



OSGi and Equinox

Creating Highly Modular Java™ Systems

Jeff McAffer • Paul VanderLei • Simon Archer



Series Editors Jeff McAffer • Erich Gamma • John Weigand

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informati.com/aw

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)

I. 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

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

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 Spring-Source (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 Spring-Source 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 Alliance¹ (<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.

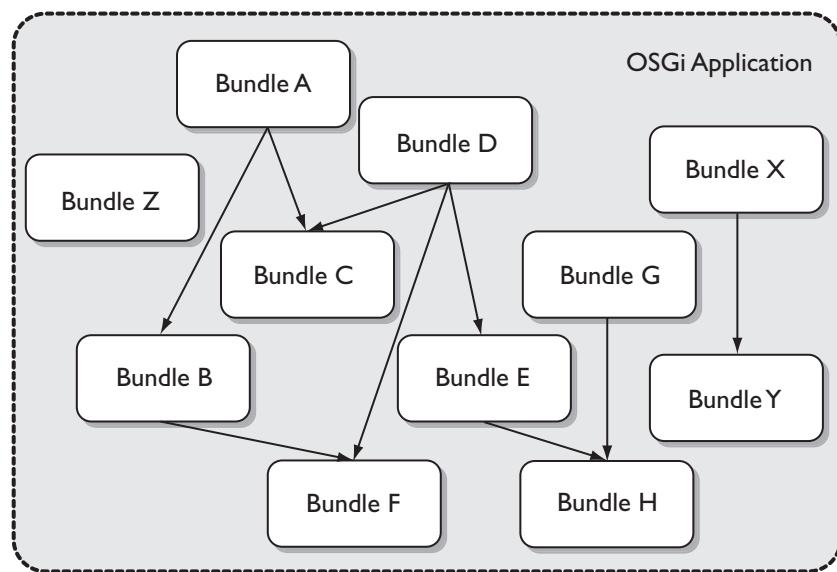


Figure 2-1 An OSGi application as a collection of interdependent bundles

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.

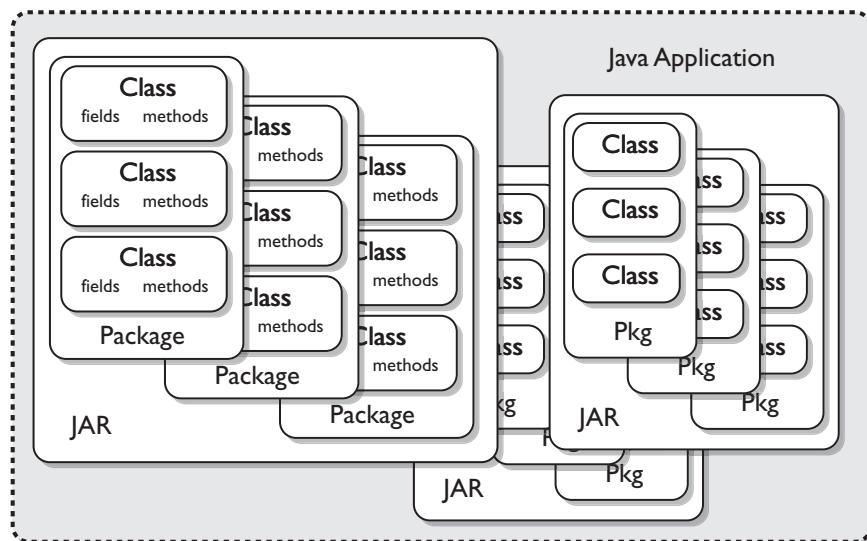


Figure 2–2 A Java application

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.

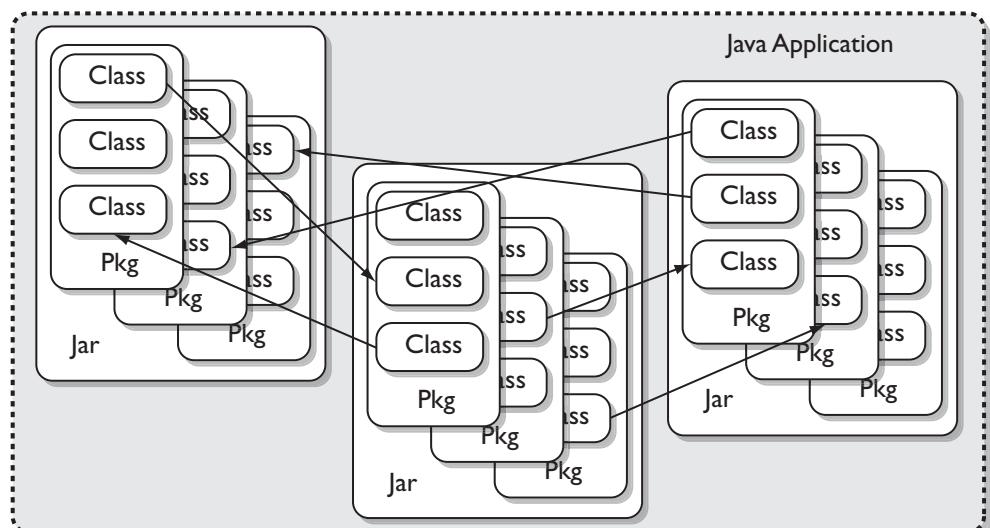


Figure 2–3 A monolithic application

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.

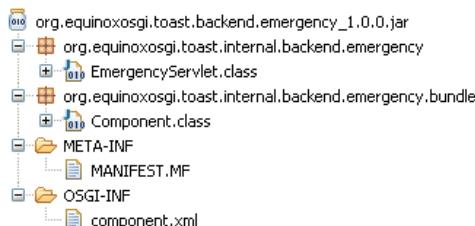


Figure 2–4 Bundle anatomy

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).

