Christoph Schmidt-Casdorff
Thorsten Vogel

OSGi

Christoph Schmidt-Casdorff
Thorsten Vogel

OSGi

Einstieg und Überblick

entwickler.press

Christoph Schmidt-Casdorff, Thorsten Vogel
OSGi – Einstieg und Überblick

ISBN: 978-3-86802-223-0

© 2009 entwickler.press
Ein Imprint der Software & Support Verlag GmbH

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Verlag GmbH
entwickler.press
Geleitsstraße 14
60599 Frankfurt am Main
Tel: +49(0) 69 63 00 89 - 0
Fax: +49(0) 69 63 00 89 - 89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktion jeglicher Art (Fotokopie, Nachdruck, Mikrofilm, Erfassung auf elektronischen Datenträgern oder andere Verfahren) nur mit schriftlicher Genehmigung des Verlags. Jegliche Haftung für die Richtigkeit des gesamten Werks kann, trotz sorgfältiger Prüfung durch Autor und Verlag, nicht übernommen werden. Die im Buch genannten Produkte, Warenzeichen und Firmennamen sind in der Regel durch deren Inhaber geschützt.

Inhaltsverzeichnis

1 Einleitung 9

1.1 Modularität 10
1.1.1 Komponentenmodell 10
1.1.2 Schwächen von Java 11
1.2 Was ist OSGi? 12
1.2.1 Historischer Abriss 14
1.3 Weitere Komponentenmodelle der Java-Welt 15
1.3.1 Java Module System 15

2 OSGi-Konzepte 17

2.1 Bundles – Module in OSGi 17
2.2 Expertenwissen: Bundles 20
2.2.1 Aufbau der Manifest-Datei 20
2.2.2 Resolving Process – Auflösung der Bundle-Abhängigkeiten 24
2.2.3 Bemerkungen zum Laufzeitverhalten 25
2.2.4 Bundle Space und Class Space 27
2.2.5 Implizite Imports – uses-Direktive 27
2.2.6 Import-Package versus Require-Bundle 29
2.2.7 Import/Export von Packages 30
2.3 Fragment Bundles 31
2.4 Expertenwissen: Fragment Bundles 32
2.4.1 Fragments und Manifest-Datei 32
2.4.2 Betrachtungen zum Classloading 33
2.4.3 Einsatz von Fragments 34
2.5 Lebensweg von Bundles 34
2.6 Expertenwissen: Lebensweg von Bundles 36
2.6.1 Objekte des Lifecyle Layers 36
2.6.2 Bundle-Objekt 37
2.6.3 Bundle Context 37
2.6.4 Persistent Storage 38
2.6.5 Environment Properties 38
2.6.6 System Bundle 39

2.6.7	Lebenszyklus eines Bundles	39
2.6.8	Bundle Activation/Deactivation	42
2.6.9	Zugriff auf Ressourcen	43
2.6.10	Änderungen von exportierten Packages	45
2.6.11	Lebenszyklus von Fragments	45
2.6.12	Events und Listeners	46
2.7	OSGi-Services	48
2.8	Expertenwissen: OSGi-Services	50
2.8.1	Was ist ein OSGi-Service?	50
2.8.2	Classloading bei Services	52
2.8.3	Service Registry	53
2.8.4	Registrierung von Services	53
2.8.5	Service Properties	54
2.8.6	Service Registration	55
2.8.7	Service Reference	56
2.8.8	Service Ranking	56
2.8.9	Nutzung von Services	57
2.8.10	Abfrage und Zugriff auf Services	57
2.8.11	Get Service/Unget Service	58
2.8.12	Filter	58
2.8.13	Service Events und Listeners	59
2.8.14	Service Factories	59
2.8.15	Betrachtungen zu Multiple Version Export	60
2.8.16	Bemerkungen zur Service Registry	61
2.9	OSGi-Framework	61
2.9.1	Open-Source-OSGi-Frameworks	63
2.10	Expertenwissen: OSGi-Framework	65
2.10.1	Package Admin Service	65
2.10.2	Start Level Service	66
2.10.3	Deployment Admin Service	68
2.10.4	Execution Environment	70
2.10.5	Startup und Shutdown	71
2.11	Security	71
2.12	Expertenwissen: Security	73
2.12.1	Java 2 Security Model	73
2.12.2	Bundle Permission Resource	75
2.12.3	Conditional Permission	75
2.12.4	Privileged Callbacks	76

