Immer in Bewegung – **Bundles** und Life Cycle

Wir haben uns im vorigen Java Magazin mit den drei herausragenden Eigenschaften von OSGi beschäftigt: Modularisierung, Laufzeitdynamik und Serviceorientierung. Jetzt ist es an der Zeit, Modularisierung und Laufzeitdynamik und deren Zusammenspiel unter die Lupe zu nehmen.

von Heiko Seeberger

Wozu brauchen wir eigentlich Modularisierung bzw. ein Module System? Die Antwort ist einfach: zur Reduktion von Komplexität. Module abstrahieren vom schwer überschaubaren Problem hin zu übersichtlichen Herausforderungen. Diese können isoliert voneinander und möglicherweise auch arbeitsteilig in Angriff genommen werden und bieten die Möglichkeit der Wiederverwendung.

Was bedeutet das konkret für Java? Als objektorientierte Programmiersprache bietet Java ein Modulkonzept in Form von Klassen und Packages. Allerdings ist diese Form der Modularisierung zu feingranular, um damit nicht-triviale Softwaresysteme adäquat zu modularisieren. Mit anderen Worten: Java fehlt ein Modulkonzept „oberhalb" der Packages.

Bevor wir betrachten, wie OSGi diese Lücke schließt, definieren wir zunächst, was wir eigentlich genau unter einem Modul verstehen wollen:

- *Self-contained*: Ein Modul ist eine Komposition aus kleineren Teilen, mit Ausnahme von wohl definierten Abhängigkeiten und kann nur als Ganzes verwendet werden.
- *Cohesive*: Die Elemente eines Moduls haben einen starken logischen Bezug zueinander.
- *Loose Coupling*: Module besitzen untereinander eine geringe Kopplung.
- *Public API*: Ein Modul besitzt eine wohl definierte öffentliche Schnittstelle und verbirgt die Interna der Implementierung.
- *Dependencies*: Die Abhängigkeiten eines Moduls sind wohl definiert.
- *Deployment Format*: Ein Modul „läuft" in einem Container bzw. einer Laufzeitumgebung und wird dort in einem wohl definierten Format installiert.

Diese Eigenschaften führen nicht nur dazu, dass komplexe Softwaresysteme in überschaubare Module heruntergebrochen werden können, wodurch zweifellos die Produktivität und Qualität in der Entwicklung gesteigert werden kann. Sie fördern auch die Flexibilität und wirken sich dadurch positiv auf

Artikelserie: OSGi in kleinen Dosen

- Teil 1: Erste Schritte mit OSGi
- **Teil 2: Immer in Bewegung – Bundles und Life Cycle**
- Teil 3: Was wünschen Sie? – Services a là OSGi
- Teil 4: Alles XML oder was? – Services auf deklarative Weise
- Teil 5: Hier wird „Service" groß geschrieben – Ausgewählte OSGi-Standardservices

Time-to-Market und die Nutzung neuer Marktchancen aus.

Beispiel und Entwicklungsumgebung

Wir beginnen mit einem Beispiel, das wir in den folgenden Teilen dieser Artikelserie sukzessive ausbauen werden. Dabei handelt es sich um ein „universelles Adressbuch“, das Kontakte aus beliebigen Repositories gemeinsam verwalten kann. Sämtlicher Sourcecode sowie die speziell benötigte Software befinden sich auf der Begleit-CD und können ebenfalls heruntergeladen werden [1].

Eclipse PDE wurde als Entwicklungswerkzeug für Bundles sowie Eclipse Equinox als OSGi-Implementierung bereits vorgestellt. Wir werden weiterhin Eclipse als Java-Entwicklungswerkzeug verwenden, aber diesmal mit Bnd [2] und Apache Felix [3], und Alternativen für die Entwicklung bzw. die OSGi-Implementierung aufzeigen.

