adesso

Java Magazin, März 2010

Flucht aus der „Jar-Hölle“

OSGi ist eine Ergänzung zu bestehenden Java-Architektur-Pattern. Es bietet die Möglichkeit, Software nach individuellen Anforderungen zu entwickeln. Außerdem können mithilfe eines starken Komponentenmodells unübersichtliche Java-Anwendungen organisiert werden. Welche Chancen und Risiken daraus erwachsen, wird in diesem Artikel anhand typischer Einsatzszenarien skizziert.

von Sven Johann und Bernd Böllert

Java-EE-Anwendungen haben sich in den letzten Jahren stark gewandelt. Von schwer zu handhabenden Programmiermodellen wie EJB 1.x/2.x hin zu annotierten, POJO-basierten EJB-3.0-Anwendungen. Trotz dieser starken Vereinfachung bleiben noch einige Probleme unbeachtet:

- Schwache Serviceregistrierung/dynamische Lokalisierung von Serviceprovidern: Mittels JNDI kann in EJB dynamisch ein Service lokalisiert werden. Aber es existiert keine Metainformation darüber, ob der Service in der geforderten Version vorliegt.
- Es fehlt ein Software-Lifecycle-Prozess auf Serviceebene. Eine JEE-Anwendung kann nur als Ganzes (*war* oder *ear*) gestartet, gestoppt und aktualisiert werden. Das widerspricht dem serviceorientierten Ansatz. Das heißt, wenn in einer SOA servicezentrierte Lifecycles realisiert werden, entsteht ein Architektur- bzw. Paradigmenbruch, da die Innensicht der Services gewechselt wird.
- Schwaches Komponentenmodell: In Java kann nur auf Klassenebene agiert werden. *Jar/war/ear*-Dateien helfen zwar, Klassen zu Komponenten bzw. Applikationen zusammenzufassen. Aber für die komponentenbasierte Entwicklung ist das wenig dienlich. Hier wird die Anwendung in logische Komponenten zerlegt, die wiederum aus logischen Komponenten bestehen usw. Ein Programmiermodell, das dies unterstützt, wäre hilfreich.
- Schwaches Komponentenmanagement: Betrachtet man eine normale Java-EE-Anwendung, die zusammen mit anderen Java-EE-Anwendungen in einem Application-Server eingesetzt wurde, kann einem schnell schwindelig werden, sobald man sich die vielen *jar*-Dateien ansieht, die für die jeweilige Anwendung benötigt werden. Welches *jar* wird für welche Anwendung benötigt? Werden alle gebraucht? In welcher Version ist welches *jar* erforderlich? Gibt es Konflikte? Jeder hat sich eine dieser Fragen bestimmt schon gestellt. Die *jar*-Dateien liegen zudem ungeordnet in allen möglichen Verzeichnissen, eine geordnete Ablage à la Maven, aus der sich alle Applikationen bedienen, wäre wünschenswert.
- Für Aufgaben, die in einer Virtual Machine (VM) erledigt werden, ist der Footprint von EJB sehr groß. Insbesondere, wenn man keine verteilten Services plant oder die Verteilung etwa über Web Services realisiert. Betrachtet man Apache Felix [1], kommt der Overhead, den das Framework mitbringt, auf eine 600 KB große Distribution. Dass OSGi gerade im Bereich der Smartphones so beliebt ist, verwundert daher nicht.

OSGi bzw. Applikationen rund um OSGi haben sich zum Ziel gesetzt, diese genannten Probleme zu lösen und bessere Java-EE-Applikationen zu ermöglichen.

Versionierung

Eine Versionierung ist notwendig, um die Kompatibilität in Servicearchitekturen

zu überprüfen. Daher ist sie ein wichtiger OSGi-Baustein. Die Versionierung der Abhängigkeiten wird im *Manifest.mf* definiert. Sie erfolgt nach folgendem Muster:

```
Import-Package: com.acme.foo;version="[1.23, 2)",
com.acme.bar;version="[4.0, 5.0)",
com.acme.baz;version=1.2
```

Die Versionen müssen als Intervalle angegeben werden. Dabei bedeutet ein einzelner Wert, dass alles ab dieser Version als erlaubte Version definiert ist. Die Intervalle können mit eingeschlossenen Grenzen *[]* oder ausgeschlossenen Grenzwerten *()* definiert werden. Kombinationen sind möglich. Obwohl es in der Spezifikation [2] keine Details zur Nummerierung gibt, wird ein dreigliedriges Versionsnummernkonzept bevorzugt. Die Wertung der Stellen *Major.Minor.Fix* ist so zu interpretieren, dass die erste Stelle bei Versionssprüngen mit Kompatibilitätsverlust verwendet wird, die zweite Stelle für Erweiterungen und die dritte Stelle für Bugfixes ohne Erweiterungen der Funktionalität.