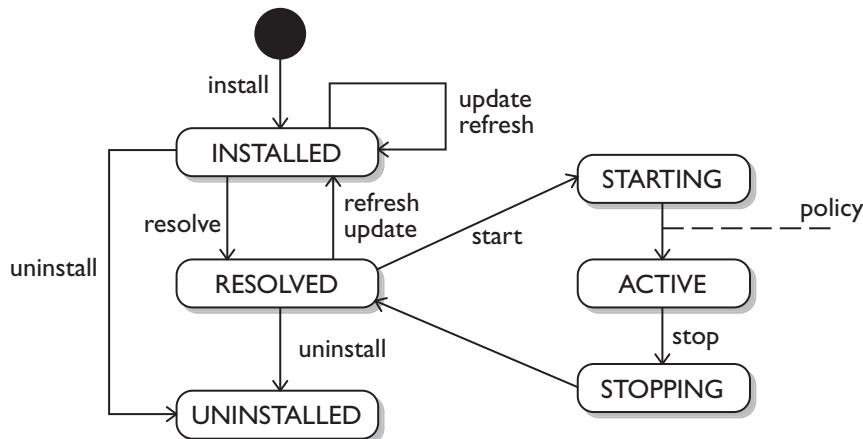


Figure 2–5 Bundle lifecycle

Every bundle starts its 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.

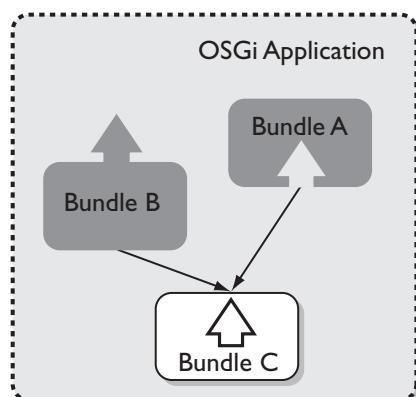


Figure 2–6 Service-based collaboration

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.

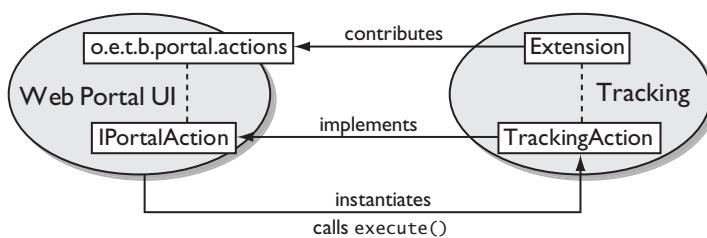


Figure 2–7 Extension contribution and use

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

\ (backslash), 249, 429–430, 440
/ (forward slash), 249, 358, 363, 440
\${buildDirectory}, 353, 365
\${buildDirectory}/features, 353, 365
\${buildDirectory}/plugins, 353, 365
\${variable} substitution, 353
&, 444
>, 444
<, 444
@noDefault, 433
@none, 433
@user.dir, 434
@user.home, 434

A

absolute:, 363
Abstraction hierarchy, 21–22
Action lookup, 209–213
action property, 213–215, 234–235
actions extensions, 276–280
activate attribute, 94–96, 101–103, 250,
 437–438, 447–450
Activation policy, 417, 420–422
Activators, 23–25, 62–64, 71, 78, 418–419
Active (bundle), 417
addAction, 211–213, 260–261
addExtension, 289–290
Addition, dynamic awareness, 375

addListener method, 52–53
Add-on features, 364–366
addProfile, 229–231
Agent (p2 architecture), 218–220, 229, 233
Airbag, 52–53, 167–169
Airbag Bundle, 60–61, 101
Airbag Bundle Activator, 94–95
Airbag domain logic, 263–265
Airbag service, 75–78
AirbagSimulator, 167–169
Alias, 308–313
Anonymous extensions, 282
Ant
 build scripts, 344
 documentation, 353
 pattern syntax, 346
 properties, 353, 356
 release engineering, 355
Apache Commons Logging, 304, 325,
 389–390
API surface area, 115–116
app (buddy policy), 398
app (Java application class loader), 424
AppenderHelper, 400–402
Appenders, 395–397
Application management, 184
Application model, 184–187
Application writers (consumers), 14
ApplicationDescriptor, 184

ApplicationHandles, 184
archivePrefix, 350
Artifact repositories, 219–220, 230, 237, 239, 352
Audio support, 226–227
Audio user interface, 181–184
Authentication and login, 314–317
Automated Management of Dependencies, 71–76, 91, 230
Automatic updating of dependencies, 73–75
Auto-start, 138, 146, 165, 419, 421–423
Auto-substitution, version numbers, 360–361

B

Back end
core bundles, 108–109
emergency bundle, 109–111
features, 222–225
launch, 121–122, 142, 148
tracking bundle, 199
backendDeployer.product, 239
backend.product, 150, 233, 325, 337
backend-war.product, 326, 329, 333, 337
Backslashes, 249, 429–430, 440
Base ID and location, 350–351
base property, 350
basearch, 349, 351, 365
baseos, 349, 351, 364
basews, 349, 351, 365
Basic tracking scenario, 201
Binary Build, 149–150, 318, 345
bind attribute, 90, 103, 111, 211, 251, 300, 443, 452
BindException, 319
bin.excludes, 346
bin.includes, 329, 346, 362
bnd, 162, 394
boot (buddy policy), 397
Boot delegation, 428