3 Entwicklung mit OSGi 77

3.1 Konsequenzen der Dynamik in OSGi 77
3.2 Expertenwissen: Dynamik in OSGi 79
3.2.1 ServiceTrackerCustomizer 79
3.2.2 ServiceTracker und Kardinalitäten 80
3.2.3 Stale References 80
3.2.4 Multithreading 81
3.3 OSGi-Design-Patterns 82
3.3.1 Whiteboard-Pattern 82
3.3.2 Extender-Pattern 84
3.4 Expertenwissen: Patterns im Detail 86
3.4.1 Vergleich der Patterns 86
3.4.2 Bundle Tracking 87

4 OSGi-Standard-Services 89

4.1 Event Handling in OSGi 89
4.1.1 LogService 90
4.1.2 EventAdmin-Service 91
4.1.3 Expertenwissen: Event Handling 93
4.2 Konfiguration 96
4.2.1 Configuration Admin Service 96
4.2.2 Metatype Service 100
4.2.3 Preferences Service 102
4.3 OSGi und andere Welten 104
4.3.1 HttpService 104
4.3.2 OSGi und Webanwendungen 108
4.4 Überblick über weitere Standardservices 111
4.4.1 ApplicationAdmin-Service 111
4.4.2 UserAdmin-Service 112
4.4.3 WireAdmin-Service 114

5 Service Component Models 119

5.1 Declarative Services 119
5.2 Expertenwissen: Declarative Services 123
5.2.1 Laufzeitrepräsentierung von Komponenten 123
5.2.2 Factory Component 124
5.2.3 Umgang mit Referenzen 125
5.2.4 Filterkriterien für Servicereferenzen 128
5.2.5 Component Configuration und Properties 128

5.3 Spring Dynamic Modules 129
5.4 Expertenwissen: Spring Dynamic Modules 131
5.4.1 Spring Extender Bundle 131
5.4.2 Export von Services 133
5.4.3 Dynamik von Services 134
5.4.4 Übersicht über die unterschiedlichen Listener 137
5.4.5 Spring DM und Configuration Admin Service 138
5.4.6 Thread Context Classloader 141
5.4.7 Spring-spezifische Manifest-Einträge 142
5.4.8 Ausblicke 143
5.5 Vergleich von Declarative Service und Spring DM 144
5.6 iPOJO 145
5.7 Expertenwissen: iPOJO 146
5.7.1 Handler 146
5.7.2 Strategien zur Instanzierung von Services 147
5.7.3 Umgang mit Referenzen 147
5.7.4 Bytecode-Manipulation 148
5.7.5 Anbindung an Configuration Admin Service 148
5.7.6 Kompositionen 148
5.7.7 Beispiel für Metadaten 149
6 Quo Vadis 151
6.1 Ausblicke auf das OSGi Release 4.2 151
6.2 Softwareentwicklung mit OSGi 153
6.2.1 Entwicklungsumgebung 153
6.2.2 Testverfahren 153
6.2.3 Auslieferung und Betrieb 154
A Anhang 155
A.1 Bibliographie 155
A.2 Linksammlung 156
Stichwortverzeichnis 161

1 Einleitung

Aufgrund des Fehlens eines Modulkonzepts in Java (Kap. 1.3) und den sich daraus ergebenden Problemen sind in letzter Zeit die Schlagworte *Modularisierung* und *Komponentenmodell* stark in den Fokus gerückt.

Dieses Buch möchte Ihnen kurz die Probleme der fehlenden Modularisierung erläutern und die derzeitige De-facto-Lösung unter Java vorstellen: OSGi (Open Services Gateway initiative).

Die OSGi-Service-Plattform ist eine Spezifikation, die Modularität in Java definiert. Es existieren mittlerweile einige Implementierungen, auch aus der Open-Source-Welt. Die bekanntesten sind sicherlich Equinox, der Kern von Eclipse, und das Apache-Projekt Felix.

Der Schwerpunkt des Buches liegt in der konzeptionellen Darstellung. Pro Kapitel wird in jeweils 2 Detailtiefen ein Thema behandelt. Die erste, für Einsteiger geeignete Abstraktionsebene beschäftigt sich mit den grundlegenden Zusammenhängen.