Bnd wird von Peter Kriens, Director of Technology der OSGi Alliance [4], entwickelt und dient unter anderem zum Erstellen von OSGi Bundles. Mithilfe von *bnd*-Dateien, deren Syntax der des Bundle Manifests ähnelt, sowie der Analyse des Imports von Klassendateien erstellt Bnd aus Java-Projekten OSGi Bundles. Es kann per Kommandozeile als Ant-Task oder Eclipse-Plug-in verwendet werden. Wir werden es als Eclipse Plug-in einsetzen, indem wir die aktuelle produktive Version (zum Zeitpunkt der Erstellung dieses Artikels 0.0.249, als *bnd-0.0.249.jar* auf der Begleit-CD) in das Verzeichnis *dropins* unserer Eclipse-Installation kopieren und Eclipse anschließend neu starten. Danach stellt das Bnd-Plug-in im Kontextmenü von *bnd*-Dateien im Package Explorer den Menüpunkt MAKE BUNDLE zur Verfügung.

Um Apache Felix ebenfalls in Eclipse zu integrieren, erstellen wir ein Java-Projekt *org.apache.felix* und stellen in den Project Properties (Kontextmenü PROPERTIES) unter JAVA BUILD PATH | SOURCE den *Default output folder* auf *org.apache.felix* oder jeden beliebigen anderen Pfad außer *org.apache.felix/bin*. Anschließend entpacken wir den Download der aktuellen produktiven Version von Apache Felix (zum Zeitpunkt der Erstellung dieses Artikels 1.2.1) in das Projektverzeichnis und fügen die Datei *bin/felix.jar* unter JAVA BUILD PATH | LIBRARIES zu den Libraries hinzu. Dann exportieren wir diese Library unter JAVA BUILD PATH | ORDER AND EXPORT, sodass unsere Projekte auf die darin enthaltene OSGi-API Zugriff haben. Anschließend erzeugen wir mittels RUN | RUN CONFIGURATIONS … eine neue Run Configuration vom Typ *Java Application*, benennen Sie mit *org.apache.felix*, wählen *org.apache.felix.Main* als Main Class und setzen die folgenden VM Arguments: *-Dfelix.cache.dir=.cache -Dfelix.cache.profile=default*. Die Ausführung dieser Run Configuration startet Apache Felix, wobei per Default „Welcome to Felix.“ auf der Konsole ausgegeben wird. Die Begleit-CD enthält unter anderem bereits dieses Projekt, sodass wir dieses nur in unseren Workspace importieren müssen.

Modularisierung à la OSGi

Die OSGi-Spezifikation adressiert das Thema Modularisierung im Module Layer des OSGi Frameworks, der für dessen weitere Schichten die Grundlage bildet. Darin wird das Bundle als OSGi-Einheit für Modularisierung definiert. Ein Bundle besteht aus Java-Klassen und Ressourcen wie Properties-Dateien und wird als JAR-Archiv im OSGi Framework installiert (Abb. 1).

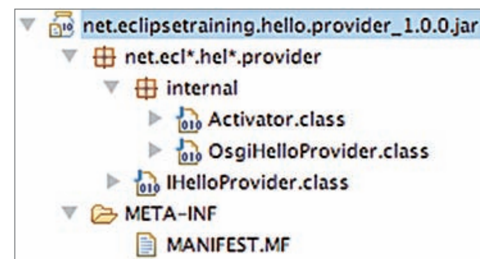

Abb. 1: Struktur eines Bundles

Darüber hinaus enthält ein Bundle Metadaten, z.B. über die öffentliche Schnittstelle oder die Abhängigkeiten, die vom OSGi Framework verwendet werden, um obige Eigenschaften sicherzustellen. Diese Metadaten werden im Bundle Manifest (*META-INF/MANIFEST.MF*) spezifiziert, das ohne-

Anzeige

hin Bestandteil der JAR-Spezifikation ist. Damit ist ein Bundle außerhalb des OSGi Frameworks ein ganz gewöhnliches JAR-Archiv und kann in jedem beliebigen Java-System verwendet werden. So sind z.B. sämtliche Libraries des Spring Frameworks seit Version 2.5 OSGi Bundles, was jedoch bei „klassischer" Verwendung völlig transparent bleibt.