boot (Java boot class loader), 424
bootclasspath, 349, 352–353
Bridged configuration, 322–329
BrowseAction, 213–214, 235
Browsers, 166, 188–190, 202–205
Buddy Class, 397–399
Buddy policies, 396–398, 424
BuddyLoader, 424
Bug reports, 320
Build naming, 350
Build scripts, 347, 356–357
build target, 348
Build templates, 357
buildDirectory, 349–350, 353, 358–360, 365–367
-buildfile, 354, 366
buildID, 349–350
buildLabel, 349–350, 366–367
build.properties, 345–353, 364–365
buildType, 349–350
build.xml, 347, 364, 366
Bundle build.properties, 346–347
Bundle IDs, 59, 92, 270, 406–407
BundleActivator, 62–63, 71, 78, 86, 380, 410, 418, 421
Bundle-Classpath, 393–394, 412, 414
BundleContext, 80–81, 135–136, 410, 447–448
BundleListeners, 377–378, 418
Bundle-relative path, 440
Bundle(s)
activation policy, 419–421
API surface area, 115–116
directories, 411–413
dynamic-enabled, 380–381
events, 23, 418
folder-based, 19
granularity, 22, 32
host, 413
JARs, 411–413

lifecycle, 22–23, 416–419
manifest editor, 59–60
manifest file, 18–19
writers, 14
Bundle-SymbolicName, 18, 59–60, 78, 92, 116, 155, 279, 281
BundleTracker, 378–379
bundle-unique name, 437
Bundle-Version, 18–19, 60–61
Bundling by injection, 388–390
Bundling by reference, 392–394
Bundling by wrapping, 390–391
Bundling using bnd, 394

C

Cache management, 152
Caching, 284–290
Capability mechanism, 220
Carbon, 180, 193
cardinality attribute, 211, 252, 443, 445–446, 449, 452
catalina run/catalina stop, 331
Channel bundle, 112–116
ChannelMessage, 112–113, 119
Circular dependencies, 57
class attribute, 439
Class loading
boot delegation, 428
class lookup algorithm, 424
Class.forName, 395–399
context class loaders, 399–401
declaring imports and exports, 424
importing *versus* requiring, 426
JRE classes, 401–402
optional prerequisites, 426
optionality, 426
re-exporting, 427
serialization, 402–403
uses directive, 426–427
x-internal and x-friends, 428

Class lookup algorithm, 424
ClassNotFoundExceptions, 395, 400, 424
Classpath hell problem, 17
-clean, 152
clear*, 211
clearLog, 298–300, 304
Client features
audio support, 226–227
climate control, 227
emergency management, 228
GPS, 227
guidance system, 228
mapping, 227
shell feature, 225–226
Client tracking bundle, 199–204
client.builder
base ID and location, 350–351
build naming and locating, 350
build target, 348
build.properties, 348–353, 360–361
cross-platform building, 351
Java class libraries and compiler control, 352–353
p2 Repository, 352
PDE build target, 348
product and packaging control, 349–350
publishing to a p2 Repository, 352
SCM access control, 351–352
client.exe, 362
client.map, 358
Client/server interaction, 107
back end emergency bundle, 109–111
bundle API surface area, 115–116
channel bundle, 112–116
constants, 119
core bundles, 108–109
emergency monitor bundle, 116–119
logging, 120–121
properties, 119–120
Require-Bundle, 116–117
running toast, 121–123

Client-side dynamic deployment, 241–242
Climate control, 227
Climate user interface, 181–183
Cocoa, 180, 193
Code libraries, integrating
 bundling by injection, 388–390
 bundling by reference, 392–394
 bundling by wrapping, 390–391
 bundling using bnd, 394
 JARs as bundles, 388
 troubleshooting class loading, 394–403
Cohesion, 18, 22, 57, 116
Collaboration, 4–5, 24–26
CommandInterpreter, 407
commandline, 333
Command-line arguments, 431
CommandProvider, 270, 407–409
Commons Logging, 304, 325, 389–390
Comparing the workspace, 38
compilelogs, 355
Compiler control, 352–353
Compiler preferences, 54
Compiler warnings, 53
compilerArg, 349, 352
component <component id>, 270
Component XML schema v1.1.0
 <component> element, 436–439
 <implementation> element, 436, 439
 namespace and schema, 435–437
 <properties> element, 436, 440–441
 <property> element, 436, 439–440
 <provide> element, 436, 442
 <reference> element, 436, 442–444
 <service> element, 436, 441–442
ComponentConstants, 457
ComponentFactory Service, 266–268
component.factory property, 266
component.name property, 266
Component(s)
 activation/deactivation, 447–450
 definition, 154–156, 271–272
 enablement, 445
 factory, 262–269
 ID, 270
 immediate attribute, 255–256, 438, 453–454
 lifecycle, 444–457
 modification, 447–450
 properties, 454–457
 referenced service policy, 449
 Strict mode, 158
 versions and version ranges, 155–156
 x-friends directive, 158, 428
 x-internal directive, 157, 428
Components (DS), 98–100, 103, 199, 212–213, 248–250
component.xml, 98–103, 188, 191, 199, 249, 251, 253, 265, 298, 301, 304, 309, 408
Concierge, 28
Conditional Permissions Admin service, 27
config.ini, 362, 423, 428–429
configs property, 351, 365
Configurable tracking, 204
Configuration area, 432–434
Configuration elements, 279–280
Configuration page, 146
ConfigurationAdmin, 201–205, 304, 437–439, 446, 454
configuration-policy attribute, 438
configurator.xml, 266
Configuring Equinox, 428–432
-console, 319, 332, 333, 406
-consolelog, 354
Constants, 119
Constraint solver SAT4J, 221
Container, 335
Context class loaders, 399–401
Context Finder, 401
Context (logging), 304–306
Contexts, 311, 314–318