Zu jedem Thema werden aber auch bestimmte Aspekte detaillierter untersucht. Dazu findet sich ein kurzer, auf diesen Aspekt fokussierter Abschnitt. Diese Abschnitte setzen in der Regel Grundlagenwissen voraus und richten sich vor allem an erfahrene Leser. Sie sind als „Expertenwissen" gekennzeichnet.

Das kompakte Format legt es nahe, den Schwerpunkt auf die Darstellung der Zusammenhänge zu legen. Daher ist es nicht als Einführung in die Programmierung von OSGi geeignet. Hierzu gibt es bereits gute und umfangreiche Bücher[1] oder aber freie Tutorien (einen Überblick finden Sie unter *http://www.osgi.org/About/HowOSGi*).

Dieses Buch bietet Ihnen hingegen die Chance, die OSGi-Konzepte und ihre Zusammenhänge unmittelbar, komprimiert und nicht durch Programmierbeispiele verwässert zu überblicken und ist ideal, um das für OSGi notwendige Hintergrundwissen aufzubauen.

---

1 Gerd Wütherich et al.: *Die OSGi Service Platform – Eine Einführung mit Eclipse Equinox*; Richard Hall et al., *Osgi in Action: Creating Modular Applications in Java*; Neil Bartlett: *OSGi In Practice* (*http://www.sei.cmu.edu/staff/kcw/00tr008.pdf*).

1.1 Modularität

Komplexe Softwaresysteme sind nur mit dem Prinzip „Teile und herrsche“ in den Griff zu bekommen. Das setzt die Möglichkeit voraus, die Software in weitestgehend autonome Softwareeinheiten (Module) zu strukturieren und diese in einer adäquaten Laufzeitumgebung zu installieren.

Hinter dem Konzept Modularität/Modularisierung steht das Bestreben, Softwareeinheiten (Module/Komponenten) unabhängig vom Systemkontext verändern und pflegen zu können. Auswirkungen der Veränderung auf andere Softwareeinheiten sollten begrenzt sein und keine ungewollten Nebeneffekte auslösen (*http://www.cs.bath.ac.uk/~jap/MATH0015/glossary.html*).

Aus dem Konzept der Modularität ergeben sich einige Anforderungen an Komponenten. Die einzelnen Forderungen sind nicht neu, machen aber in dieser Zusammenstellung den Kern von Komponentenmodellen aus *(http://www.sei.cmu.edu/library/abstracts/reports/00tr008.cfm)*.

Die Forderungen betreffen einerseits die logische Struktur der Software und andererseits das Deployment, also die Installation der physischen Softwareeinheiten.

1.1.1 Komponentenmodell

Komponenten müssen klar definierte Abgrenzungen zu anderen Komponenten haben. Dahinter verbergen sich die Implementierungsdetails. Eine Komponente veröffentlicht Schnittstellen, über die auf sie zugegriffen werden kann. Diese Schnittstellen definieren den Vertrag zwischen der Komponente einerseits und dem Komponentenframework und weiteren Komponenten andererseits.

Die Abhängigkeiten zu anderen Komponenten werden durch die Komponente explizit veröffentlicht. Sie werden nicht durch die Komponente selber, sondern durch das Komponentenframework aufgelöst und überwacht (zum Zeitpunkt der Installation).

Die Laufzeitumgebung von Komponenten (Komponentenframework) (de-)installiert Komponenten und ist dafür verantwortlich, die Abhängigkeiten zwischen den Komponenten aufzulösen und zu überwachen. Komponenten müssen daher nicht wissen, wie und wo abhängige Komponenten zu finden sind.

Komponenten sind innerhalb des Komponentenframeworks identifizierbar. Sie müssen Kriterien angeben können, nach welchen sie auszuwählen sind (z. B. Name, Version, Eigenschaften, ...). In einem Komponentenframework können unterschiedliche Versionen einer Komponente installiert werden.

Eine Komponente ist eine unabhängige und individuell gepackte Softwareeinheit. Sie kann unabhängig von anderen Komponenten (de-)installiert werden. Eine der wichtigsten und wohl auch komplexesten Aufgaben des Frameworks ist es, auf Auswirkungen einer (De-)Installation zu reagieren.

Wird eine Komponente installiert, so müssen ihre Abhängigkeiten zu anderen Komponenten aufgelöst werden.