Jedes Bundle besitzt einen obligatorischen symbolischen Namen, der mittels *Bundle-SymbolicName* spezifiziert wird. Wir empfehlen als Best Practice, bei der Benennung die Reverse-Domain-Name-Konvention zu verwenden, um Namenskollisionen zu vermeiden. Dabei beginnt der symbolische Name mit dem umgekehrten Domain-Namen und spezifiziert dann „Sinn und Zweck", z.B. *com.weiglewilczek.example.osgi.contacts.core*. Gemeinsam mit der Version, die mittels Bundle-Version gesetzt wird, identifiziert der symbolische Name ein Bundle eindeutig innerhalb des OSGi Frameworks. Daraus wird unmittelbar ersichtlich, dass es möglich ist, das gleiche Bundle (mit demselben symbolischen Namen) in mehreren unterschiedlichen Versionen zu installieren. Zusammen mit der weiter unten erläuterten Möglichkeit, bei Abhängigkeiten auf bestimmte Versionen abzuzielen, bietet OSGi damit einen Ausweg aus der leider allzu häufig auftretenden Java-Sackgasse, bestimmte Libraries gleichzeitig in verschiedenen Versionen zu benötigen.

Der lokale Klassenpfad eines Bundles kann aus mehreren Verzeichnissen innerhalb des Bundles bestehen und sogar eingebettete JAR-Archive enthalten. Wichtig: Es ist nicht möglich, Klassen oder JAR-Archive außerhalb des Bundles zu verwenden. Das OSGi Framework sorgt durch eine ausgeklügelte Classloading-Architektur dafür, dass Bundles self-contained sind und ihr lokaler Klassenpfad ausschließlich aus Elementen innerhalb des Bundles besteht. Wird der entsprechende Manifest Header *Bundle-Classpath* weggelassen, wird per Default das Wurzelverzeichnis des Bundles verwendet. Dies ist empfehlenswert, um Bundles als „ganz normale" JAR-Libraries verwenden zu können.

Unser Beispiel, 1. Iteration

Für unser Beispiel benötigen wir zwei Bundles:

- *com.weiglewilczek.example.osgi.contacts.core*: Das Kern-API des Adressbuchs.
- *com.weiglewilczek.example.osgi.contacts.core.inmemory*: Eine nicht persistente Implementierung des Kern-APIs.

Dazu erstellen wir je ein gleichnamiges Java-Projekt mit Eclipse. Das Kern-API-Bundle enthält im Package *com.weiglewilczek.example.osgi.contacts.core* die Bean Contact mit Properties für Vor- und Nachname, Anschrift etc.:

```
public class Contact {
  private String firstName;
  private String lastName;
  ...
}
```

Des Weiteren enthält es das Interface *ContactRepository* zur Verwaltung der Kontakte:

```
public interface ContactRepository {
  void add(Contact contact);
  Contact[] getAllContacts();
  void remove(Contact contact);
}
```

Das Projekt für das Implementierungs-Bundle benötigt natürlich eine Abhängigkeit zum Projekt für das Kern-API, sodass wir diese über die Project Properties (Java Build Path | Projects) hinzufügen. Anschließend erstellen wir ein *InMemoryContactRepository*, das *ContactRepository* implementiert, indem es alle *Contacts* in einer *Collection* verwaltet:

```
public class InMemoryContactRepository implements
                                        ContactRepository {
  private final Collection<Contact> contacts;
  ....
}
```

Die öffentliche Schnittstelle

Wie oben erläutert, kann ein Bundle a priori nur auf die „eigenen" Klassen zugreifen. Werden andere Klassen benötigt, so sind zweierlei Voraussetzungen zu erfüllen. Erstens müssen nutzende Bundles explizit ihre Abhängigkeiten im Bundle Manifest deklarieren (Import) und zweitens müssen Bundles, die anderen Bundles einen Teil ihrer Klassen als öffentliche Schnittstelle zur Verfügung stellen, dies ebenfalls im Bundle Manifest deklarieren (Export).