- Continuation character \ (backslash), 249, 429–430, 440
Contribution IDs, 281
Control properties, 346–347
ControlCenter, 258, 336–338
Copy into Workspace, 38
Core bundles, 108–109, 198
Core tracking bundle, 198–199
Coupling, 17, 22, 24, 57, 68–70, 301, 426, 442
CreateAction, 234
createComponent, 267–268
createContext, 316
createDefaultHttpContext, 308–309
createExecutableExtension, 279–280, 289, 396
Cross-platform issues, 154, 344, 351, 365
CruiseControl, 367
Crust shell, 174–175, 225–226, 361–362, 419
Crust widgets, 175
crust.ini, 362
Ctrl-key commands, 46–47, 53, 73, 170
Custom callbacks, 347, 356–357
customAssembly, 357
customTargets.xml, 359
cvs, 359
- D**
- D, 120, 353, 422, 428, 428, 431
Data areas, 432–434
deactivate attribute, 94–96, 101–103, 250, 380, 437–438, 447–450
Debugging
 DS apps, 269–270
 missing property files, 441
 release engineering, 355–356
Declarative Services (DS), 247, 435
 accessing referenced services, 450–453
 activate, deactivate, and modified, 437–438, 447–450
 <component> element, 436–439
components, 98, 199, 292, 453–457
console commands, 270
debugging, 269–270
-Dequinox.ds.print=true, 104, 121, 270, 441
editor, 99
factory components, 262–269
immediate components, 255–256, 438, 453–454
<implementation> element, 436, 439
launching, 269–270
lifecycle, 444–457
model, 247–248
modifying the bundles, 98–104
naming conventions, 103
PDE tooling, 270–273
<properties> element, 436, 440–441
<property> element, 436, 439–440
<provide> element, 436, 442
providing services, 253–255
<reference> element, 436, 442–444
referencing services, 250–252, 254–255
root element, 437
scr namespace identifier, 436
<service> element, 436, 441–442
start levels, 383
target properties, 456–457
unbind methods, 102
Whiteboard Pattern, 256–262
XML schema v1.1.0, 435–444
Decoupling, 67, 70, 82, 126, 257
delay (property), 200–204
delayChanged, 200, 203–204
Delayed component instantiation, 213
Delta pack, 40, 43–44, 153–154, 348
Dependencies, 57, 61, 72–75
Dependencies page, 144–146
Dependency injection, 55, 69, 87, 118, 126, 135, 211, 257
dependent (buddy policy), 398
Deployed (bundle), 416

-Dequinox.ds.print=true, 104, 121, 270, 441
Device drivers, 161
Device simulation, 165–167
DeviceSimulatorServlet, 166, 170
diag command, 407
Directory bundle layout, 411–412
dis (disable component), 270
Discouraged access, 157–158, 230–232
Discovery service, 335
dispose, 186, 268, 380–381
Distributed system, 340–341
Distributed Toast, 335–336
doGet method, 109–110, 202, 208–209
Duplication approach, 162
Dynamic best practices, 371
 dynamic awareness, 374–378
 dynamic enablement, 379–382
Extender Pattern and BundleTracker,
 378–379
services, 383–384
start levels, 382–383
startup and shutdown, 382–385
Dynamic configuration, 197
 back end tracking bundle, 199
 basic tracking scenario, 201
 client tracking bundle, 199–204
 configurable tracking, 204
 ConfigurationAdmin, 201
 core tracking bundle, 198–199
 persistent configuration, 205
 tracking scenario, 197–198
Dynamic extension, 284–291
dynamic (policy attribute setting), 211, 444,
 449
Dynamic services, 85
 Declarative Services (DS), 97–105
 Service Activator Toolkit (SAT), 86, 93–97
 Service Trackers, 86–93
 StartLevel service, 85
DynamicImport-Package, 399–400, 402

E

Easymock, 126–130, 132
Eclipse-BuddyPolicy, 397–398
Eclipse Communication Framework (ECF),
 115, 322, 334–335
Eclipse Delta pack, 40, 43–44, 153–154, 348
Eclipse-ExtensibleAPI, 414–415
Eclipse Help online, 220
Eclipse-PlatformFilter, 414
Eclipse Integrated Development Environment
 (IDE), 10, 37, 40, 248, 277, 355, 372
Eclipse-RegisterBuddy, 397
Eclipse Rich Ajax Platform (RAP), 208, 314
Eclipse Rich Client Platform (RCP) SDK, 40, 44
Eclipse SDK, 143, 248
Eclipse Update Manager, 217, 222
eclipse.ignoreApp, 104–105
eclipse.product, 429
Ecosystems, 7
Emergency bundle, 102–104, 109–111
Emergency domain logic, 176–178
Emergency management (client feature), 228
Emergency monitor bundle, 95–96, 116–119
Emergency user interface, 176, 179–181
EmergencyMonitor, 54, 88–90, 120, 127
EmergencyMonitorTestCase, 128, 130
EmergencyServlet, 109–111, 122, 170
enableComponent, 438, 445–446
enabled attribute, 438, 445
enableFrameworkControls, 333
Ensemble, 8, 10–11
entry attribute, 440
Equinox, 27–28
 cache management, 152
 configuring and running, 428–432
 console, 406–409
 LogService implementations, 304–306
 p2, 218–221, 236–239, 352, 425
 SDK, 40, 44, 153, 313
 Servlet Bridge, 324–328, 333

