinoxosgi.toast.core/MANIFEST.MF
Bundle-SymbolicName: org.equinoxosgi.toast.core
Bundle-Version: 1.0.0
Export-Package: org.equinoxosgi.toast.core;version=1.2.3,
 org.equinoxosgi.toast.core.services;version=8.4.2
```

2.4.2 Importing a Package

Exporting a package makes it visible to other bundles, but these other bundles must declare their dependency on the package. This is done using the `Import-Package` header.

The following manifest snippet shows a bundle that imports several packages. As with exports, the set of imported packages is given as a comma-separated list. Notice here that the import for each package can be qualified with a *version range*. The range specifies an upper and lower bound on exported versions that will satisfy the requirements of this bundle. Versions, version ranges, and dependency management are discussed throughout the book as they form a key part of developing, maintaining, and deploying modular systems.

```
org.equinoxosgi.toast.core/MANIFEST.MF
Bundle-SymbolicName: org.equinoxosgi.toast.core
Bundle-Version: 1.0.0
Import-Package: org.osgi.framework;version="[1.3,2.0.0)"
 org.osgi.service.cm;version="[1.2.0,2.0.0)"
```

2.4.3 Requiring a Bundle

It is also possible to specify a dependency on an entire bundle using a `Require-Bundle` header, as shown in the following manifest fragment:

```
org.equinoxosgi.toast.dev.airbag.fake/MANIFEST.MF
Bundle-Name: Toast Fake Airbag
Bundle-SymbolicName: org.equinoxosgi.toast.dev.airbag.fake
Bundle-Version: 1.0.0
Import-Package: org.eclipse.core.runtime.jobs,
 org.equinoxosgi.toast.core;version="[1.0.0,2.0.0)",
 org.equinoxosgi.toast.dev.airbag;version="[1.0.0,2.0.0)"
Require-Bundle: org.eclipse.equinox.common; bundle-version="3.5.0"
```

With this approach, a bundle is wired directly to the prerequisite bundle and all packages it exports. This is convenient but reduces the ability to deploy bundles in different scenarios. For example, if the required bundle is not, or cannot be, deployed, the bundle will not resolve, whereas the actual package needed may be available in a different bundle that can be deployed.

Requiring bundles can be useful when refactoring existing systems or where one bundle acts as a façade for a set of other bundles. Requiring a bundle also allows for the specification of dependencies between modules that do not deliver Java code and so do not export or import packages.

THE HISTORY OF `Require-Bundle`

Historically, Eclipse projects use `Require-Bundle` because that is what the original Eclipse runtime supported. Now that Eclipse is OSGi-based, many of these bundles would be better off using `Import-Package`. This is happening over time as the need for this additional flexibility is recognized.

2.4.4 Enforcing Modularity

Given these capability and requirements statements, the OSGi framework *resolves* the dependencies and wires bundles together at runtime. Modularity in an OSGi system is enforced through a combination of wires and standard Java language visibility rules. To manage this, the framework gives each bundle its own class loader. This keeps separate the classes from the different bundles. When a bundle is uninstalled or updated, its class loader, and all classes loaded by it, are discarded. Having separate class loaders allows the system to have multiple versions of the same class loaded simultaneously. It also enforces the standard Java type visibility rules, such as package visible and public, protected and private, in a bundle world.

2.5 Modular Design Concepts

Given these constructs, how do we talk about OSGi-based applications? One way is to look at the abstraction hierarchy:

Application > Bundle > Package > Type > Method

This shows that a bundle is an abstraction that is bigger than a package but smaller than an application. In other words, an application is composed of bundles; bundles are composed of packages; packages are composed of types; and types are composed of methods. So, just as a type is composed of methods that implement its behavior, an application is composed of bundles that implement its behavior. The task of decomposing an application into bundles is similar to that of decomposing an application into types and methods.

Another way to talk about OSGi-based systems is to talk about decomposition. Key to high-quality design at all levels is the decomposition used. We talk about and measure decomposition along three axes: granularity, coupling, and cohesion. Here we relate these terms to the OSGi environment:

Granularity—Granularity is the measure of how much code and other content is in a bundle. Coarse-grained bundles are easy to manage but are inflexible and bloat the system. Fine-grained bundles give ultimate control but require more attention. Choosing the right granularity for your bundles is a balance of these tensions. Big is not necessarily bad, nor small, good. In some ways granularity is the overarching consequence of coupling and cohesion.