- Wie soll das Framework reagieren, wenn diese Abhängigkeiten nicht aufzulösen sind, weil z. B. die erforderlichen Komponenten nicht installiert sind?
- Wie soll das Framework reagieren, wenn durch die Deinstallation einer Komponente die Abhängigkeiten zu anderen installierten Komponenten verletzt werden?
- Wie steht es in Java mit der Unterstützung der Modularitätskonzepte? Die Aufmerksamkeit, die OSGi entgegengebracht wird, lässt vermuten, dass dort einige Mängel zu finden sind.

Mithilfe eines Komponentenmodells lässt sich Modularisierung umsetzen.

1.1.2 Schwächen von Java

Auf Ebene der logischen Strukturierung bietet Java Klassen, Packages und Sichtbarkeiten. Sichtbarkeiten unterstützen neben Klassen und Packages keine weiteren Strukturierungsgrößen.

Ein Package ist keine aggregierende Einheit. Subpackages eines Packages haben keinerlei Beziehungen zu ihren Eltern-Packages, sodass ein Package keine Gesamtsicht auf den Baum aller Inhalte bietet. Daher ist das Package als logisches Modul ungeeignet. Das Konzept des Moduls als logische Strukturierung fehlt in Java.

Java bietet als Auslieferungseinheit im Wesentlichen das JAR-File. Dieses grenzt sich aber nicht von anderen JAR-Files im Classpath ab, sodass das JAR-File nicht ohne weiteres zur Unterstützung eines Modulkonzepts taugt. Insbesondere existiert keine Möglichkeit, Klassen/Packages zu versionieren und mehre Versionen derselben Klasse im System verfügbar zu halten.

In einer Java-Anwendung unterliegen alle JAR-Files einer für alle Klassen gemeinsamen Classloading-Hierarchie. JAR-Files sind nicht mehr voneinander abgegrenzt.

Sie bieten auch keine Möglichkeit, Interfaces zu ihrer Nutzung anzugeben. Klassen werden in der gesamten Classloading-Hierarchie publiziert angegeben. Somit bieten JAR-Files keine Basis, Modulkonzepte umzusetzen. Auch auf der physischen Ebene fehlt das Konzept eines Moduls.

Mit JEE kommen weitere Auslieferungsformate (WAR, EAR), die den Modulgedanken deutlich weiter unterstützen. Dies sind aber keine allgemeinen Konzepte, sondern sie sind auf spezielle Typen von Anwendungen und Laufzeitumgebungen zugeschnitten.

Die angesprochenen Kritikpunkte können umgangen werden und es ist möglich, auch in Java Modularisierung einzuführen (mit OSGi!), aber es wird derzeit noch kein entsprechendes Konzept durch Java selbst unterstützt.[2]

1.2 Was ist OSGi?

Nach den Erläuterungen aus Kapitel 1.1 ist es naheliegend, dass die OSGi-Technologie eine Lösung zur Modularisierung in Java anbietet.

Module als installierbare Softwareeinheiten heißen in OSGi *Bundle* und erfüllen alle Forderungen an ein Modulkonzept, das mittels eines Komponentenmodells umgesetzt wird. Dort ist beschrieben, wie ein Bundle definiert wird, welche Schnittstellen es veröffentlicht und welche Schnittstellen anderer Bundles es importieren möchte.

OSGi unterstützt darüber hinaus eine Dynamik von Modulen, die es erlaubt, Bundles zur Laufzeit zu installieren, zu ersetzen oder zu deinstallieren (Kap. 2.1 und 2.2). Diese Dynamik wird mit dem Begriff des *Bundle Life Cycle* umschrieben.

Die spezifizierte Laufzeitumgebung (das OSGi-Framework) setzt das Komponentenmodell um und verwaltet eine komplexe Sicherheitsarchitektur (Kap. 2.5).

2 Eine detaillierte Darstellung der Modularisierungsprobleme in Java finden Sie in: Neil Bartlett: *OSGi In Practice*.

Neben der Modularisierung bietet OSGi eine serviceorientierte Infrastruktur. Services registrieren sich zentral und bieten Interfaces an. Zusätzlich können sie sich durch so genannte Service-Properties qualifizieren. Nach registrierten Services kann gesucht, und diese können schließlich genutzt werden. Es kann mehrere Anbieter desselben Interfaces geben.

