HANSER

Leseprobe

Bernd Weber, Patrick Baumgartner, Oliver Braun

OSGi für Praktiker

Prinzipien, Werkzeuge und praktische Anleitungen auf dem Weg zur "kleinen SOA"

ISBN: 978-3-446-42094-6

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42094-6

sowie im Buchhandel.

Kapitel 6

Migration – Java-Archive zu Bundles erweitern

Das Ziel dieses Kapitels ist die Migration eines konventionellen Java-Archivs zu einem OSGi-Bundle. Dazu wird beispielhaft die freie iText-Bibliothek herangezogen, analysiert und mit OSGi-Metadaten angereichert, um sie anschließend in einer OSGi-Plattform als Bundle installieren und von anderen Bundles aus verwenden zu können. Das Beispiel soll zeigen, wie man bei einer solchen Migration vorgeht, und als Leitfaden für die Migration eigener oder anderer freier Java-Archive zu OSGi-Bundles dienen. Es werden verschiedene Wege aufgezeigt, um den unterschiedlichen Konstellationen einer Migration gerecht zu werden.

6.1 Analyse des Java-Archivs

Die Bibliothek iText[1] von Lowagie[2] ist ein wunderbares Open Source-Werkzeug zur programmatischen Erzeugung von PDF-Dateien. Neben deren inhaltlicher Erstellung in all ihren Facetten bietet es die Möglichkeit, PDF-Dateien aufzuteilen oder zusammenzuführen, sie zu verschlüsseln und daraus ausführbare Präsentationsdateien zu erstellen. iText liegt zum Zeitpunkt der Drucklegung dieses Buches in Version 5.0.0 als Standard Java-Archiv ohne OSGi-Metadaten vor; eine ältere Version 2.0.9 wurde von SpringSource bereits zu einem OSGi-Bundle migriert und in deren Enterprise Bundle Repository eingestellt (siehe Kapitel 3.4.3 auf Seite 62). Wer allerdings die aktuelle Version benötigt, muss selbst Hand anlegen und die Bibliothek um die notwendigen Metadaten erweitern.

[1] http://itextpdf.com
[2] http://www.lowagie.com

6.1.1 Ermitteln von Abhängigkeiten mit Hilfe von bnd

Soll ein Java-Archiv zu einem OSGi-Bundle erweitert werden, analysiert man zunächst die bestehenden Abhängigkeiten. Dafür betrachtet man alle von den Klassen des Archivs importierten Packages. Hierbei werden interne und externe Verweise sichtbar, Letztere unterteilen sich zudem in notwendige und optionale Abhängigkeiten.

Das bnd-Tool unterstützt die Analyse von Abhängigkeiten bestehender Java-Archive sehr gut. Aufgerufen mit dem Pfad der zu analysierenden Datei schaltet bnd anhand der Dateiendung in den Subbefehl `print` und zeigt detaillierte Informationen zu dem Java-Archiv:

Listing 6.1: Ausgabe von bnd print (Auszug)

```
$ bnd print iText-5.0.0.jar
[MANIFEST iText-5.0.0.jar]
Ant-Version               Apache Ant 1.7.0
Created-By                1.5.0_15-b04 (Sun Microsystems
    Inc.)
Manifest-Version          1.0

[IMPEXP]

[USES]
com.itextpdf.text         com.itextpdf.text.
    error_messages
                          com.itextpdf.text.factories
                          com.itextpdf.text.html
                          com.itextpdf.text.pdf
[...]
com.itextpdf.text.pdf     com.itextpdf.text
                          [...]
                          javax.crypto
                          javax.imageio
                          [...]
                          javax.xml.parsers
                          org.bouncycastle.asn1
                          [...]
                          org.w3c.dom
                          org.xml.sax
[...]

One error
1 : Unresolved references to [javax.crypto, javax.imageio
    , javax.imageio.metadata, javax.imageio.plugins.jpeg,
    javax.imageio.stream, javax.xml.parsers, org.
    bouncycastle.asn1, org.bouncycastle.asn1.cmp,  org.
    bouncycastle.asn1.cms, org.bouncycastle.asn1.ocsp, org
    .bouncycastle.asn1.pkcs, org.bouncycastle.asn1.tsp,
    org.bouncycastle.asn1.x509, org.bouncycastle.cms, org.
```

```
bouncycastle.crypto, org.bouncycastle.crypto.engines,
org.bouncycastle.crypto.modes, org.bouncycastle.crypto
.paddings, org.bouncycastle.crypto.params, org.
bouncycastle.jce.provider, org.bouncycastle.ocsp, org.
bouncycastle.tsp,org.w3c.dom, org.xml.sax] by class(es
) on the Bundle-Classpath[...]
```

6.1.2 Analyse der Abhängigkeiten

Die in Abschnitt 6.1.1 gezeigte Ausgabe von bnd führt zunächst unter `[MANIFEST iText-5.0.0.jar]` den Inhalt der Manifest-Datei mit drei weitgehend uninteressanten Einträgen auf, der anschließende Abschnitt `[IMPEXP]` zur Anzeige der im- und exportierten Packages ist leer. Mit `[USES]` beginnt eine lange Liste der verwendeten Packages, die zumeist eigene, also in der JAR-Datei befindliche Packages enthält. Der für das Funktionieren des Bundles relevante Teil ist die Fehlermeldung am Ende der Ausgabe, beginnend mit `Unresolved references to`. Hier zeigt bnd Bezüge nach außen auf, die nicht aufgelöst werden können. Diese externen Verweise lassen sich unterteilen in solche, die wie `javax.crypto` und `org.w3c.dom` Teil einer bestimmten Java-Edition sind, und solche wie `org.bouncycastle.asn1`, die nicht darunter fallen.

6.1.3 Laufzeitumgebung

Nun bringt eine OSGi-Plattform nicht automatisch eine Standard oder gar Enterprise Edition von Java mit – schließlich liegt der Ursprung von OSGi im Bereich der Kleingeräte weitab der Java Application Server. Welche verfügbaren Klassen und Methoden eine OSGi-Plattform von Haus aus aufweist, lässt sich über die Methode `String getProperty(String name)` des *BundleContext*s erfahren, aufgerufen mit dem vordefinierten Wert *org.osgi.framework.Constants.FRAMEWORK_EXECUTIONENVIRONMENT*. Der resultierende String enthält eine Liste aller von der Plattform bereitgehaltenen Laufzeitumgebungen (Execution Environments), die den darauf installierten Bundles zur Verfügung stehen.