- Equinox x-friends, 156–158, 428
Equinox x-internal, 156–158, 428
Event listener bundle, 257
Event source bundle, 257
Event sources, 210, 377
Event strategy, 450–451
Events, 23
.exe, 148
Executable, 147–148, 282–283, 292, 363, 430–432
Executables feature, 348
Execution environment, 147–148
Export directives, 428
Exported Packages, 60, 73–74, 116, 156–158
Exporting Toast, 149–152
Export-Package, 18–20, 60–61, 115–116, 156, 401, 427
ext (buddy policy), 397
ext (Java class loader), 424
extendedFrameworkExports, 333
ExtendedLogEntry, 304–306
ExtendedLogReaderService, 304–305
ExtendedLogService, 304–306
Extender Pattern and BundleTracker, 378–379
Extensible user interface, 173
 climate and audio, 181–184
 crust, 173–175
 emergency scenario, 175–181
 navigation and mapping, 187–194
 OSGi application model, 184–187
Extensible Web portal, 207–216
ExtensionActionLookup, 278–279
Extension(s), 25, 275
 addExtension, 289–290
 anonymous, 282
 caching, 284–287
 contribution IDs, 281
 deltas, 283–284, 286–288
 dynamic, 284–291
 and extension points, 25–26, 278–280, 312–313
Extension Registry, 275–278, 283–284, 292–293
factories, 282–283, 292
named, 282
object caching, 287–290
removeExtension, 289–290
services and extensions, 290–292
singletons, 279, 281, 291
tracker, 289–290
External bundle JARs, 393
extra.<library>, 346
- ## F
- Factory components (DS)
 airbag domain logic, 263–265
 ComponentFactory Service, 266–268
 declaring, 265–266
 launching Toast, 269
Factory ID, 439
Fake Airbag, 163–164
Fake GPS, 164
Feature build, 362–366
Feature Builder, 364–366
feature IDs, 224
feature.builder/build.properties, 364
featurefragmentplugin@elementId=, 358
features/customBuildCallbacks, 357
FeatureSync, 241–242
Felix, 28
Fetching from an SCM system, 358–360
Fetching the maps, 360
Fetching the product file, 359
fetchTag, 352
FileLocator, 412
Firefox, 188, 192
Folder bundles, 391
Folder-based bundles, 19
Food chain, 75, 94, 111, 177, 255, 373–374, 454
forceContextQualifier, 361

Forward slash, 249, 358, 363, 440
Fragment bundles, 128, 249, 413–415
framework (class loader), 424
Framework events, 23
Framework implementations, 27–28
Framework (OSGi), 26–27
frameworkLauncherClass, 333
FrameworkUtil.getBundle, 377
Friends directive, 158
Futures (IFuture object), 340

G

Galileo SR1 release, 36
Generic types, 53
getAvailableFeatures, 229, 231–232
getBundle, 297, 378
getCache, 286
getContextClassLoader, 401
getException, 297
getExtension, 282, 284, 287–288
getExtensionDeltas, 283–284, 287–288
getExtensionPoint, 284
getExtensionRegistry, 289
getHelp, 407–408
getImportedService, 95
getImportedServiceNames, 95–96, 98
getKind, 263
getLevel, 297
getLog, 297
getLogger, 305
getMapFiles, 360
getMessage, 297
getObjects, 290
getOrientation, 263
getRow, 263
getServiceReference, 297
getServlet, 289–290
getTime, 297
getUsingBundle, 442

GlassFish, 321
Google Earth, 180, 187–195, 227–228
GPS, 169
GPS bundle, 58, 98–100
GPS Bundle Activator, 94
Gps class, 51–52
GPS service, 69–75
Granularity, 22, 32
Guidance system, 228

H

Headless, 108, 197, 239, 354, 357, 394
headless-build, 357, 364
Host bundle (fragments), 413
HTTP support, 307
BindException, 319
Configuring the port, 312
contexts, 314–318
extensions, 312–313
JAAS, 314–318
Jetty, 313–314
port query, 319–320
registering a servlet, 309–313
secured client, 317–318
http.address, 320
http.port, 320
HttpRegistryManager, 288–289
http.schema, 320
HttpService, 109–111, 121, 166, 285,
 308–314, 318–320, 374, 376
HttpServlet, 109–110, 199, 208–209
http.timeout, 319–320
Hudson, 367

I

IActionLookup, 210–211, 214
IAirbagListener, 52–54
IArtifactRepositoryManager, 230

- IChannel, 112, 114, 115, 119, 131, 200, 254, 338
IClimateControl interface, 181–183
IClimateControlListener interface, 181–182
Icons, 147–148, 174, 178–180, 183–184, 273
ICrustDisplay, 174, 186, 292, 419
ICrustShell, 174, 178, 183, 191, 251, 256
IDs, 281, 350
IEngine, 230
IExecutableExtension, 283
IExtension, 288–290
IExtensionChangeHandler, 288–289
IExtensionDeltas, 283–284, 286–288
IExtensionRegistry, 282
IExtensionTracker, 288–289
ignore (configuration-policy), 439
IGoogleEarth, 189–192
IGps, 70–71, 73, 79–80, 99–100, 169, 190, 200, 253
Illegal XML characters, 444
IMappingScreen, 191–192
IMetadataRepositoryManager, 230
Immediate components, 255–256, 438, 453–454
Implementation, 162–165, 178
<implementation> element, 436, 439
Imported Packages, 61, 73–74
Importing *versus* requiring, 426
Import-Package, 19–21, 78, 394
Imports and exports, declaring, 424
Imports, organizing, 53, 73
Initialization file, 430
initializeTracker, 289–290
init-param, 328, 332
initparams, 309–310
injection, 87, 388–390
Install area (Equinox data area), 432–434
InstallAction, 235
Installed (bundle), 23, 416
install.xml, 235
Instance area (Equinox data area), 432–434
integer, 448
Integrating code libraries, 387
bundling by injection, 388–390
bundling by reference, 392–394
bundling by wrapping, 390–391
bundling using bnd, 394
JARs as bundles, 388
troubleshooting class loading problems, 394–403
interface attribute, 442–443
Interface duplication, 162
Interface/implementation separation, 138, 162–165, 178
Interfaces, 162–165
Internal directive, 157
Internal packages, 71, 115–116, 157–158
Internet Explorer, 192
Introspection, 332, 377, 406
IPlanner, 230
IPortalAction, 25–26, 209–215, 258–262, 276–281
IProfileRegistry, 230
IProvisioner, 229
IRegistryChangeEvent, 283, 286–288
IRegistryChangeListener, 283, 286
ISafeRunnable, 377
ITrackingConstants, 198, 200, 203
IUs (installable units), 219–222, 231–232
- J**
- jarsigner, 36
Java Archives (JARs), 15–18, 188, 346, 357, 388, 391–395
Java Authentication and Authorization Service (JAAS), 307, 314–318
Java class libraries, 352–353
Java database connectivity (JDBC), 392–394
Java Development Tools (JDT), 36
Java Runtime Environment (JRE), 36, 356, 401–402, 425