Dieses Modell entkoppelt die Bereitstellung und Nutzung von Services. OSGi stellt Programmiermodelle für den Umgang mit Services bereit. Entsprechend der Dynamik von Bundles sind auch Services hochgradig dynamisch. Programmiermodelle müssen darauf achten, dass Services jederzeit registriert oder deregistriert werden können (Kap. 2 und Kap. 3). Auf Basis dieses Modells existieren Komponentenmodelle, die die Zusammenarbeit zwischen Services regeln. Insbesondere kümmern sie sich um die Dynamik der Services (Kap. 5).

Diese Konzepte werden in einer OSGi-Infrastruktur spezifiziert, die alle Konzepte umfasst. Bundles werden zur Wiederverwendung entwickelt. Daher ist typischerweise zum Entwicklungszeitpunkt nicht klar, auf welchem OSGi-Framework sie eingesetzt werden. Andererseits haben Bundles aber gewisse Anforderungen an die Version und den Umfang der *JRE (Java Runtime Environment)*. OSGi definiert daher unterschiedliche Profile (*Execution Environments*), die durch ein OSGi-Framework unterstützt werden können. Das Bundle spezifiziert, welches *Execution Environment* es benötigt, und das OSGi-Framework entscheidet bei der Installation, ob das Bundle in der aktuellen Umgebung ausführbar ist.

Abbildung 1.1 zeigt das Zusammenspiel der unterschiedlichen Layer. Einerseits bauen die Layer aufeinander auf, andererseits stellen sie ihre Funktionalität aber der Entwicklung der Bundles zur Verfügung. Ein Bundle kann sich an jeden Layer wenden und dessen Funktionalität nutzen.Bundles können also die OSGi-Mechanismen unterschiedlicher Ebenen nutzen.

Diese Mechanismen sind in der *OSGi Core Specification festgelegt*. Jedes OSGi-konforme Framework muss diese Spezifikation vollständig umsetzen (*http://www.osgi.org/Release4/Download*). Darüber hinaus spezifiziert OSGi optionale Infrastrukturservices wie Logging, HTTP-Anbindung etc. (Kap. 4). Diese werden in der *OSGi Service Compendium Specification* definiert. Sie stützen sich auf die OSGi-Infrastruktur und sind bei konformer Implementierung auf jedem OSGi-Framework einzusetzen.

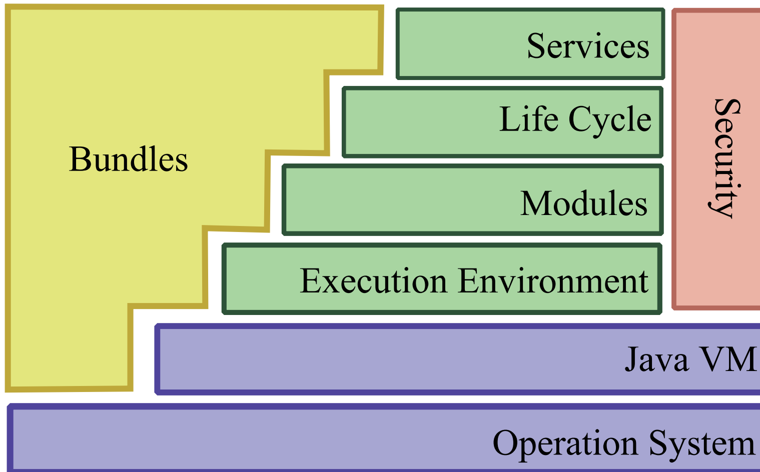

Abbildung 1.1: Überblick über die OSGi-Architektur

OSGi bietet also Komponentenmodelle auf zwei Ebenen. Das erste Komponentenmodell gewährleistet die Modularisierung und beschreibt den Vertrag zwischen Bundles bzw. zwischen Bundles und dem OSGi-Framework. Auf der zweiten Ebene finden sich die Komponentenmodelle zwischen Services. Derzeit ist eines standardisiert (Declarative Services, Kap. 5.1), weitere sind auf dem Weg dahin (Spring Dynamic Modules, Kap. 5.2). Darüber hinaus gibt es noch weitere Ansätze (Kap. 5.3).