Mit dem Manifest Header Export Package werden die Namen sämtlicher Packages spezifiziert, die zur öffentlichen Schnittstelle gehören. Dabei werden die Package-Namen mit Kommata voneinander getrennt: *Export-Package: p1,p2*. Es ist möglich und empfehlenswert, pro Package eine Versionsnummer anzugeben, was durch das *version*-Attribut erzielt werden kann: *Export-Package: p;version="1.0.0"*. Um Namenskollisionen bei den exportierten Packages zu vermeiden, empfehlen wir für diese ebenfalls eine Namenskonvention: Das oberste Package sollte den symbolischen Namen des Bundles tragen, z.B. *com.weiglewilczek.example.osgi.contacts.core.inmemory*. Um darüber hinaus auf den ersten Blick erkennen zu können, was exportiert wurde und was nicht, bietet sich an, nicht-exportierte Packages mit einem *internal* im Namen zu versehen, z.B. *com.weiglewilczek.example.osgi.contacts.core.inmemory.internal*. Bei der Verwendung

Manifest Header	Bedeutung
Bundle-SymbolicName	Eindeutiger Name innerhalb des OSGi Frameworks. Obligatorisch! Empfehlung: Reverse Domain Name Convention
Bundle-Version	Dient zusammen mit dem symbolischen Namen als eindeutige ID. Format: „major.minor.micro.qualifier", z.B. "1.2.3.test". Default "0.0.0"
Bundle-Classpath	Listet alle Verzeichnisse und/oder eingebettete JARs auf, die zum lokalen Klassenpfad des Bundles gehören. Default „."
Bundle-Activator	Voll-qualifizierter Name eines BundleActivators, dessen Methoden beim Starten und Stoppen des Bundles aufgerufen werden
Import-Package	Listet die Abhängigkeiten in Form von Package-Namen auf
Export-Package	Listet die öffentliche Schnittstelle in Form von Package-Namen auf

Tabelle 1: Wichtige Manifest Header

von Eclipse PDE bekommen wir dadurch sogar den Vorteil, Compiler-Warnungen für Discouraged Access zu erhalten, wenn wir solche Packages importieren.

Unser Beispiel, 2. Iteration

Das Kern-API-Bundle muss natürlich das Package *com.weiglewilczek.example.osgi.contacts.core* als öffentliche Schnittstelle deklarieren. Wie oben erläutert, erstellen wir das Bundle Manifest nicht manuell, sondern verwenden Bnd als Eclipse-Plug-in, um aus Java-Projekten Bundles einschließlich Bundle Manifest zu erzeugen. Bnd ermittelt bestimmte Informationen für das Bundle Manifest, z.B. die zu importierenden Packages, durch Analyse der Klassendateien. Allerdings benötigt Bnd auch eine Reihe von Metadaten in Form einer *bnd*-Datei, die dem Bundle Manifest sehr ähnelt, aber hinsichtlich der Syntax leichter handzuhaben ist. Darüber hinaus gibt es zahlreiche Hilfsmittel, z.B. die Verwendung von Wildcards. Für das Kern-API-Bundle benötigen wir die folgende *bnd*-Datei, die wir im Wurzelverzeichnis des Projekts ablegen:

```
# com.weiglewilczek.example.osgi.contacts.core.bnd
Bundle-Version: 1.0.0
Export-Package: com.weiglewilczek.example.osgi.
                    contacts.core;version=1.0.0
```

Über Make bundle im Kontextmenü der *bnd*-Datei im Package Explorer können wir nun das Kern-API-Bundle erstellen. Es wird unter dem Namen der *bnd*-Datei ebenfalls im Wurzelverzeichnis des Projekts abgelegt und enthält das folgende Bundle Manifest:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.weiglewilczek.example.
                    osgi.contacts.core
Bundle-Version: 1.0.0
Export-Package: com.weiglewilczek.example.osgi.
                    contacts.core;version="
 1.0.0"
Import-Package: com.weiglewilczek.example.osgi.
                    contacts.core;version="
 1.0.0"
```

Abhängigkeiten

Für die Deklaration der Abhängigkeiten gibt es zwei Möglichkeiten. Mit dem Manifest Header *Import-Package* werden die Namen von Packages spezifiziert, die vom Bundle benötigt werden. Alternativ kann der Manifest Header *Require-Bundle* verwendet werden, mit dem benötigte Bundles spezifiziert werden, von denen alle exportierten Packages genutzt werden können. Allerdings hat *Require-Bundle* mehrere Nachteile, insbesondere wird das Prinzip der losen Kopplung verletzt. Beispielsweise wäre es in den meisten Fällen unsinnig, wenn sich ein Bundle ganz konkret von einem MySql-Bundle abhängig macht, nur weil es das JDBC API benötigt. Mit *Import-Package* hingegen haben wir die Möglichkeit, die Abhängigkeiten auch durch andere Bundles zu erfüllen, z.B. mit einem Oracle-, einem Derby- oder einem anderen Bundle, das das JDBC API exportiert. Wir werden daher im Folgenden ausschließlich *Import-Package* verwenden. Analog zu Export-Package werden die Namen der benötigten Packages kommasepariert aufgelistet: *Import-Package: p1,p2,p3*.

Mit dem *version*-Attribut werden Versionsintervalle spezifiziert. Dabei stehen eckige Klammern für geschlossene und runde Klammern für offene Intervalle, d.h. im ersten Fall gehört die betroffene Versionsnummer noch zum Intervall und im zweiten Fall nicht:

```
Import-Package: p;version="[1.0.0,2.0.0)"
```

Es ist auch möglich, nur eine Versionsnummer anzugeben, was einem nach oben offenen Intervall entspricht, sodass alle Versionsnummern größer oder gleich sind:

```
Import-Package: p;version="1.0.0"
```

Wozu dienen nun die Versionsnummern? Das OSGi Framework versucht, zu einem bestimmten Zeitpunkt im Lebenszyklus von Bundles deren Abhängigkeiten aufzulösen, indem zu jedem Package-Import ein passender Package-Export gesucht wird. Dabei werden natürlich die Versionsnummern berücksichtigt, sofern sie spezifiziert wurden. Es gibt noch einige weitere Details hinsichtlich der Auflösung von Abhängigkeiten, z.B. optionale Imports, spezifische Attribute etc. Für diese sei auf die hervorragende OSGi-Spezifikation [5] verwiesen.

Unser Beispiel, 3. Iteration

Um das Implementierungs-Bundle zu erstellen, benötigen wir zunächst die folgende *bnd*-Datei:

```
# com.weiglewilczek.example.osgi.contacts.core.
                              inmemory.bnd
Bundle-Version: 1.0.0
Private-Package: com.weiglewilczek.example.osgi.
                    contacts.core.inmemory.internal
```

Da wir hier kein Package exportieren, müssen wir Bnd mittels *Private-Package* mitteilen, welche Packages zum Bundle gehören. Alternativ zur expliziten Angabe von Namen könnte auch die Wildcard „*“ verwendet werden. Um die zu importierenden Packages müssen wir uns keine Gedanken machen, weil Bnd diese anhand der Klassendateien analysiert. Das resultierende Bundle Manifest sieht folgendermaßen aus.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.weiglewilczek.example.
                    osgi.contacts.core.inme
 mory
Bundle-Version: 1.0.0
Import-Package: com.weiglewilczek.example.osgi.
                    contacts.core,org.osgi.
 framework;version="1.4"
```

Bundle Life Cycle

Das OSGi Framework stellt die Laufzeitumgebung für Bundles dar: Bundles können installiert, aktualisiert und wieder deinstalliert werden. Zudem können sie gestartet und wieder gestoppt werden. Abbildung 2 zeigt die Zustände, die ein Bundle während seines Lebenszyklus innerhalb des OSGi Frameworks einnehmen kann.

Bevor wir auf die Frage eingehen, wie der Lebenszyklus gesteuert werden kann, widmen wir uns der Bedeutung der einzelnen Zustände und deren Übergängen. Durch das Installieren eines Bundles wird dieses innerhalb des OSGi Frameworks persistiert, d.h. in einem lokalen Bundle Cache abgelegt.

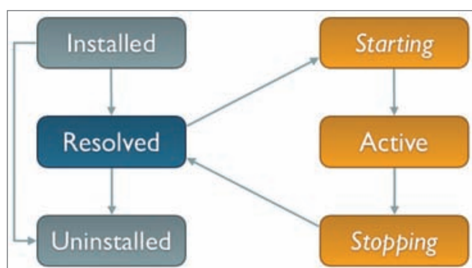

Abb. 2: Zustände innerhalb des Bundle Life Cycle

Die Voraussetzung dafür ist, dass es ein gültiges Format und gültige Metadaten enthält. Bei der Installation wird dem Bundle eine eindeutige ID vom Typ *long* zugewiesen und das Bundle wird in den Zustand *INSTALLED* versetzt.