Coupling—Coupling is an outward view of the number of relationships between a bundle and the rest of the system. A bundle that is highly coupled requires many other bundles and generally makes many assumptions about its surrounding context. On the other hand, loosely coupled bundles operate independently and offer you the flexibility to compose your application to precisely meet your changing requirements without dragging in unnecessary dependencies.

Cohesion—Cohesion is an inward view of the relevance of the elements of a bundle to one another. In a highly cohesive bundle, all parts of the bundle are directly related to, and focused on, addressing a defined, narrowly focused topic. Low-cohesion bundles are ill-defined dumping grounds of random content. Highly cohesive bundles are easier to test and reuse, and they enable you to deliver just the function you need and nothing more. A common pitfall is to consider a bundle to be either an entire subsystem or an entire layer in the application's architecture, for example, the domain model or the user interface. A highly cohesive bundle often provides a solution to part, but not all, of a problem.

These ideas are not unique to OSGi—they are tenets of good design practices and fundamental to object-oriented and agile approaches. In the case of OSGi, however, the system is designed to expose and enforce key aspects of coupling, cohesion, and granularity, making the benefits directly tangible. OSGi encourages you to decompose your application into right-grained bundles that are loosely coupled and highly cohesive.

2.6 Lifecycle

OSGi is fundamentally a dynamic technology. Bundles can be installed, started, stopped, updated, and uninstalled in a running system. To support this, bundles must have a clear lifecycle, and developers need ways of listening to and hooking into the various lifecycle states of a bundle (see Fig. 2-5).

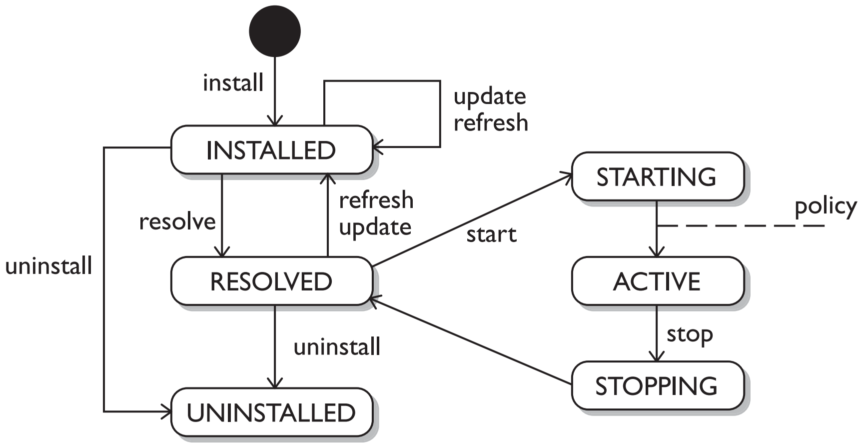

Figure 2–5 Bundle lifecycle

Every bundle starts it runtime life in the *installed* state. From there it becomes *resolved* if all of its dependencies are met. Once a bundle is resolved, its classes can be loaded and run. If a bundle is subsequently started and transitions to the *active* state, it can participate in its lifecycle by having an *activator*. Using the activator, the bundle can initialize itself, acquire required resources, and hook in with the rest of the system. At some point—for example, on system shutdown—active bundles get stopped. Bundles with activators have a chance to free any resources they may have allocated. Bundles transition back to the resolved state when they are stopped. From there they may be restarted or *uninstalled*, at which time they are no longer available for use in the system.

All of this state changing surfaces as a continuous flow of events. Bundles support dynamic behavior by listening to these events and responding to the changes. For example, when a new bundle is installed, other bundles may be interested in its contributions.

The OSGi framework dispatches events when the state of the bundles, the services, or the framework itself changes.

Service events—Fired when a service is registered, modified, or unregistered

Bundle events—Fired when the state of the framework's bundles changes, for example, when a bundle is installed, resolved, starting, started, stopping, stopped, unresolved, updated, uninstalled, or lazily activated

Framework events—Fired when the framework is started; an error, warning, or info event has occurred; the packages contributing to the framework have been refreshed; or the framework's start level has changed

2.7 Collaboration

OSGi-based systems are composed of self-describing bundles as outlined previously. Bundles can collaborate by directly referencing types in other bundles. That is a simple pattern familiar to all Java programmers, but such systems are tightly coupled and miss out on the real power of modularity—loose coupling and dynamic behavior.