1.2.1 Historischer Abriss

OSGi ist eine Standardisierungsorganisation, gegründet im Jahr 1999. Begonnen hat OSGi auf dem Gebiet der *Mobile Automotive Applications*, also Anwendungen für mobile Endgeräte. An einigen Stellen kann OSGi diese Herkunft nicht verbergen, insbesondere bei der Definition einiger Infrastrukturservices.

Mittlerweile engagiert OSGi sich auch im *Java-Middleware*-Bereich. Die OSGi-Spezifikation hat die Version 4.1 erreicht, zum Zeitpunkt der Veröffentlichung dieses Buchs wird an Version 4.2 gearbeitet.

1.3 Weitere Komponentenmodelle der Java-Welt

1.3.1 Java Module System

Mit Java 7 beabsichtigt Sun, ein zu OSGi konkurrierendes Modulkonzept einzuführen. Dazu existiert der JSR 294, der alle in Kapitel 1.1 aufgestellten Anforderungen an ein Komponentenmodell erfüllt.

Die Zukunft von OSGi im Zusammenhang mit der Weiterentwicklung von Java an sich wird zum Zeitpunkt der Drucklegung dieses Buches heiß diskutiert. Zum einen gibt es Bestrebungen, die Java Runtime zu modularisieren (*http://blogs.sun.com/mr/entry/jigsaw*). Zum anderen soll es aber auch für Entwickler möglich werden, die eigene Software nativ in einer modularen Bauweise zu entwickeln. (JSR277, JSR291, JSR294).[3] Hierbei hat sich Sun allerdings von JSR 277 (Integration von OSGi in Java) entfernt und verfolgt nun priorisiert JSR 294 (Modularität in Java), der in Java 7 zum Einsatz kommen soll. Eine vollständige Integration von OSGi in den Java-Kern wird bis weit nach Erscheinen von Java 7 warten müssen.

Des Weiteren wird im Rahmen des JSR 291 an einem nativen Komponentensystem für Java gearbeitet. Die OSGi Alliance hat bereits eine Referenzimplementierung für JSR 291 vorgelegt.[4]

3 JSR 277: Java Module System *http://jcp.org/en/jsr/detail?id=277;* JSR 291: Dynamic Component Support for Java SE *http://jcp.org/en/jsr/detail?id=291;* JSR 294: Improved Modularity Support in the Java Programming Language *http://jcp.org/en/jsr/detail?id=294*

4 OSGi Alliance JSR 291 Reference Implementation *http://www2.osgi.org/Download/File?url=/download/jsr291-ri-final.zip*

2 OSGi-Konzepte

2.1 Bundles – Module in OSGi

Nachdem in Kapitel 1.2 beschrieben wurde, welche Forderungen an Komponenten gestellt werden, wollen wir nun untersuchen, mit welchen Strategien und Verfahren das OSGi-Framework diese umsetzt.

OSGi-Komponenten (Bundles) sind als JAR-Dateien gepackt. Sie enthalten *class-files* oder andere aus JAR-Files bekannte Ressourcen. Es können auch weitere JAR-Dateien mitgepackt werden, die den Classpath des Bundles erweitern. Damit erinnert ein Bundle in seiner Struktur an ein WAR (vgl. *WEB-INF/lib*) oder RAR.

Die Datei *META-INF/MANIFEST.MF* (Manifest-Datei siehe *JARMF*) des Bundles beschreibt den Vertrag mit anderen Bundles und dem OSGi-Framework. Diese Form der Abhängigkeit zwischen Bundles bezieht sich auf die Ressourcen, die ein Bundle von anderen Bundles importiert (*Importer*) oder die es für andere Bundles bereitstellt (*Exporter*). Exportierte Packages können durch andere Bundles importiert werden, nicht exportierte gehören dem Bundle privat.

OSGi kennt zwei Verfahren, wie ein Bundle Ressourcen eines anderen Bundles nutzen kann (*Bundle Import*). Zum einen kann ein Bundle explizit Bundles anfordern (*Require Bundle*) oder aber zum anderen Packages verlangen, ohne das exportierende Bundle zu kennen (*Import Package*).

```
Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorld
Bundle-Version: 1.0
Bundle-SymbolicName:sample.HelloWorld
Bundle-ClassPath: .,
    target/dependency/junit-3.8.1.jar
Bundle-Activator:sample.internal.Activator
```

Listing 2.1: Beispiel einer Manifest-Datei