Java Servlet API, 309
Java Software Development Kit (SDK), 36, 330
Java system components, 15–18
Java virtual machine (JVM), 15
JavaScript, 190
javax.servlet, 109, 325
javax.servlet.http, 325
javax.servlet.http.HttpServlet, 209
javax.servlet.resources, 325
Jetty, 313–314, 322–323
Jingle, 334
JMock, 127
join, 77, 384–385
JUnit, 126, 128–130, 136–139
junit.jar, 411

K

Key/value pairs, 120, 285, 328, 347
Keyboard shortcuts, 46–47, 53, 73, 170
Knopflerfish, 28

L

label property, 214–215
<<lazy>>, 420
Launch configuration, 63–64, 148, 205
Launcher Name, 148
Launching page, 147–148
launch.ini, 328
Lazy activation, 417, 419–420, 422
LDAP filters, 406, 414, 444, 456
Licensed material, 391
Lifecycle (bundle), 22–23, 291, 410, 416
Linux, 192, 319
listBundles, 407–408
Listener/Observer Pattern, 52–54, 77, 79, 86, 177–179, 182, 257, 283, 286–288, 292, 298, 300–305

Listeners
 addListener method, 52–53
 BundleListeners, 377–378
 dynamic best practices, 376
 Event listener bundle, 257
 IAirbagListener, 52–54
 IClimateControlListener interface, 181–182
 IRegistryChangeListener, 283
 registry, 286, 288, 381
 removeListener method, 52–55
 service listeners, 86
 weak listener list, 377
locateService, 103, 259–260, 452–453
Logging, 295
 clearLog, 298–300, 304
 getLog, 297
 getLogger, 305
 log appenders, 395
 log service specification, 295–298
 LogEntry, 297
 LogFilter, 305
 logging levels, 296
 logInfo, 299–300
 LogListener, 298
 LogReaderService, 295, 297–298, 301–303
 LogService, 133–134, 298–301, 304–306
 LogUtility, 119–121, 133, 303–304
 setLog, 135–137, 298–300, 304
 writing to the log, 296–297
Logical location, 312
Login, 314–318
log4j, 304, 395
LogService.LOG_DEBUG, 296
LogService.LOG_ERROR, 296
LogService.LOG_INFO, 296
LogService.LOG_WARNING, 296
LogUtility, 119–121, 133, 303–304
Lookup algorithm, 424
Lookup strategy, 450–452

M

Mac platform issues, 180, 185–186, 193
main class, 55–56
ManageAction, 235
Map, 447–448
Map files, 351, 358–360
Mapping, 187–192, 227
mapsCheckoutTag, 349, 351, 360
mapsRepo, 349, 351, 360
mapsRoot, 349, 351, 360
Marker properties, 351–353
mBedded Server, 28
Metadata, 219–220
Metadata repositories, 219–220, 230–231, 237–239, 352
META-INF, 18, 60, 149, 346
MockAirbag, 127, 131–138
Mocking, 127, 132
MockLog, 131, 133–136
modified attribute, 437–438, 447–450
Modularity, 20–22

N

name attribute, 437, 440, 443
Named extensions, 282
Names with . (dot), 455
Namespace, 220
namespace and schema, 435–437
Naming conventions, 103, 224, 455
NASA, 8, 10–11
Native configuration, 322
Navigation and mapping
extensibility, 191–192
Google Earth integration, 187–190
mapping support, 191
user interface, 192–194
Navigation operations, 46–47
Nested DS components, 437
Nested reference elements, 250

netstat -anb, 319, 331
NetWeaver, 321
newInstance, 268, 454
NoClassDefFoundErrors, 395, 400

O

Object constructors, 380
Object handling, 376–377
Observer Pattern, 177, 181
Observers, 376–377
Open Services Gateway initiative, 13n
optional (configuration-policy), 439
Optional prerequisites, 426
Optional service, 446, 449
Orbit, 390
OSGi Alliance, 13
OSGi application model, 184–187
OSGi Mobile Expert Group (MEG), 184
OSGi R4.2 Enterprise Expert Group (EEG), 333–334
osgi.bundles, 422–423, 429
OSGI-INF, 98
osgi.instance.area, 429
osgi.noShutdown, 104–105
Overview page (Toast), 143–144

P

Package Explorer, 53, 56, 70–71, 150
Package filtering, 61
Package visibility, 156–158
Packaging, 141
Binary Build, 149–150
component definition, 154–158
exporting Toast, 149–152
launch configuration, 142–143
platform-specific code, 152–154
product configurations, 141–149
versions and version ranges, 155–156