Ein Bundle, das mehr als die für eine OSGi-Plattform notwendige Minimalmenge an Klassen und Methoden benötigt, kann dies über den Manifest-Eintrag *Bundle-RequiredExecutionEnvironment* angeben, der eine oder mehrere Execution Environments enthalten kann. Die möglichen Eintragungen sind im Kasten *Execution Environments* auf Seite 110 dargestellt. Beim Auflösen (Resolving) eines Bundles gleicht das Framework seine eigenen Execution Environments mit den vom Bundle geforderten ab. Gibt es einen Treffer, kann das Bundle aufgelöst werden, seine Anforderungen an die Laufzeitumgebung sind erfüllt. Ansonsten bleibt das Bundle im Zustand *installed* und kann nicht verwendet werden.

Execution Environments

Ein Execution Environment definiert die mindestens vorhandene Menge an Klassen und Methoden, die einem Bundle zur Laufzeit zur Verfügung steht. Umgekehrt definiert ein Bundle über den Header *Bundle-RequiredExecutionEnvironment* alle Laufzeitumgebungen, unter denen es ausführbar ist. Diese Laufzeitumgebungen werden als kommaseparierte Liste angegeben, z.B. *Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0,OSGi/Minimum-1.1*.

In Kapitel 3.3 der OSGi Core Specification wird festgelegt, wie ein Execution Environment definiert und benannt sein soll. Der minimale Umfang für OSGi-Frameworks ist in Kapitel 999 des OSGi Service Compendiums definiert und kann als JAR von der OSGi Alliance heruntergeladen werden[3]. Neben diesen beiden OSGi-spezifischen Execution Environments gibt es einige weitere aus dem Bereich der Klein- und Mobilgeräte (CDC und CLDC), die als Java Specification Requests (z.B. JSR-218 und JSR-219) in den Java Community Process eingebracht wurden. Zudem sind alle Standard Editionen von Java seit Version 1.2 als Execution Environment definiert. Die gebräuchlichsten Definitionen finden sich in der Tabelle 3.2 des oben genannten Core-Kapitels und sind nachfolgend abgedruckt:

OSGi/Minimum-1.2	Das Minimum zur Implementierung eines OSGi Frameworks
CDC-1.1/Foundation-1.1	Entspricht dem J2ME Foundation Profile
J2SE-1.2	Java 2 SE 1.2.x
J2SE-1.3	Java 2 SE 1.3.x
J2SE-1.4	Java 2 SE 1.4.x
J2SE-1.5	Java 2 SE 1.5.x
JavaSE-1.6	Java SE 1.6.x
CDC-1.1/PersonalBasis-1.1	J2ME Personal Basis Profile
CDC-1.1/PersonalJava-1.1	J2ME Personal Java Profile

Doch zurück zur iText-Bibliothek und deren Migration zu einem OSGi-Bundle. Die grundlegenden Anforderungen an ein Bundle müssen erfüllt sein. Also müssen zumindest der *Bundle-SymbolicName* und die *Bundle-Version* definiert werden. Außerdem müssen die Packages von iText festgelegt werden, die anderen Bundles zur Verfügung stehen sollen, damit iText auch in Bundle-Form weiterhin als Bibliothek genutzt werden kann. Schließlich gilt es, die vorhandenen Abhängigkeiten zu Drittbibliotheken geeignet in der Migration zu berücksichtigen. Das kann auf verschiedene Weise umgesetzt werden. Dieses Buch beleuchtet die Varianten

[3] http://www.osgi.org/Download/File?url=/download/r4v42/ee.minimum.jar
bzw. http://www.osgi.org/Download/File?url=/download/r4v42/ee.foundation.jar

- Standardmigration,
- Beistellungsmigration und
- Transitive Migration.

6.2 Standardmigration

Die Standardmigration ist – nomen est omen – die häufigste Form der Migration. Sie geht davon aus, dass die Abhängigkeiten zu Drittbibliotheken zumeist optionaler Natur sind und beispielsweise die Einbindung in ein bestimmtes Framework zum Ziel haben. Der wichtigste Aspekt dieser Vorgehensweise ist eine möglichst schnelle Migration des Java-Archivs in ein Bundle, sei es explizit mittels bnd oder Maven oder *on the fly* durch Werkzeuge wie `pax-wrap-jar` von PAX Construct.

6.2.1 Festlegen der Metadaten

Auch eine schnelle Migration kommt nicht umhin, einige Metadaten für das Bundle festlegen zu müssen. Während der *Bundle-SymbolicName* und die *Bundle-Version* noch aus dem Bibliotheksnamen und ihrer Version übernommen werden könnten, sind die übrigen benötigten Header nicht mehr ganz so trivial zu ermitteln. Und auch der *Bundle-SymbolicName* bedarf eines gewissen Feinschliffs.

Bundle-ID

Ein Bundle benötigt stets die Angabe eines symbolischen Namens und einer Version. Sie bilden zusammen die Bundle-ID und müssen innerhalb einer OSGi-Plattform eindeutig sein. Um hier keine Verwechslung zuzulassen und zugleich den Urheber der Migration aufzunehmen, hat es sich eingebürgert, dem ursprünglichen Package-Namen die Maven-GroupId des Migrationsurhebers voranzustellen und als den symbolischen Namen zu verwenden. Als Version wird die Bibliotheksversion übernommen. Damit sind die ersten zwei Manifest-Einträge definiert:

```
Bundle-SymbolicName: biz.gossipmonger.com.itextpdf.text
Bundle-Version: 5.0.0
```

Execution Environment

iText referenziert eine Menge von externen Packages, die teilweise dem Java Standard entsprechen. Die von iText benötigten Java Packages sind:

- javax.crypto
- javax.imageio

- javax.imageio.metadata
- javax.imageio.plugins.jpeg
- javax.imageio.stream
- javax.xml.parsers
- org.w3c.dom
- org.xml.sax

und stehen seit Java 1.4.2 in der Standard Edition zur Verfügung. Damit benötigt iText als Mindestvoraussetzung diese Laufzeitumgebung, kann aber auch mit späteren Versionen betrieben werden. Um dies dem Framework kundzutun, wird folgender Manifest-Eintrag definiert:

```
Bundle-RequiredExecutionEnvironment: J2SE-1.4,J2SE-1.5,
   JavaSE-1.6
```

Importieren von Packages

Der Verweis auf `org.bouncycastle.asn1` und auf weitere Packages mit dem Präfix `org.bouncycastle` findet sich in keinem der vordefinierten Execution Environments. Bouncy Castle[4] ist ein Provider für die Java Cryptography Extension (JCE) und wird von iText unter anderem zur Verschlüsselung oder Signierung von PDF-Dateien herangezogen. Hierbei handelt es sich um eine optionale Abhängigkeit, die lediglich dann zum Tragen kommt, wenn tatsächlich verschlüsselt oder signiert wird. Für die Zwecke des Buches wird auf diese Fähigkeiten verzichtet, die Abhängigkeit zu BouncyCastle sollte folglich als optional gekennzeichnet werden und nicht dazu führen, dass ein BouncyCastle-Bundle in der OSGi-Plattform installiert sein *muss*, um das iText-Bundle auflösen zu können.