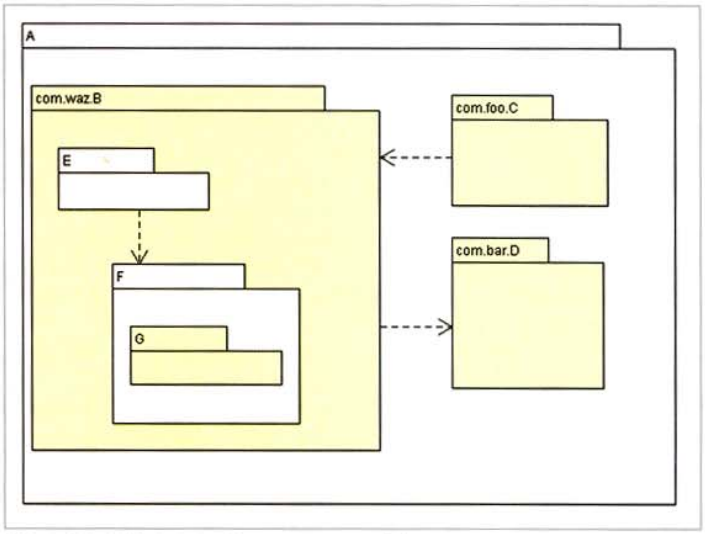

Abb. 1: Beispiel für logische Komponenten eines Softwaresystems

Lifecycle

Der Lifecycle aller OSGi-Bundles ist gleich. Durch den Schritt *install* wird ein Bundle, sofern es korrekt beschrieben ist, installiert. Es kann in den Zustand *resolved* überführt werden, wenn alle importierten Referenzen erfüllt sind. Viele Frameworks belassen Bundles im „Installed Status" und lösen sie erst auf, wenn sie angefordert werden. Das hat zur Folge, dass das System Ressourcen nur dann belegt, wenn sie notwendig sind. Dadurch verkürzt sich auch die Zeit, bis ein System „initial" zur Verfügung steht, beispielsweise bei Eclipse auf Equinox. Im Zustand *resolved* steht das API den dienstnutzenden Bundles zur Verfügung. Optional kann ein *BundleActivator* definiert sein, der in der Startphase synchronisiert (Threadsafe) das Bundle initialisiert. Symmetrisch dazu gibt es im Activator optional die Stopphase, um Ressourcen wieder frei zu geben.

Im Zustand *uninstalled* wird ein Bundle aktualisiert bzw. kann entfernt werden, wenn es aktuell nicht im Zugriff ist. Ist das Bundle durch andere installierte Bundles im Zugriff, bleibt es verwendbar bis zum Neustart des Frameworks. Im Lifecycle besteht auch die Möglichkeit, auf Serviceebene ein „Hot Deployment" durchzuführen, d. h. im laufenden Framework werden Aktualisierungen auf Bundle-Ebene ausgeführt.

Servicekomponentenmodelle

Motivation für Servicekomponentenmodelle sind folgende:

1. Vermeidung von Glue Code. POJOs sind leicht zu testen. Glue Code ist repetitiv und sollte, wenn möglich, vermieden werden.
2. Die definierten Serviceübergänge können leicht mit allgemeiner Funktionalität erweitert werden, z. B. Logging, Thread-Kontrolle und Transactions-Behandlung.
3. Verwaltung der Beziehung von Interfaces (Service), Implementierungen (Serviceprovider) und Metainformationen, z. B. Kardinalität, also die Entscheidung, ob Singleton oder nicht, kann per Konfiguration erfolgen.

Starkes Komponentenmodell durch OSGi

Verschiedene Autoren haben Vorgehensmodelle zur komponentenbasierten Architekturentwicklung beschrieben, z. B. Gernot Starke und Peter Hruschka [3], Ralf Westphal [4] oder das Fraunhofer IESE [5]. Allen geht es prinzipiell darum, eine Softwarearchitektur durch stetige Verfeinerung zu entwickeln. Man beginnt mit einer sehr abstrakten Sicht auf das System und dessen Kontext und geht immer mehr in Richtung Details vor. So besteht ein Softwaresystem aus mehreren Komponenten, die wiederum aus Komponenten bestehen, die wiederum aus Komponenten bestehen usw. Gemäß der Devise „Divide and Conquer" verfeinern wir ein System schrittweise, um nicht durch Details erschlagen zu werden. Die Komponenten des Systems werden durch die Services beschrieben, die sie exportieren, und jene, die sie von anderen Komponenten zum korrekten Funktionieren importieren müssen (Vor- und Nachbedingungen bzw. Zustandsverwaltung der Serviceaufrufe nicht berücksichtigt).