- Parallel development, 161, 165
parent (buddy policy), 398
Password, 314–318
Path separator, 440
PDE (Plug-in Development Environment), 36, 148, 152
Build, 344–345, 362–363
Build target, 348
Build templates, 357
tooling, 270–273
Permissions, 27
perm_pattern, 363
Persistent configuration, 205
Persistent ID (PID), 198, 203, 319–320, 437, 439, 445–446, 455
Persistent starting, 422
Physical location, 312
Platform, 7, 26–27
Platform-specific code, 152–154
Platform-specific issues, 180
Pluggable services, 161–170
pluginPath, 349, 351, 357
Plug-ins view, 47
plugins/customBuildCallbacks, 357
plugin.xml, 26, 47, 276
POJO, 33, 126
policy attribute, 252, 443–444, 449
Port 8080, 121
Port 8081, 170, 312
Port management, 319–320, 331, 456
Port query, 319–320
Portal actions, 212–215
PortalServlet, 208–210, 256, 258
postSetup, 359
processSync, 241
Producers (bundle writers), 14
Product configurations
 configuration page, 146
 creating, 142–143
 dependencies page, 144–146
 launching page, 147–148
 overview page, 143–144
 productizing the client, 149
 running the product, 148
 sync with launch configuration, 148
Product files, release engineering, 359
product (property), 349
productBuild.xml, 348, 356, 366
Productizing the client, 149
Profile registry, 230
Properties, 119–120
<properties> element, 436, 440–441
<property> element, 436, 439–440
PropertyManager class, 111, 119–120, 122
Protocol providers, 335
<provide> element, 253, 436, 442–443
providing services, 68–69, 98–101, 253–255
Provisioner, 229–233
Provisioning UI, 235
p2 (Equinox)
 architecture, 218–219
 artifacts, 220
 director, 221, 239
 engine, 221
 metadata, 219–220
 profiles, 220, 425
 repository, 220, 236–238, 352
 wiki, 220
p2.artifact.repo, 349, 352, 365
p2.compress, 349, 352, 365–366
p2.gathering, 349, 352, 365–366
p2.inf, 231, 237
p2.metadata.repo, 349, 352, 365

Q

- qualifier, 155, 361–362
Qualifying version numbers, 361–362
Quality, 22, 126, 131, 248
Query, 210, 231, 286, 305, 409

R

Raw types (compiler warnings), 53, 283
Re-exporting bundles, 427
Refactoring, 59, 70, 71, 79, 176
Reference (bundling), 392–394
<reference> element, 103–104, 250–252, 261, 436, 442–444, 449–452, 456
Reference implementation (RI), 9
Referenced services, 68–69, 102–104, 250, 446, 450–453
REF_WEAK, 289–290, 377
registered (buddy policy), 396–398
Registering/unregistering a servlet, 309–313
registerObject, 289–290
registerResources, 308–311
registerService, 73, 186
registerServlet, 111, 210, 309–311
Registration Pattern, 210
Registry change events, 283, 286–288
Registry change listeners, 286, 288, 381
Regression testing, 164–165
Release engineering, 343
 add-on features, 363–366
 Ant properties, 353
 build.properties, 345–353, 360, 364–365
 client.builder, 347–353
 control properties, 346–347
 cross-platform building, 351
 custom build scripts, 347, 356–357
 debugging, 355–356
 Eclipse delta pack, 348
 feature.builder, 364–366
 fetching from an SCM system, 358–360
 Java class libraries, 352–353
 map files, 358–360
 p2 repository, 352
 PDE Build, 344–345
 product files, 345, 359
 repositories, 352, 357
 root files, 362–363

SCM access control, 351–352
SCM integration, 359–360
version numbers, 360–362
WARs, 367
Remote Method Invocation (RMI), 402
Remote Services, 322, 333–341
Remote service client, 338–339
Remote service host, 336–337
Removal, dynamic awareness, 375
removeAction, 211–212, 260–261
removeExtension, 289–290
removeListener method, 52–55
repoBaseLocation, 352, 357, 364–365
Repositories, 219–220, 230–231, 236–238, 242, 352, 357
repository directory, 355
require (configuration-policy), 439
Require-Bundle, 19–21, 78, 116–117
Required services, 80, 97, 104, 446
Resolved (bundle), 17, 23, 283–284, 417
Reverse domain name convention, 59
Root element, 437
Root files, 362–363
root.<os.ws.arch>, 362
Roots, 231
ROOT_TAG, 232
ROOT.war, 330
running, 385
Runtime, 26–27

S

Safari, 188, 192
SafeRunner, 377
Samples Manager, 37–39
SAT4J constraint solver, 221
SAX (Simple API for XML) classes, 425
SCM access control, 351–352, 365
SCM integration, 359–360
SCM system, 358–360

- scr namespace identifier, 436
Secured client, 317–318
Security, 27, 314–318
Semantic naming, 103
Separation approach, 162–164
Separation of concerns, 210, 301, 311, 324
Serialization, 402–403
Server side, 321
 bridged configuration, 322–333
 distributed Toast, 335–336
 Eclipse Communication Framework (ECF),
 334–335
 embedding OSGi, 323–333
 <init-param>s, 333
 native configuration, 322
 remote service client, 338–339
 remote service host, 336–337
 Remote Services, 335
 servers and OSGi, 322–323
 service discovery, 339–340
 Service Location Protocol (SLP), 339–340
 Servlet Bridge, 324–325, 327–328, 333
 solo configuration, 322
 Web Application Root (WAR) files, 326–329
 Zeroconf, 339
Service Activator Toolkit (SAT), 93–97, 303
Service Component Runtime (SCR), 98, 248,
 270
Service discovery, 339–340
<service> element, 436, 441–442
Service events, 23
Service listeners, 86
Service Location Protocol (SLP), 339–340
Service platform (sp) commands, 331–332
Service registry, 24–26
Service Trackers, 86–93
ServiceActionLookup, 211–213, 259–261
Service-based action lookup, 210–213
Service-Component, 100, 164, 249, 379,
 413, 444
 servicefactory attribute, 441
 service.id property, 409
 service.ranking property, 409
 ServiceReference, 80, 186, 212, 261, 296,
 451–452
 ServiceRegistration, 73
 Services, 17, 24, 67, 85
 acquiring, 79–81
 definition, 67
 and extensions, 290–292
 launching, 81
 optional, 446, 449
 registering the airbag service, 75–78
 registering the GPS service, 69–75
 required, 80, 97, 104, 446
 troubleshooting, 82
 services command, 406
 ServiceTrackerCustomizers, 88–90
 servlet, 327–328
 Servlet Bridge, 324–328, 333
 servletbridge.jar, 327
 servlet-class, 327–328
 servlet-mapping, 328
 set*, 211, 230
 setAirbag method, 54–56
 setChannel, 118–119
 setGps method, 55–56
 setHttp, 110–111, 309–310
 setLog, 135–137, 298–300, 304
 setUp, 135–136
 Shell feature, 225–226
 short status command, 91–92, 332, 406, 415
 Signing, 391
 Simple Configurator, 333, 383, 429
 SimpleLoginModule, 317
 Simulated devices, 167–170
 Simulator framework, 165–167
 Singletons, 120, 279, 281, 291, 415–416
 skipBase, 349, 351, 353, 364
 skipFetch, 349, 352, 359, 365