Im Bundle-Header *Import-Package* angegebene Packages sind standardmäßig mit dem Attribut `resolution:=mandatory` versehen. Damit müssen diese Packages zur Laufzeit bereitstehen, um das Bundle auflösen zu können. Für eine optionale Auflösung kann dieses Attribut mit dem Wert `optional` versehen werden. Dieses Attribut kann sowohl einem einzelnen Package als auch einer Menge von Packages mit demselben Präfix (Package-Wildcard) beigestellt sein. Damit steht der nächste Manifest-Eintrag fest, der die zu importierenden BouncyCastle-Packages als `optional`, alle übrigen als `mandatory` definiert:

```
Import-Package: org.bouncycastle**;resolution:=optional,*
```

Behandlung interner Abhängigkeiten

Die Verweise auf in der Bibliothek selbst befindliche Packages sind interne Abhängigkeiten und müssen hinsichtlich der OSGi-Metadaten prinzipiell nicht ge-

[4] http://www.bouncycastle.org/

sondert betrachtet werden. Zwar wird in der OSGi-Spezifikation empfohlen, solche Selbstreferenzierungen sowohl in die Menge der exportierten als auch der importierten Packages aufzunehmen, um beim Auflösen der Abhängigkeiten (Resolving) der beteiligten Bundles für Schnittstellen und gemeinsame Typen denselben Classloader verwenden zu können. Doch im Falle von iText spielt dies keine Rolle, da diese Bibliothek nicht in Form von OSGi-Diensten über Schnittstellen genutzt werden kann, sondern lediglich statisch gebunden wird. Zudem ist die Bibliothek nicht darauf vorbereitet, mit anderen Package-Versionen als den selbst mitgebrachten umgehen zu können; es wäre daher ein Risiko, diesen OSGi-Mechanismus auf iText anzuwenden.

Exportieren von Packages

Um die iText-Bibliothek als Bundle sinnvoll einsetzen zu können, müssen deren öffentliche Packages exportiert werden. Nur so können andere Bundles deren Klassen verwenden und einen Nutzen aus dem Bundle ziehen. Der Einfachheit halber wird vorausgesetzt, dass sämtliche Packages unterhalb von `com.itextpdf.text` öffentlicher Natur sind. Damit genügt die Angabe einer doppelten Wildcard für den Export des Haupt-Packages und seiner Unter-Packages:

```
Export-Package: com.itextpdf.text**
```

6.2.2 Durchführen der Migration

Um nun die Bibliothek zu migrieren, wird eine bnd-Steuerdatei namens *itext.bnd* erstellt mit allen zuvor festgelegten Manifest-Einträgen. Diese Steuerdatei wird noch ergänzt um den Pfad der zu migrierenden JAR-Datei und um einen sprechenden Namen. Das Ergebnis ist folgende Steuerdatei:

Listing 6.2: itext.bnd

```
bsn: biz.gossipmonger.com.itextpdf.text
version: 5.0.0
Bundle-RequiredExecutionEnvironment: J2SE-1.4,J2SE-1.5,
   JavaSE-1.6
Import-Package: org.bouncycastle**;resolution:=optional,*
Export-Package: com.itextpdf.text**
Bundle-SymbolicName: ${bsn}
Bundle-Version: ${version}
Bundle-Name: iText PDF Library (OSGified)
-classpath: iText-5.0.0.jar
-output: ${bsn}-${version}.jar
```

Die Migration wird mit dem Befehl `bnd itext.bnd` durchgeführt. bnd erkennt anhand der Endung, dass es sich um eine Steuerdatei handelt, und führt damit automatisch den Unterbefehl `build` aus.

6.2.3 Bereitstellung

Der Befehl erstellt die Datei `biz.gossipmonger.com.itextpdf.text-5.-0.0.jar`, welche nun noch in das lokale Maven-Repository installiert werden muss, damit sie in den Erstellungslauf des Gesamtprojekts integriert werden kann. Hierzu wird der Maven-Befehl `mvn install:install-file` verwendet, dem die Metadaten groupId, artifactId, version, packaging und file beigestellt werden. Dabei enthalten groupId und artifactId sowohl den Bundle-Bereitsteller als auch den Ersteller der Bibliothek selbst und lassen damit keine Verwechslung mit der nicht migrierten Variante zu. Die Version bleibt bei 5.0.0, da sich an der Bibliothek außer den Metadaten nichts geändert hat. Der Installationsbefehl für die Standardmigration lautet:

Listing 6.3: Installation der migrierten Bibliothek

```
$ mvn install:install-file \
 -DgroupId=biz.gossipmonger.com.itextpdf \
 -DartifactId=biz.gossipmonger.com.itextpdf.itext \
 -Dversion=5.0.0 \
 -Dpackaging=jar \
 -Dfile=biz.gossipmonger.com.itextpdf.text-5.0.0.jar
```

6.3 Beistellungsmigration

In der Beistellungsmigration nimmt das Zielarchiv, also das resultierende OSGi-Bundle, das zu migrierende Java-Archiv sowie alle Bibliotheken auf, die zur Auflösung seiner und aller weiteren transitiven Abhängigkeiten benötigt werden.

Die in der Standardmigration geschilderte optionale Behandlung der Abhängigkeiten ist eine von drei Möglichkeiten, Abhängigkeiten eines Bundles aufzulösen. Da sie die schnellste Form der Migration ist, wird sie häufig angewandt und dabei gerne übersehen, dass es durchaus sinnvoll ist, über alternative Migrationsstrategien nachzudenken und diesen möglicherweise sogar den Vorzug zu geben.

Ein Bundle ist – wie ein Java-Archiv oder eine sonstige Zip-Datei – ein Container, der Einträge in hierarchischer Form enthält. Solche Einträge können analog zum Java-Klassenpfad in den Manifest-Header *Bundle-ClassPath*[5] aufgenommen werden und stehen dann in der dort angegebenen Reihenfolge dem Classloader des Bundles zur Verfügung. Es empfiehlt sich, die mitverpackten Bibliotheken nicht im Stammverzeichnis, sondern an einer dafür vorgesehenen Stelle abzulegen, beispielsweise in einem Verzeichnis `lib` unterhalb des Stammverzeichnisses des Bundles.