Auf der Ebene einer Programmiersprache wie z. B. Java kann diese logische Sicht nicht abgebildet werden. Es stehen zwar *jar*-, *war*- und *ear*-Dateien zur Verfügung, aber diese bilden unser System aus logischer Sicht nur mittelmäßig ab. Eine *ear*-Datei besteht aus verschiedenen Kindelementen wie *war*- und *jar*-Dateien. Es ist keine hierarchische und logische Zerlegung des Systems auf Programmiersprachenebene möglich. Auch gibt es keinerlei Informationen über das Abhängigkeitsnetz der *jar*-Dateien: Welche *jar*-Dateien (in welcher Version) benötigt eine *jar*-Datei, damit sie eingesetzt werden kann? Welche Services ex-

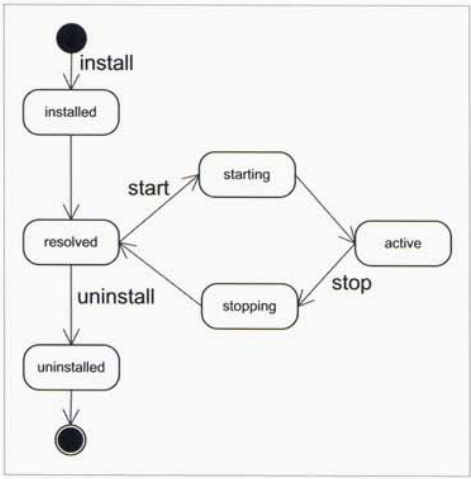

Abb. 2: Lifecycle von OSGi-Bundles

portiert sie und wer nutzt sie? Gut wäre es, mehrere *jar*-Dateien zu *Composite-jar*-Dateien zusammenfassen zu können und diese wiederum zu *Composite-jar*-Dateien usw. zu vereinen. Ebenfalls wäre es vorteilhaft, explizit die Abhängigkeiten der *Composite-jar*-Dateien untereinander sehen zu können.

Mit solchen Informationen ergeben sich völlig neue Möglichkeiten im Umgang mit einem System. Logische Komponenten können nun auch physisch abgebildet werden. Auch automatische Komponententests werden erleichtert. Denn die testende Komponente gibt Auskunft darüber, von welchen Services anderer Komponenten sie abhängt, die wir dann mocken können. Auch die Analyse wird vereinfacht, da Reverse-Engineering-Tools ein Komponenten-Abhängigkeitsnetz erzeugen können.

OSGi lässt es zu, einige Lücken zwischen logischen Komponenten und physischen Java-Elementen wie *jar*, *war* und *ear* zu schließen. In einem OSGi-Bundle sind die Abhängigkeiten zu anderen Bundles explizit beschrieben. Zusätzlich wird interessierten Bundles mitgeteilt, auf welche Services sie zugreifen können. Das steht in der Manifest-Datei des OSGi-Bundles (Kasten: „Import/Export der Komponente „com.acme.waz" in OSGi").

Nur ein Problem löst OSGi noch nicht: den verfeinerten Komponentenbaum, also Komponenten, die aus anderen zusammengebaut werden. Die Betonung liegt auf *noch nicht*. Denn genau das ist für die neue OSGi-Generation geplant. Der „Early Draft 3" der OSGi-Version 4.2 beschreibt den „RFC 0138 Multiple Frameworks In One JVM" [6]. Dieser erlaubt es, Vater-Kind-Beziehungen auf OSGi-Ebene zu definieren. Die Beziehungen werden in der *Composite-Manifest*-Datei beschrieben. Endgültig soll das aber erst in OSGi 5 passieren. Trotzdem implementieren einige OSGi-Frameworks wie Eclipse Equinox [7] oder Google Lemmon [8] den RFC bereits teilweise.