- skipMaps, 349, 352, 360, 365
SoftReferences, 377
solo configuration, 322
some.bundle/plugin.xml, 282
source.<library>, 346
sp_deploy, 331
sp_redeploy, 331
sp_start, 331
sp_stop, 331
sp_test, 331
sp_undeploy, 331
special_executable, 363
Splash screen, 430
Split packages, 76
Spring DM, 321
ss command (short status), 91–92, 332, 406, 415
Standard Widget Toolkit (SWT), 35, 173–175, 179–180, 188, 190–192, 414–415, 427
Start levels, 82, 146, 382–383
Starting, 422
Starting (bundle), 417
StartLevel service, 85
startup method, 55, 78, 136–137, 265, 268
Static imports, 129
static (policy attribute setting), 443, 449
Stopping (bundle), 417
Strict mode, 158
svn, 359
Symmetry, 55
Synchronization (Product Export), 151
Synchronized launch and product configurations, 148
System Bundle, 26–27, 425
System deployment with p2, 217
 architecture, 218–219
 artifacts, 220
 back end features, 222–225
 client features, 225–228
 client-side dynamic deployment, 241–242
 director, 221
engine, 221
exporting, running, and provisioning, 235–241
feature IDs, 224
p2 metadata, 219–220
profiles, 220
provisioner, 229–233
repositories, 220
Web UI, 233–235
System integrators (consumers), 14
SystemTestCase, 135, 137
System-testing Toast, 131–139
- T**
- target attribute, 444, 456
Target editor, 41–45
Target platform, 39–46, 104, 138
Target properties, 456–457
Telematics, 31–32, 35, 49–56
Templates, 347–349, 356–357, 364, 367
Testing, 125
 Easymock, 126–130, 132
 fragment bundles, 128
 JUnit, 126, 128–130, 136–139
 mocking, 127, 132
 regression testing, 164–165
 static imports, 129
 system-testing, 131–139
 test cases, 127–130, 135–137
 test harness, 127, 131–134
 unit-testing, 126–130
Tickle, 235, 241–242
Toast
 dynamic-awareness, 375
 evolution, 34–35
 exporting, 149–152
 sample code, 36–39
 target content, 46
 target platform setup, 39–46

ToastBackEnd, 142, 148, 170, 229, 239
 ToastLogReader, 301, 306
 toast.war, 330–331
 toast.zip, 238, 330
 toFileURL, 188–189
 Tomcat, 323, 331
 topLevelElementID, 364–365
 topLevelElementType, 364–365
 Touchpoints, 218, 221
 Trackers, 289–290
 Tracking code, 198–201
 Tracking Scenario, 197–198
 TrackingConfigServlet, 202–204, 310–312
 TrackingMonitor, 197–203
 TrackingServlet, 197–199
 transformedRepoLocation, 357, 364–365
 Troubleshooting
 class loading problems, 394–403
 ClassNotFoundExceptions, 395, 400, 424
 HttpService, 318–320
 server side, 332–334
 services, 82
 Twitter, 334
 type attribute, 440

U

unbind attribute, 88–90, 102–104, 211, 251, 300, 304, 443
 ungetService, 80–81
 UninstallAction, 235
 Uninstalled, 23, 378, 417
 uninstall.xml, 235
 Unit-testing Toast, 126–130
 unregister, 309–310
 Unresolved, 23, 281, 378
 Unzip, 389, 391
 updateDelay, 200–203
 URL, 44, 188, 331

URL pattern, 327–328
 UrlChannel, 114–115, 122
 URLConverter, 188
 User Admin service, 27
 User area (Equinox data area), 433–434
 User interface
 climate and audio, 183–184
 emergency, 176, 178–181
 navigation and mapping, 192–194
 See also Extensible user interface
 uses directive, 426–427
 UTF-8, 249
 util package, 427
 Utility classes
 constants, 119
 logging, 120–121
 properties, 119–120

V

Validity testing, 376
 value attribute, 439
 Version numbers, 155, 297, 360–361, 364–365
 VM arguments, 104, 120–121, 137, 158, 170, 432
 void <method>, 451

W

wait, 384–385
 war.builder/customTargets.xml, 367
 watchFor, 134–136
 Weak listener list, 377
 WeakReferences, 289–290, 377
 Web archive (WAR), 326–333, 367
 Web interface, 165–166, 169–170, 204
 Web portal, 207
 action lookup, 209–213
 delayed component instantiation, 213
 portal actions, 212–215

PortalServlet, 208–210, 256, 258
Whiteboard Pattern, 210, 215–216
Web UI, 233–235
WEB-INF, 326, 330–332
WebSphere, 321, 323
web.xml, 326–327, 332–333
Whiteboard Pattern, 210, 215–216, 256–262,
 377
Whitespace in config.ini files, 430
Widgets, 173–175
Wildcards, 46, 249
Windows, 192, 360
Workspace bundles, 63, 138, 393
Wrapping a code library, 390–391

X

Xalan, 401
x-friends, 156–158, 428
x-internal, 156–158, 428
XML, 26, 98, 248–249, 261, 401
XML namespace and schema, 435–437
XML-compliance, 437
XMPP, 334–335

Z

Zeroconf, 339
Zip, 238, 344, 352, 357, 367, 389, 391
Zombie bundles, 375