Das Stammverzeichnis eines Bundles wird per Definition über den Pfad „/“ oder dessen Synonym „.“ angesprochen und ist stets der Beginn der Bundle-internen

[5] Bundle Class Path, Kap. 3.8.1 der OSGi Core Specification

Pfadauflösung, sozusagen das *current working directory* des Bundles. Ist kein *Bundle-ClassPath* angegeben, verwendet der Bundle-Classloader das Stammverzeichnis. Dieses sollte in einem spezifischen *Bundle-ClassPath* stets an erster Stelle angegeben werden, gefolgt von den übrigen gewünschten Container-Einträgen.

Um nun die für das Beispiel iText benötigten Abhängigkeiten in das Bundle aufzunehmen und die Verweise zu definieren, muss die bnd-Steuerdatei der Standardmigration an einigen Stellen geändert werden. Von den verschiedenen Varianten, in denen die BouncyCastle-Archive angeboten werden, wird jeweils diejenige mit dem Hinweis auf JDK14-Kompatibilität verwendet, z.B. `bcprov-jdk14-144.jar`, die das Ausführen in einer J2SE 1.4.x-Umgebung und höher erlaubt.

Die benötigten Packages sind auf mehrere BouncyCastle-Java-Archive aufgeteilt, die beizustellenden Archive sind `bcprov-jdk14-144.jar`, `bcmail-jdk14-144.jar` und `bctsp-jdk14-144.jar`. Sie sollen im Container-Verzeichnis `/lib` abgelegt werden. Der Klassenpfad des Bundles muss entsprechend definiert werden und erhält zudem als führenden Eintrag das Stammverzeichnis:

```
Bundle-Classpath: ., lib/bcprov-jdk14-144.jar, lib/bcmail
    -jdk14-144.jar, lib/bctsp-jdk14-144.jar
```

6.3.1 Aufnahme von Ressourcen

Damit die Java-Archive den Weg in das Bundle und dort in das richtige Verzeichnis finden, werden sie über den bnd-spezifischen Bundle-Header *Include-Resource* definiert. Dieser weist bnd an, Ressourcen in das Zielarchiv (Bundle) zu kopieren. Für die Aufnahme der BouncyCastle-Archive benötigt man die Form einer Zuweisung. Der Teil links vom Gleichheitszeichen bestimmt den Zielort im Bundle, während der rechte Teil den Fundort der Ressource im Dateisystem relativ zum aktuellen Verzeichnis darstellt. Der Bundle-Zielort muss dabei denselben relativen Pfad aufweisen, wie er oben im *Bundle-ClassPath* eingetragen ist. bnd legt beim Kopieren das Zielverzeichnis `lib` selbständig an. Das Ergebnis ist ein weiterer Eintrag in der Steuerdatei:

```
Include-Resource: lib/bcprov-jdk14-144.jar=bcprov-jdk14
    -144.jar, lib/bcmail-jdk14-144.jar=bcmail-jdk14-144.
    jar, lib/bctsp-jdk14-144.jar=bctsp-jdk14-144.jar
```

Details zu *Include-Resource*

Die nachfolgenden Ausführungen sind für die Umsetzung des Beispiels unerheblich, können aber für komplexere Migrationen zu Rate gezogen werden. Grundsätzlich kann der Header *Include-Resource* (in der nachfolgenden Erweiterten Backus-Naur-Form als `iclause` referenziert) die folgenden Formen annehmen[6]:

[6] Quelle: http://www.aqute.biz/Code/Bnd#include-resource

Listing 6.4: Include-Resource-Definition

```
iclause    ::= inline | copy
copy       ::= '{' process '}' | process
process    ::= assignment | simple
assignment ::= PATH '=' simple
simple     ::= PATH parameter*
inline     ::= '@' PATH ( '!/' PATH? ('/**' | '/*')? )?
parameters ::= 'flatten' | 'recursive' | 'filter'
```

Im Fall von `assignment` oder `simple` kann der `PATH`-Parameter auf eine Datei oder ein Verzeichnis verweisen. Liegt eine JAR-Datei im Klassenpfad, genügt die Angabe des Dateinamens (ohne Verzeichnisanteil). Die `simple` -Variante platziert die Ressource im Stammverzeichnis des Zielarchivs und schneidet dabei den Pfadanteil ab. Wird z.B. eine Datei `src/a/b.c` angegeben, legt bnd eine Datei `b.c` im Stammverzeichnis des Zielarchivs an.

Zeigt `PATH` auf ein Verzeichnis, werden lediglich die enthaltenen Dateien kopiert, nicht aber das Verzeichnis im Zielarchiv angelegt. Soll die Ressource innerhalb des Zielarchivs in einem bestimmten Verzeichnis liegen, wird die `assignment`-Variante verwendet; der Teil vor dem Gleichheitszeichen bestimmt dabei den Zielpfad, der dahinter die Ressource im Dateisystem. Wird eine Datei dort nicht gefunden, sucht bnd den Java-Classpath nach passenden Einträgen ab und nimmt den erstbesten mit demselben Dateinamen.

Bei der `inline`-Variante wird eine ZIP- oder JAR-Datei angegeben, welche – abgesehen vom Manifest – im Zielarchiv vollständig ausgepackt werden soll, es sei denn, es ist eine Einschränkung angegeben. Eine Einschränkung kann entweder die Angabe einer bestimmten Datei oder eines Verzeichnisses sein, Letzteres ggf. gefolgt von „*“ oder dem gleichwertigen „**“ für die rekursive Bestimmung aller Einträge des Verzeichnisses.

Die `simple` und die `assignment`-Variante können Dateien mit geschweiften Klammern (z.B. {foo.txt}) aufweisen, welche vorverarbeitet und darin enthaltene bnd-Variablen oder -Makros dabei aufgelöst werden. Die Direktive `recursive:` bestimmt, dass das angegebene Verzeichnis rekursiv in das Zielarchiv aufgenommen wird. Die `flatten:`-Direktive wiederum bestimmt, dass das Ergebnis rekursiver Dateisuche ohne Unterverzeichnisse im Zielarchiv landet. Die `filter:`-Direktive schließlich ist ein optionaler Präprozessor auf den Ressourcennamen und verwendet dasselbe Format wie oben bei den geschweiften Klammern.