Starkes Komponentenmanagement durch OSGi

Die Verwaltung von *jar*-Dateien in einer Java-EE-Anwendung ist so katastrophal, dass sie manchmal als „Jar-Hölle" beschrieben wird (u. a. unter [9]). Selbst in kleinen Anwendungen ist es mittlerweile extrem schwer, die Abhängigkeiten der *jar*-Dateien untereinander nachzuvollziehen. Schlimmer noch: *jar*-Dateien sind völlig unübersichtlich in verschiedensten Verzeichnissen abgelegt, z. B. in einem *ear-lib*-Ordner, mehreren *war-lib*-Ordnern, im Server-lib des Application-Servers oder in einem *endorsed*-Verzeichnis des Application Servers. Welche Datei wann und von wem in welcher Version angezogen wird, bleibt dem Entwickler oftmals ein Rätsel. Es scheint auch anders zu gehen, denkt sich der Nutzer von Maven oder Eclipse. Hier gibt es mehr Ordnung: Maven hat eine saubere Verwaltung von *jar*-Dateien im Maven Repository. Ähnlich sieht es bei Eclipse aus: Es gibt ein Verzeichnis (*/plugins/*), in dem alle *jar*-Dateien mit Versions- und Abhängigkeitsinfo zu finden sind.

SpringSource stellt mit dem *dm*-Server [10] einen OSGi-Application-Server bereit, der sich dieser Probleme annimmt: Der *dm*-Server besitzt ein Repository-Verzeichnis, in dem alle benötigten *jar*-Dateien genau einmal als OSGi-Bundles zur Verfügung gestellt werden (wenn nötig, in mehreren Versionen). Die *jar*-Dateien sind allerdings nicht so schön geordnet wie bei Maven, sondern werden flach abgelegt. Alle Enterprise-Applikationen, die im *dm*-Server deployt sind, bedienen sich dann aus diesem Repository. Die Enterprise-Applikation selbst wird im Pickup-Verzeichnis abgelegt. Es gibt hier vielfältige Möglichkeiten (als *jar*-, *war*- oder *par*-Datei), die z. B. unter [11] ausführlich beschrieben werden. SpringSource bietet mit der Spring-Toolsuite [12] eine gute Unterstützung für OSGi und den Spring-*dm*-Server, so wie es auch für klassische Webapplikationen bekannt ist. Der *dm*-Server wird übrigens ein Eclipse-Projekt werden: Da kann man sich schon auf weitere Innovationen freuen [13].

OSGi überall

Die Einsatzmöglichkeiten für ein Servicekomponentenmodell sind vielfältig. Im Großen (SOA) aber auch im Kleinen (den Embedded Systems) ist OSGi im Einsatz. Neu ist die Verwendung auf Google-Android-Systemen und im Cloud Computing dank Scala. Aber was

Import/Export der Komponente „com.acme.waz" in OSGi

In Abbildung 1 sehen wir, dass die Komponente *com.acme.waz.1.0* der Anwendung *com.acme.system* Services der Komponente *com.acme.bar.2.3* nutzt. OSGi beschreibt das durch einen Eintrag in der Import-Package-Liste: *Import-Package: com.acme.bar;version=2.3.*

Genauso funktioniert der Export von Services. com.acme.waz.1.0 exportiert Services, die von com.acme.foo.2.2 genutzt werden. Das wird durch die Export-Package-Liste beschrieben: *Export-Package: com.acme.waz.*

Das heißt auch, dass nur die Klassen sichtbar sind, deren Package explizit exportiert wird. Wir haben also neue Sichtbarkeitsregeln: *public* ist nun nicht mehr *public* für alle Klassen, sondern nur noch für Klassen innerhalb des Bundles. Für andere Bundles sind nur noch die Teile des Bundles *public*, die durch die Manifest-Datei festgelegt wurden. Die Komponente *com.acme.waz.1.0* sieht also nur die *public*-Services des Packages *com.acme.bar* und keine Implementierungsdetails, z. B. von *com.acme.bar.impl.*

macht OSGi nun für diese Systeme so interessant? Google Android umfasst eine VM, die Java sehr stark ähnelt. Daher ist eine Portierung von OSGi-Frameworks möglich. Auf Android-Geräten gibt es eine Menge von nebenläufigen Prozessen, die oft denselben Service benötigen, manchmal aber in unterschiedlichen Versionen. Das ist, wie bereits beschrieben, mit OSGi ohne Weiteres möglich. Ein anderes Plus bietet die Möglichkeit, die Services während des Betriebs zu aktualisieren, was in einer Multithread-Umgebung ohne OSGi nicht trivial ist. Da Apache Felix eine sehr ressourcenschonende Implementierung ist, bietet sich für die Verwendung mit Android an, es auf schwacher Hardware einzusetzen. Die gleichen Punkte machen OSGi auch in der Verwendung im Cloud Computing interessant. Scala stellt für Cloud Computing die notwendige Architektur mit einem Actor Model und funktionalen Sprachelementen zur Verfügung. Was fehlt, ist ein Lifecycle sowie die Verwaltung von parallelen Versionen. Scala hat diese Schwäche gelöst, indem die etablierte OSGi-Architektur verwendet wird. Da Scala auf Java basiert und in einer Anwendung zwischen Scala und Java hin- und hergewechselt werden kann, bietet es sich an, den Wechsel jeweils über eine OSGi-Serviceschicht zu entkoppeln. Durch diese Entkopplung ist es möglich, existierende Anwendungen schrittweise zu modularisieren und wahlweise in Scala oder Pure Java zu realisieren.