Dem OSGi Framework steht offen, ob es sofort nach der Installation eines Bundles oder erst später versucht, dessen Abhängigkeiten aufzulösen. Eclipse Equinox und Apache Felix z.B. verhalten sich hier lazy, um Ressourcen zu schonen. Beim so genannten Resolving versucht das OSGi Framework für jedes zu importierende Package ein passendes exportiertes Package zu finden. Nur wenn dies für alle nicht optionalen Package-Imports gelingt, wird das Bundle in den Zustand *RESOLVED* versetzt, andernfalls bleibt es *INSTALLED* und kann nicht verwendet werden, bis möglicherweise zu einem späteren Zeitpunkt alle Abhängigkeiten aufgelöst werden können.

Falls mehrere passende exportierte Packages vorliegen, wird das mit der höchsten Version verwendet. Falls immer noch kein eindeutiger Bezug hergestellt werden kann, wird die Version der exportierenden Bundles verglichen und falls auch dies ohne eindeutiges Ergebnis bleibt, wird das Bundle mit der kleinsten ID ausgewählt. Auf diese Weise kann vom OSGi Framework immer eine eindeutige Zuordnung zwischen einem importierenden und einem exportierenden Bundle getroffen werden: Dies wird Wiring genannt. Es ist jedoch empfehlenswert, mittels Versionsnummern und gegebenenfalls weiterer Attribute möglichst genaue Vorgaben zu treffen, um „selbst die Kontrolle zu behalten".

Wichtig: Bundles im Zustand *RESOLVED* können bereits verwendet werden, indem andere Bundles ihr API benutzen. Es ist nicht erforderlich, jedes Bundle zu starten, denn dies dient einem besonderen Zweck: Mit dem Manifest Header *Bundle-Activator* wird der voll qualifizierte Name einer Klasse angegeben, die das Interface *BundleActivator* implementiert:

```
public interface BundleActivator {
  void start(BundleContext context);
  void stop(BundleContext context);
}
```

Wenn ein Bundle einen Activator spezifiziert, wird beim Starten vom OSGi Framework eine Instanz erzeugt und dessen *start()*-Methode und beim Stoppen dessen *stop()*-Methode aufgerufen. Da diese Aufrufe synchron vom Framework Thread erfolgen, ist es wichtig, dass sie so schnell wie möglich zurückkehren, um die Ausführung des Systems nicht zu blockieren. Langlaufende Vorgänge sollten daher in eigenen Threads ausgeführt werden.

Beim Starten wird das Bundle zunächst in den Zustand *STARTING* versetzt. Wenn ein Activator spezifiziert ist, wird dessen *start()*-Methode aufgerufen und wenn diese erfolgreich zurückkehrt, wird das Bundle auf *ACTIVE* gesetzt, andernfalls wieder auf *RESOLVED*. Analog verhält es sich mit dem Stoppen. Um die Startup-Performance zu verbessern, kann ein Bundle mit dem Manifest Header *Bundle-ActivationPolicy* spezifizieren, dass die Aktivierung *lazy* erfolgen soll. Dann wird der Aufruf der *start()*-Methode solange verzögert und das Bundle verbleibt solange im Zustand *STARTING*, bis zum ersten Mal eine Klasse aus dem Bundle geladen wird.

Wenn ein Bundle deinstalliert wird, geht es zunächst in den Zustand *UNINSTALLED* über. Sofern es keine Packages exportiert, die im Wiring verwendet wurden, wird es sofort aus dem persistenten Speicher des OSGi Frameworks gelöscht. Andernfalls bleiben die Package-Exports bis zu einem Neustart des OSGi Frameworks erhalten, sodass durch das Deinstallieren eines Bundles nicht automatisch alle abhängigen Bundles funktionsunfähig werden.