Das nachfolgende Beispiel veranschaulicht die Verwendung verschiedener Varianten und deren Kombination anhand eines auszupackenden Archivs, einer vorzubearbeitenden Datei und einer Kopie in ein Zielverzeichnis:

```
Include-Resource: @osgi.jar, {LICENSE.txt}, acme/Merge.
   class=src/acme/Merge.class
```

6.3.2 Festlegen der Metadaten

Die aus der Standardmigration hervorgegangene Steuerdatei kann beinahe unverändert für die Beistellungsmigration wiederverwendet werden. Allerdings ist die darin enthaltene Definition des Bundle-Headers *Import-Package* im Falle der Beistellungsmigration nicht mehr sinnvoll, da die Auflösung von BouncyCastle-Packages nun vorausgesetzt werden kann. Die aufgenommenen Java-Archive bringen wiederum neue externe Abhängigkeiten mit sich, die importiert werden müssen. Welche das sind, kann man herausfinden, indem man die bereits beschriebenen zwei neuen Bundle-Header *Include-Resource* und *Bundle-ClassPath* in die bisherige Steuerdatei aufnimmt und *Import-Package* mittels eines vorangestellten Doppelkreuzes „#“ vorläufig auskommentiert. Der `bnd build`-Lauf wird zwar fehlerlos absolviert, das Ergebnis von `bnd print` lautet jedoch im vorliegenden Beispiel:

```
One error
1 : Unresolved references to [javax.activation, javax.
    crypto.interfaces, javax.crypto.spec, javax.mail,
    javax.mail.internet, javax.naming, javax.naming.
    directory, javax.security.auth.x500, junit.framework,
    junit.textui]
```

Diese Abhängigkeiten sind transitiv hinzugekommen und von bnd nicht automatisch aufgelöst worden. Neben einigen `javax`-Packages aus der Java Standard Edition tauchen zwei junit-Packages auf, die im Produktivsystem sicherlich entbehrlich sind und daher mit dem Attribut für optionale Auflösung in die *Import-Package*-Liste aufgenommen werden. Die `javax`-Packages werden lediglich mit ihrem Namen aufgeführt, da sie keiner Sonderbehandlung bedürfen.

Um die beiden Varianten – mit und ohne enthaltenes BouncyCastle-Archiv – auseinanderhalten zu können, werden die *Bundle-Version* und der *Bundle-Name* um einen Hinweis ergänzt. Die übrigen Einträge in der bisherigen Steuerdatei, insbesondere der *Bundle-SymbolicName*, bleiben unverändert, werden aber gemeinsam mit den neuen und geänderten Einträgen als eigene Steuerdatei namens `itext-bc.bnd` abgelegt.

6.3.3 Durchführen der Migration

Die Steuerdatei hat folgenden Inhalt:

Listing 6.5: itext-bc.bnd

```
bsn: biz.gossipmonger.com.itextpdf.text
version: 5.0.0.a
Bundle-RequiredExecutionEnvironment: J2SE-1.4,J2SE-1.5,
    JavaSE-1.6
Export-Package: com.itextpdf.text**
Bundle-SymbolicName: ${bsn}
```

```
Bundle-Version: ${version}
Bundle-Name: iText PDF Library (OSGified) with
   BouncyCastle Archives
-classpath: iText-5.0.0.jar
-output: ${bsn}-${version}.jar

Include-Resource: \
        lib/bcprov-jdk14-144.jar=bcprov-jdk14-144.jar, \
        lib/bcmail-jdk14-144.jar=bcmail-jdk14-144.jar, \
        lib/bctsp-jdk14-144.jar=bctsp-jdk14-144.jar

Bundle-Classpath: \
        ., \
        lib/bcprov-jdk14-144.jar, \
        lib/bcmail-jdk14-144.jar, \
        lib/bctsp-jdk14-144.jar

Import-Package: \
        javax.activation, \
        javax.crypto.interfaces, \
        javax.crypto.spec, \
        javax.mail, \
        javax.mail.internet, \
        javax.naming, \
        javax.naming.directory, \
        javax.security.auth.x500, \
        junit.framework;resolution:=optional, \
        junit.textui;resolution:=optional, \
        *
```

Die Migration wird mittels `bnd itext-bc.bnd` wie gehabt durchgeführt.

6.3.4 Bereitstellung

Dieses Bundle wird mit der Version *5.0.0.a* als Hinweis auf die Aufnahme des BouncyCastle-Archivs und zur Unterscheidung gegenüber anderen Varianten in das lokale Maven-Repository installiert. Die übrigen Angaben für den Maven-Installationsbefehl sind – abgesehen vom Bundle-Dateinamen – dieselben wie bei der Standardmigration. Der Installationsbefehl für die Beistellungsmigration lautet:

Listing 6.6: Installation des migrierten Bundles (Beistellung)

```
$ mvn install:install-file \
 -DgroupId=biz.gossipmonger.com.itextpdf \
 -DartifactId=biz.gossipmonger.com.itextpdf.itext \
 -Dversion=5.0.0.a \
 -Dpackaging=jar \
 -Dfile=biz.gossipmonger.com.itextpdf.text-5.0.0.a.jar
```

6.4 Transitive Migration

Die transitive Migration beschreibt die Migration aller beteiligten Java-Archive, also sowohl die der im Fokus stehenden Bibliothek als auch die aller sonstigen Bibliotheken, von denen erstere transitiv abhängt.

Dies ist naturgemäß die aufwendigste Form der Migration, die immer dann gewählt werden sollte, wenn die transitiven Java-Archive in weiteren Projekten genutzt werden, die ebenfalls zur Migration anstehen, oder wenn ein überwiegender Anteil der Abhängigkeiten bereits als Bundle vorliegt. Hier lohnt sich stets eine Recherche in den im Netz verfügbaren Bundle Repositories (siehe Kapitel 3.4.3 auf Seite 62). Zudem gilt es bei jeder betroffenen Bibliothek zu verifizieren, ob gegebenenfalls eine etwas ältere, aber bereits migrierte Bibliotheksversion verwendet werden kann.

6.4.1 Festlegen der Metadaten

Um iText transitiv zu migrieren, müssen die Abhängigkeiten zu den BouncyCastle-Packages über Bundles aufgelöst werden, die diese Packages exportieren. Es gilt zunächst festzustellen, ob bereits ein OSGi-Bundle vorliegt, das die benötigten BouncyCastle-Packages exportiert. Eine Recherche im SpringSource Enterprise Bundle Repository nach `bouncy castle` ergibt zwei Treffer: *The Bouncy Castle Crypto APIs 1.39.0* und *The Bouncy Castle Secure Email 1.39.0*.