To loosen the coupling between modules, there must be a collaboration mechanism, a third party, that acts as an intermediary and keeps the collaborators at arm's length. The typical OSGi mechanism for this is the *service registry.* Equinox, of course, supports the service registry but also adds the *Extension Registry.* These complementary approaches are outlined in the following sections and discussed in more detail throughout the book.

2.7.1 Services

The OSGi service registry acts like a global bulletin board of functions coordinating three parties: bundles that define service interfaces, bundles that implement and register service objects, and bundles that discover and use services. The service registry makes these collaborations anonymous—the bundle providing a service does not know who is consuming it, and a bundle consuming a service does not know what provided it. For example, Figure 2-6 shows Bundle C that declares an interface used by Bundle B to register a service. Bundle A discovers and uses the service while remaining unaware of, and therefore decoupled from, Bundle B. Bundle A depends only on Bundle C.

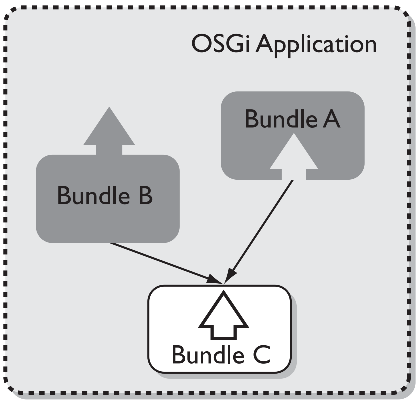

Figure 2–6 Service-based collaboration

Services are defined using a Java type, typically a Java interface. The type must be public and reside in a package that is exported. Other bundles—and perhaps even the same bundle—then implement the service interface, instantiate it, and register the instance with the service registry under the name of the service interface. The classes that implement the service, being implementation details, generally are not contained in packages that are exported.

Finally, a third set of bundles consumes the available services by importing the package containing the service interface and looking up the service in the service registry by the interface name. Having obtained a matching service object, a consuming bundle can use the service until done with it or the service is unregistered. Note that multiple bundles can consume the same service object concurrently, and multiple service objects may be provided by one or more bundles.

The dynamic aspect of service behavior is often managed in conjunction with the lifecycle of the bundles involved. For example, when a bundle is started, it discovers its required services and instantiates and registers the services it provides. Similarly, when a bundle is stopped, its bundle activator unregisters contributed services and releases any services being consumed.

2.7.2 Extensions and Extension Points

The Equinox Extension Registry is a complementary mechanism for supporting inter-bundle collaboration. Under this model, bundles can open themselves for extension or configuration by declaring an *extension point*. Such a bundle is essentially saying, "If you give me the following information, I will do . . . ." Other bundles then *contribute* the required information to the extension point in the form of *extensions*.

In this book we use the example of an extensible web portal that allows actions to be contributed and discovered via the Extension Registry. In this approach the portal bundle declares an `actions` extension point and a contract that says,

> "Bundles can contribute `actions` extensions that define portal actions with a path, a label, and a class that implements the interface `IPortalAction`. The portal will present the given label to the user organized according to the given path and such that when the user clicks on the label, a particular URL will be accessed. As a result of the URL request, the portal will instantiate the given action class, cast it to `IPortalAction`, and call its `execute` method."

Figure 2-7 shows this relationship graphically.

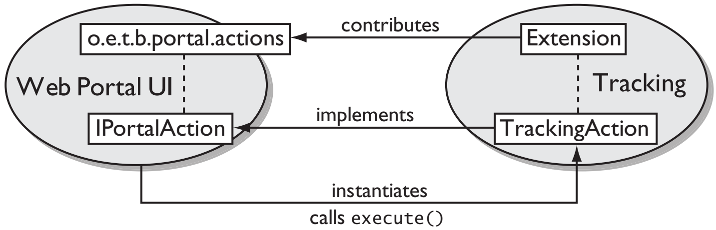

Figure 2–7 Extension contribution and use

Extension-to-extension-point relationships are defined using XML in a file called `plugin.xml`. Each participating bundle has one of these files. As bundles are resolved in the system, their extensions and extension points are loaded into the Extension Registry and made available to other bundles. A full set of Extension Registry events is broadcast to registered listeners along the way. Extension and extension points can also be managed programmatically.

2.8 The OSGi Framework

The modularity mechanisms described previously are largely implemented by the *OSGi Framework*. As such, an OSGi application is a collection of one or more bundles executing in an OSGi framework. The framework takes care of all the dependency resolution, class loading, service registrations, and event management.