Fazit und Ausblick

Probleme mit aktuellen Java-EE-Anwendungen lassen sich also mit OSGI mindern, wenn nicht sogar lösen. Versionsprobleme von *jar*-Dateien gehören der Vergangenheit an. Jeder bekommt, was er benötigt. Anwendungen sind dank des geringen Footprints, Lazy Loading und des Lifecycle-Modells ressourcenschonend, starten schneller und lassen sich zügiger und gezielter aktualisieren.

OSGi hat seinen Ursprung in den Embedded Systems im Automotive- oder Mobiltelefonbereich, aber alle in diesem Artikel vorgestellten Vorteile haben zu einer weiteren Verbreitung in Richtung Desktop (z. B. Eclipse) und Infrastruktur/Middleware (z. B. Oracle Weblogic basiert auf OSGi, JBoss stellt auf OSGi um) geführt. Der folgerichtige Schritt ist, dass nun auch Enterprise-Anwendungen diesen Weg einschlagen. Spring macht dies mit dem SpringSource-dm-Server. Weitere Schritte werden folgen. Das OSGi-Framework entwickelt sich stetig weiter und auch die Toollandschaft in diesem Bereich wird weiter wachsen, um die Entwicklung zu vereinfachen. Wichtig ist, dass OSGi die Lücke zwischen logischem Entwurf und physikalischer Implementierung schließen wird. Logische Komponenten finden sich bald nicht mehr nur in Modellen wieder, sondern es wird sie dank der Composite Bundles auch auf physikalischer Ebene geben. Ein Schritt in diese Richtung ist mit dem Bundle-Konzept und der Import-/Exportdefinition der Schnittstellen in der Manifest-Datei schon getan.

> Probleme mit aktuellen Java-EE-Anwendungen lassen sich mit OSGi mindern.

Wir dürfen gespannt sein, wie Oracle/Sun darauf reagiert. Der erste Schritt war, dass Oracle und SpringSource das Eclipse-Gemini-Projekt [14] für Enterprise OSGi gegründet haben. ■

Sven Johann ist Senior Software Engineer bei der adesso AG und befasst sich seit 2002 mit Komponentenentwicklung und seit 2006 mit OSGi.

Bernd Böllert ist Senior Software Engineer bei der adesso AG und befasst sich seit 1999 mit Unternehmensarchitekturen und Methoden rund um Java.

Links & Literatur

[1] Apache Felix OSGi R4 Service Platform: http://felix.apache.org/site/index.html
[2] OSGI-R4-Spezifikation: http://www.osgi.org/Download/Release4V42
[3] Gernot Starke und Peter Hruschka, arc42-Architektur-Template: http://www.arc42.de
[4] Ralf Westphal, „The Architects Napkin": http://geekswithblogs.net/thearchitectsnapkin/Default.aspx
[5] Colin Atkinson et al.: „Component-based Product Line Engineering with UML", Addison-Wesley Longman, 2001
[6] OSGI-R4-Spezifikation, Early Draft 3, Seite 47 bis 87: http://www.osgi.org/download/osgi-4.2-early-draft3.pdf
[7] Eclipse Equinox OSGi Platform: http://www.eclipse.org/equinox/
[8] Google Lemmon: http://code.google.com/p/lemmon/
[9] Jar-Hell bei Wikipedia: http://en.wikipedia.org/wiki/JAR_hell#JAR_hell
[10] SpringSource dm Server: http://www.springsource.com/products/dmserver
[11] Eberhard Wolff, kostenloses Spring-dm-Server-E-Book: http://www.dpunkt.de/ebooks_pdf/free/3231.pdf
[12] SpringSource-Toolsuite: http://www.springsource.com/products/sts
[13] Blog von Adrian Coyler: http://blog.springsource.com/2010/01/12/dm-server-project-moves-to-eclipse-org/
[14] Eclipse Gemini Proposal: http://eclipse.org/proposals/gemini/
[15] Java Module System: http://www.jcp.org/en/jsr/detail?id=277
[16] Scala Language: http://www.scala-lang.org/