Ein Blick auf die Liste der *Exported Packages* des Crypto-Bundles zeigt, dass es bereits die meisten der benötigten Packages exportiert – allerdings fehlen die Packages `org.bouncycastle.cms` und `org.bouncycastle.tsp`. Das E-Mail-Bundle exportiert `org.bouncycastle.cms` und weist Abhängigkeiten zu den Bundles *Java Activation API 1.1.1* und *Java Mail 1.4.1* auf. Das Package `org.bouncycastle.tsp` wiederum wird von keinem der Bundles des SpringSource Enterprise Bundle Repository exportiert.

Zum Zeitpunkt der Drucklegung dieses Buches steht BouncyCastle in der Version 1.44.0 zur Verfügung. Auf der Website[7] findet sich eine Tabelle mit Java-Archiven in verschiedenen Varianten und einer Spalte namens TSP, die Verweise auf Archive mit dem benötigten Package enthalten. Ob das TSP-Package in Version 1.44.0 friedlich mit den bei Spring vorrätigen BouncyCastle-Bundles der Version 1.39.0 zusammenarbeitet, ist ohne Test nicht herauszubekommen. Es erscheint daher ratsam, die zur Auflösung der iText-Abhängigkeiten benötigten BouncyCastle-JARs `bcprov-jdk14-144.jar`, `bcmail-jdk14-144.jar` und `bctsp-jdk14-144.jar` selbst zu analysieren und um die notwendigen Metadaten anzureichern. Das bisher geschilderte Vorgehen dient dabei als Richtschnur.

[7] http://www.bouncycastle.org/latest_releases.html

6.4.2 Migration der BouncyCastle-Archive

Da es sich insgesamt um drei zu migrierende Archive handelt, lohnt ein Blick auf deren Gemeinsamkeiten. Alle Archive sind ab der J2SE 1.4 lauffähig, tragen dieselbe Versionsnummer und – abgesehen vom Präfix – denselben Dateinamen. Die Ziel-Bundles sollen außerdem alle nach dem Schema <Symbolischer Name>-<Version>.jar benannt werden. Daraus ergibt sich eine Menge von Variablen, Headern und Direktiven, die in einer allgemeinen Steuerdatei abgelegt wird:

Listing 6.7: bc-common.bnd

```
-classpath: ${artifactName}-jdk14-144.jar
bsn: biz.gossipmonger.org.bouncycastle.${artifactName}

version: 1.44.0
-output: ${bsn}-${version}.jar

Bundle-RequiredExecutionEnvironment: J2SE-1.4, J2SE-1.5,
   JavaSE-1.6
Bundle-SymbolicName: ${bsn}
Bundle-Version: ${version}
```

Der sowohl von der `-classpath`-Direktive in Zeile 1 als auch der `bsn`-Variablen in Zeile 2 referenzierte `artifactName` ist hier nicht definiert und muss folglich in den spezifischen Steuerdateien für die einzelnen Archive angegeben werden, damit alle Variablen aufgelöst werden können.

Migration des TSP-Archivs

Die Analyse des BouncyCastle TSP-Archivs ergibt lediglich Abhängigkeiten zu anderen BouncyCastle-Packages; es enthält zudem keine Packages privaten Charakters. Damit verbleiben nur wenige spezifische Informationen, um dieses Archiv unter Einbeziehung der allgemeinen Migrationsvorschriften in ein Bundle zu verwandeln:

Listing 6.8: bc-tsp.bnd

```
-include: bc-common.bnd
artifactName: bctsp
Bundle-Name: Bouncy Castle TSP (Time Stamp Protocol)
Export-Package: org.bouncycastle.tsp
```

Die `-include`-Direktive bindet die allgemeinen Regeln ein, anschließend wird der `artifactName` definiert. Ein verständlicher Bundle-Name und der Export des einzigen enthaltenen Packages runden die wenigen benötigten Einträge für diese Migration ab.

Migration des Mail-Archivs

Die Vorschriften zur Migration des Mail-Archivs sind gleich strukturiert, enthalten allerdings einen interessanten Header für den Export der Packages:

Listing 6.9: bc-mail.bnd

```
-include: bc-common.bnd
artifactName: bcmail
Bundle-Name: Bouncy Castle Secure Mail (CMS and S/MIME)
Export-Package: !*.examples, org.bouncycastle**
```

Der erste Eintrag dieses Headers weist bnd an, keine Packages in die Exportliste aufzunehmen, die auf `.examples` enden. Diese Packages sind offenbar privater Natur und sollen daher von anderen Bundles nicht benutzt werden können. Der zweite Eintrag bestimmt, dass alle übrigen Packages mit dem Präfix `org.bouncycastle` in den Bundle-Header *Export-Package* aufgenommen werden sollen.

Migration des Provider-Archivs

Zu guter Letzt kommt das Provider-Archiv an die Reihe; strukturell ebenfalls gleich aufgebaut, enthält es einen weiteren Eintrag für das Importieren von Packages. Die referenzierten junit-Packages sollen wie in der Beistellungsmigration mit optionaler Auflösung attribuiert, der Rest des *Import-Package*-Headers von bnd automatisch ermittelt werden:

Listing 6.10: bc-prov.bnd

```
-include: bc-common.bnd
artifactName: bcprov
Bundle-Name: Bouncy Castle Crypto Provider
Export-Package: !*.examples, org.bouncycastle**
Import-Package: junit.*;resolution:=optional, *
```

Bereitstellung

Die drei erstellten Bundles werden nun in das lokale Maven-Repository installiert; beispielhaft sei hier der Befehl zur Installation des Provider-Bundles abgedruckt:

Listing 6.11: Installation BouncyCastle Provider-Bundle

```
$ mvn install:install-file \
 -DgroupId=biz.gossipmonger.org.bouncycastle \
 -DartifactId=biz.gossipmonger.org.bouncycastle.bcprov \
 -Dversion=1.44.0 \
 -Dpackaging=jar \
```

```
-Dfile=biz.gossipmonger.org.bouncycastle.bcprov-1.44.0.
    jar
```

6.4.3 Gemeinsame Migration aller BouncyCastle-Archive

Die drei BouncyCastle-Archive lassen sich wie eben beschrieben einzeln migrieren. Es ist aber auch möglich, diese drei Archive gemeinsam in ein neues Archiv zu verpacken und mit OSGi-Metadaten zu versehen. Dies ist letztlich eine Kombination aus der transitiven und der Beistellungsmigration.