TERMINOLOGY

The phrases "the OSGi framework," "the OSGi runtime," and "the service platform" are often used interchangeably and are typically abbreviated to just "the framework," "the runtime," or "the platform."

The framework is reified in a running system as the *System Bundle*. Representing the OSGi framework as a bundle allows us to view the entire platform consistently as a collection of collaborating bundles. While the System Bundle is clearly special, it contains a manifest, exports packages, provides and consumes services, and broadcasts and listens to events like any other bundle.

The System Bundle differs from other bundles in that its lifecycle cannot be managed. It is started automatically when the framework is started and continues in the active state until the framework is stopped. Stopping the System Bundle causes the framework to shut down. Similarly, the System Bundle cannot be uninstalled while running, since doing so would cause the framework to terminate.

The other bundles in an OSGi system are installed into the framework and started as needed. The set of installed bundles in a framework is persisted from run to run—when the framework is shut down and relaunched, the same set of bundles is present and started in the new framework. As such, bundles need to be installed and started only once.

Interestingly, the framework specification does not say how the framework itself is started or how the initial set of bundles is installed. In general it is envisioned that there is an external management agent installing and uninstalling, and starting and stopping, bundles. This may be a central service provider, systems integrator, provisioning agent, or the end user. This approach is powerful, as it makes the framework equally applicable in a wide range of scenarios.

The framework also supplies some rudimentary data management facilities. Each bundle is given its own data area to use as required. The data written in this area is persisted for as long as the bundle is installed in the framework.

2.9 Security

The OSGi specifications include security as a fundamental element. In addition to the standard Java 2 permissions, OSGi-specific permissions are defined throughout the framework and supplemental services. For example, with the system running in secure mode, bundles require distinct permissions to register and look up services and access properties.

The permissions in a system are managed by special-purpose services such as the Conditional Permissions Admin service. This service can be used to manage permissions on a per-bundle basis by, for example, giving bundles certain permissions if they are digitally signed by particular parties. In addition, the User Admin service facilitates the management of user-level or application permissions based on the current user's identity and role.

The real value of the OSGi permission model is that it is used throughout the entire framework and service set.

2.10 OSGi Framework Implementations

At the time of this writing there have been four major revisions of the OSGi specifications. Over the ten-year history of OSGi there have been many implementations. The current R4.x specifications are implemented by several open-source and commercial entities:

Equinox—Perhaps the most widely used open-source OSGi implementation, Equinox is the base runtime for all Eclipse tooling, rich client, server-side, and embedded projects. It is also the reference implementation for the core framework specification, several service specifications, and JSR 291. It is available under the Eclipse Public License from http://eclipse.org/equinox.

Felix—Originally the Oscar project, the Felix open-source project at Apache supplies a framework implementation as well as several service implementations. It is available under the Apache License v2 from http://felix.apache.org.

Knopflerfish—The Knopflerfish open-source project supplies an R4.x framework implementation as well as several service implementations. It is available under a BSD-style license from http://knopflerfish.org.

mBedded Server—This commercial R4.x implementation from ProSyst is used in a number of embedded application areas. ProSyst offers several additional service implementations. It is available under commercial terms from http://prosyst.com.

Concierge—Concierge is an open-source highly optimized and minimized R3.0 specification implementation that is suitable for use in small embedded scenarios. It is available under a BSD-style license from http://concierge.sourceforge.net.

Despite the many features and functions included in the base framework, implementations are very small and run on minimal JVM implementations. Concierge weighs in at a mere 80K disk footprint. The base specification-compliant parts of R4.x implementations tend to have a 300–600K disk footprint. Implementations such as Equinox include considerable additional functionality such as enhanced flexibility, advanced signature management, and high scalability in their base JARs but still stay under 1M on disk.

2.11 Summary

The OSGi framework specification is a good example of power through simplicity and consistency. The technology is based on a small number of simple but general notions such as modularity and services. OSGi's origins in the embedded world drive a minimalist approach that is present throughout the specification.

It is this simplicity that allows the framework to be extended and applied in a wide range of situations. This is the key value in OSGi—its universality. The Eclipse adoption of OSGi and its subsequent spread to use in the rich client and now server world bring real power to Java developers and system integrators.

Index

C

D

E

F

G

H

I

J

M

N

O

P

Q

R

S

T

X

Z