Festlegen der Metadaten

Aus der Beistellungsmigration wird der Bundle-Header *Include-Resource* übernommen, dieses Mal mit vorangestelltem *inline*-Kennzeichner „@“ für jedes der drei Archive:

```
Include-Resource: @bcprov-jdk14-144.jar, @bcmail-jdk14
    -144.jar, @bctsp-jdk14-144.jar
```

Auf den dortigen *Bundle-ClassPath* kann hingegen verzichtet werden, da die Inhalte der drei Archive auf oberster Ebene in einem neuen Archiv zusammengeführt und damit über das Stammverzeichnis angesprochen werden können.

Die weiteren Migrationsvorschriften können aus der Steuerdatei für die BouncyCastle Provider Migration übernommen werden, die die Obermenge aller Migrationsvorschriften für die drei Archive bildet. Allerdings müssen sowohl der *artifactName* als auch der *Bundle-Name* angepasst werden. Zudem ist an zwei Stellen ein Eingriff notwendig: Einerseits werden mehrere Dateien im Klassenpfad benötigt, und dieser soll andererseits nicht von der Definition der übergreifenden Steuerdatei bc-common.bnd wieder überschrieben werden. Daher wird dieser eingebundenen Datei eine Tilde vorangestellt, die verhindert, dass ihr Inhalt bisherige Definitionen überschreibt. Und zum Zweiten wird die Direktive `-classpath` mit den drei Archiven versehen.

Durchführen der Migration

Damit ergibt sich eine Steuerdatei bc-all.bnd mit folgendem Inhalt:

Listing 6.12: bc-all.bnd

```
-include: ~bc-common.bnd
artifactName: bcall
Bundle-Name: Bouncy Castle Crypto Provider, Mail and TSP
Export-Package: !*.examples, org.bouncycastle**
Import-Package: junit.*;resolution:=optional, *
Include-Resource: @bcprov-jdk14-144.jar, \
                  @bcmail-jdk14-144.jar, \
                  @bctsp-jdk14-144.jar
```

```
-classpath: bcprov-jdk14-144.jar, \
            bcmail-jdk14-144.jar, \
            bctsp-jdk14-144.jar
```

Die Migration wird mittels `bnd bc-all.bnd` durchgeführt.

Bereitstellung

Das Bundle mit den Inhalten der drei Archive wird nun wie folgt in das lokale Maven-Repository installiert:

Listing 6.13: Installation des BouncyCastle-Bundles

```
$ mvn install:install-file \
 -DgroupId=biz.gossipmonger.org.bouncycastle \
 -DartifactId=biz.gossipmonger.org.bouncycastle.bcall \
 -Dversion=1.44.0 \
 -Dpackaging=jar \
 -Dfile=biz.gossipmonger.org.bouncycastle.bcall-1.44.0.
    jar
```

6.4.4 Festlegen der Metadaten

Die Steuerdatei aus der Standardmigration von iText kann mit wenigen Änderungen für dessen transitive Migration verwendet werden. Analog zur Beistellungsmigration kommen die Abhängigkeiten zu den BouncyCastle-Packages nun ohne das Attribut `resolution:=optional` aus. Da dies die einzige Besonderheit in der Definition des *Import-Package*-Headers war, kann dieser komplett entfallen, und bnd verwendet den Standardwert „*“ zur Listung aller gefundenen Import-Statements. Wer allerdings sichergehen möchte, dass eine ganz bestimmte Version dieser eingebundenen Packages verwendet wird, kann dies anstelle des `resolution`-Attributs mit dem Attribut `version` festlegen. Sind die Packages nicht versioniert, das Bundle hingegen schon, kann stattdessen auch auf die `bundle-version` verwiesen werden. Dies ist auch in diesem Fall ratsam und wird folgendermaßen definiert (siehe Abschnitt 6.5 auf Seite 124):

```
Import-Package: org.bouncycastle**;bundle-version="
   [1.44.0, 1.44.0]", *
```

Zudem wird wie in der Beistellungsmigration die Versionsnummer um einen Qualifizierer `.b` ergänzt, um auch diese Variante von den anderen unterscheiden zu können.

6.4.5 Durchführen der Migration

Die resultierende Steuerdatei ähnelt derjenigen der Standardmigration sehr:

Listing 6.14: itext-transitiv.bnd

```
bsn: biz.gossipmonger.com.itextpdf.text
version: 5.0.0.b
Bundle-RequiredExecutionEnvironment: J2SE-1.4, J2SE-1.5,
   JavaSE-1.6
Import-Package: org.bouncycastle**;bundle-version="1.44.0
   ", *
Export-Package: com.itextpdf.text**
Bundle-SymbolicName: ${bsn}
Bundle-Version: ${version}
Bundle-Name: iText PDF Library (OSGified)
-classpath: iText-5.0.0.jar
-output: ${bsn}-${version}.jar
```

Die Durchführung der transitiven Migration unterscheidet sich nicht von der Durchführung der bisherigen Alternativen: `bnd itext-transitiv.bnd`.

6.4.6 Bereitstellung

Die erstellte Datei `biz.gossipmonger.com.itextpdf.text-5.0.0.b.jar` wird in das lokale Maven-Repository installiert; der hierfür notwendige Maven-Befehl entspricht inhaltlich den bisherigen:

Listing 6.15: Installation des transitiv migrierten Bundles

```
$ mvn install:install-file \
 -DgroupId=biz.gossipmonger.com.itextpdf \
 -DartifactId=biz.gossipmonger.com.itextpdf.itext \
 -Dversion=5.0.0.b \
 -Dpackaging=jar \
 -Dfile=biz.gossipmonger.com.itextpdf.text-5.0.0.b.jar
```

6.5 Versionen und Versionsbereiche

OSGi definiert in Kapitel 3.2.5 der Core Specification eine Syntax für Versionsnummern. Eine Versionsnummer ist aufgeteilt in die Bereiche *major*, *minor*, *micro* und *qualifier*, jeweils getrennt durch einen Punkt. Der *qualifier* darf alphanumerische Zeichen, den Unter- und den Bindestrich enthalten; alle anderen sind numerische Werte. Die Angabe einer Version dient der eindeutigen Identifikation eines Artefakts.

Einem Bundle kann über den Header *Bundle-Version* eine Version zugewiesen werden; unterbleibt dies, wird als Version *0.0.0* verwendet. Ein Package kann mit einer bestimmten Version exportiert werden; fehlt dieses Attribut, wird ebenfalls die Version *0.0.0* verwendet.

6.5.1 Referenzierung über Versionen

Werden Bundles oder Packages durch *Require-Bundle* bzw. *Import-Package* referenziert, kann eine Version die Auflösung weiter eingrenzen. Die Angabe einer Versionsnummer wird in diesen Fällen stets als Intervall interpretiert. *Import-Package: p;version="1.2.3"* bedeutet daher nicht die Festlegung auf **genau** die Version 1.2.3, sondern fordert vielmehr **mindestens** die Version 1.2.3, auch 1.4 oder 2.1 wären zulässig. Diese implizite Intervallangabe ist vielleicht gewöhnungsbedürftig, aber logisch und vereinfacht die Angabe der häufig benötigten Regel *verwende mindestens Version x*. Soll tatsächlich nur genau eine Versionsnummer zum Tragen kommen, muss hingegen das vollständige mathematische Intervall angegeben werden, z.B. *Import-Package: p;version="[1.2.3, 1.2.3]"* mit jeweils einschließenden eckigen Klammern. Runde Klammern schließen aus. Weitere Details und Beispiele zu Versionsbereichen sind in Kapitel 3.2.6 der OSGi Core Specification enthalten.

Hängt man von Bibliotheken ab, die noch keinen ausgereiften Zustand erreicht haben oder häufig aktualisiert werden, empfiehlt sich die Angabe eines klar umrissenen Versionsbereichs. Damit stellt man eine Mindestversion sicher und vertraut auf die Abwärtskompatibilität künftiger Artefakte bis zu einem gewissen Grad. Üblich sind solche „Vertrauensintervalle“ auf der zweiten Versionsnummer, also z.B. `version=[1.44, 2)`, welches Aktualisierungen mit Änderungen der Versionsnummer an der zweiten Stelle (Minor changes) akzeptiert, nicht jedoch einen erstrangigen Versionswechsel von 1 auf 2 oder gar noch höher.

6.5.2 Konventionen

Es gibt leider keine Festlegungen hinsichtlich des Zusammenhangs von Versionierung und Abwärtskompatibilität. Größere Intervalle sind daher mit Vorsicht zu genießen und können zu schwer nachvollziehbaren Fehlern führen. Allerdings versucht ein wirklich lesenswerter Blog-Eintrag von Peter Kriens[8], hier für mehr Klarheit zu sorgen. Demnach ist die einfachste Variante von Versionierung die einer festen Identität. A importiert B in Version 1. Ändert sich B, erhält es Version 2, und A muss neu kompiliert werden und ändert sich zudem ebenfalls durch die geänderte Abhängigkeit. Damit erhält auch A eine neue Version usf. Das ist für dynamische oder Drittsoftware verwendende Systeme kein praktikables Vorgehen, kleinste Änderungen ziehen fortwährend einen Rattenschwanz an Zwangs-Updates nach sich.

Diesem Zwang kann man mit Versionsbereichen entfliehen. Wenn A für B einen Versionsbereich [1,2) vorsieht, kann sich B von 1.1 nach 1.2 und 1.3 ändern, ohne dass A deswegen neu kompiliert oder sonst wie geändert werden müsste. Ist eine Änderung von B abwärtskompatibel, so wird die *minor*-Version (die zweite Stelle) hochgezählt; ist sie es nicht, wird die *major*-Version (die erste Stelle) erhöht und die *minor*-Version zurückgesetzt. Für wen die Abwärtskompatibilität tatsächlich gilt, hängt allerdings von der Verwendung ab. Wird eine Schnittstelle

[8] Peter Kriens about Versions: http://www.osgi.org/blog/2009/12/versions.html

um eine Methode erweitert, bleibt für deren Verwender alles beim Alten, während deren Implementierer auf jeden Fall Handlungsbedarf hat. Daher wird der Implementierer einer Schnittstelle deren Versionsbereich anders definieren, nämlich als mögliche Wechsel auf der dritten Stelle (*micro*), zum Beispiel [1.1,1.2). Änderungen auf der dritten Stelle sind folglich für Bugfixes und kleine Änderungen ohne Außenwirkung da.

Zwar sind all diese Betrachtungen ohne Normierungskraft, doch beleuchten sie das Prinzip der Versionierung sehr gut. Mit dieser Art der Handhabung von Versionen hat es zum Beispiel Eclipse geschafft, die immense Flut von Komponenten und deren Abhängigkeiten in den Griff zu bekommen. Es steht jedem frei, dasselbe zu tun.

6.6 Fazit

Mit dem dargestellten strukturierten Vorgehen ist die Erweiterung auch komplexer Java-Archive um OSGi-Metadaten recht schnell erledigt. Besonderes Augenmerk sollte man auf die Analyse von Abhängigkeiten legen und sich gut überlegen, welche Migrationsstrategie für das konkrete Vorhaben am besten geeignet scheint. Die schnellste Variante ist die *Standardmigration* mit der optionalen Auflösung (resolution:=optional) von Abhängigkeiten; allerdings kann es hierbei zu Laufzeitfehlern, zum Beispiel zu einer ClassNotFoundException kommen, die man zur Erstellungszeit kaum erkennen kann. Böse Zungen sprechen daher auch von der „Optimistenmigration“.

Weniger anfällig ist die *Beistellungsmigration*, aber sie ist etwas aufwendiger und weist große Ähnlichkeiten mit dem Konzept aufgeblähter, viel Beiwerk enthaltender Bibliotheken auf. Ihr größter Vorteil ist sicherlich die Gewissheit, alle notwendigen Packages dabeizuhaben, und damit Versionskonflikten oder einer scheiternden Auflösung einen Riegel vorzuschieben. Allerdings ist die regelmäßige Größe solcher Migrationsartefakte mit den ursprünglichen Prinzipien von OSGi – klein, klar, schnell – kaum vereinbar.

Am saubersten fährt man mit der *transitiven Migration*, die leider zugleich den größten Aufwand bedeutet. Sie lohnt sich vor allem dann, wenn viele Projekte auf denselben Bibliotheken basieren und peu à peu migriert werden sollen. Nur so gelingt es nachhaltig, die bisher üblichen oder durch die Beistellungsmigration neu entstehenden „Bytemonster“ auf die tatsächlich notwendige Größe einzudampfen, ohne benötigte Funktionalität zu verlieren. Wer sich an die Erstellung von bnd-Steuerdateien gewöhnt und deren Syntax verinnerlicht hat, sollte keine Schwierigkeiten mit der Migration beliebig vieler einzelner Bibliotheken haben und sich daher der transitiven Migration widmen.