Beim Aktualisieren wird ein Bundle erneut eingelesen und persistiert. Dabei bleibt die Bundle ID erhalten, wohingegen durch Deinstallieren und anschließendes erneutes Installieren eine neue Bundle ID vergeben wird. Das OSGi Framework versetzt das Bundle beim Aktualisieren wieder in den Ausgangszustand, sodass z.B. ein ursprünglich gestartetes Bundle zunächst gestoppt, dann erneut eingelesen und aufgelöst und danach wieder gestartet wird. Auf diese Weise kann ein Softwaresystem im laufenden Betrieb feingranular – nämlich auf Ebene von Bundles – aktualisiert werden. Mit anderen Worten bietet OSGi ein Hot Deployment auf Modulebene.

Befehl	Bedeutung
Ps	Listet die installierten Bundles mit Basisinformationen auf
install <url>	Installiert das Bundle von der angegebenen URL
update <id>	Aktualisiert das angegebene Bundle
resolve <id>	Löst das angegebene Bundle auf
refresh <id>	Führt ein Package Refresh für das angegebene Bundle
start <id>	Startet das angegebene Bundle
stop <id>	Stoppt das angegebene Bundle
uninstall <id>	Deinstalliert das angegebene Bundle
Shutdown	Beendet das OSGi Framework
Help	Listet die verfügbaren Befehle mit Erläuterungen auf

Tabelle 2: Wichtige Befehle der Apache Felix Text Shell

Life Cycle Management

Wie kann nun der Lebenszyklus von Bundles gesteuert werden? Dazu bietet das OSGi Framework ein schlankes API an, das im Wesentlichen aus den folgenden drei Klassen besteht: *BundleContext*, *Bundle*, *BundleListener*.

Der *BundleContext* stellt die einzige Schnittstelle zur Interaktion mit dem OSGi Framework dar. Um eine Referenz auf den *BundleContext* zu erhalten,

muss ein Activator verwendet werden, in dessen *start()*- und *stop()*-Methoden der *BundleContext* für das jeweilige Bundle übergeben wird. Mit dessen *installBundle()*-Methoden kann ein Bundle im OSGi Framework installiert werden. Als Rückgabe erhalten wir eine Referenz vom Typ *Bundle*, mit der wir den Lebenszyklus des Bundles steuern können, z.B. mit den Methoden *start()*, *stop()*, *update()* und *uninstall()*. Um den Lebenszyklus beliebiger Bundles zu verfolgen und gegebenenfalls darauf zu reagieren, können *BundleListener* über den *BundleContext* beim OSGi Framework registriert werden. An diese werden *BundleEvents* versendet, die Informationen über die Änderung im Lebenszyklus enthalten.

Wichtig: Es gibt kein API im OSGi Framework zur Steuerung des Resolving. Dazu gibt es einen optionalen Standardservice, den Package Admin Service, der Operationen wie *refreshPackages()* und *resolveBundles()* anbietet. Sowohl Eclipse Equinox als auch Apache Felix enthalten diesen Service.

Für das Life Cycle Management bieten die OSGi-Implementierungen in der Regel so genannte Management Agents als Bestandteil der Implementierung oder als separate Bundles an. Eclipse Equinox enthält die Equinox Console und Apache Felix bringt per Default die Bundles *Shell* und *Shell Text UI* mit. Beides sind textbasierte Management Agents mit sehr ähnlichen Funktionen. Darüber hinaus gibt es weitere Management Agents auf Basis von Swing oder webbasierte Agents. Letztendlich kann jedes Softwaresystem auf Basis von OSGi einen eigenen Management Agent in Form eines oder mehreren Bundles implementieren. Da wir für unser Beispiel Apache Felix verwenden, stellen wir in Tabelle 2 die wichtigsten Befehle für dessen Text Shell vor.

Unser Beispiel, 4. Iteration

Damit unser Beispiel bereits in diesem Stadium aktiv werden kann, fügen wir einen Activator (Listing 1) hinzu, der in der *start()*- und *stop()*-Methode Tracing-Ausgaben nach *System.out* schreibt. Weiter er-

Listing 1

```
public class Activator implements BundleActivator {

  public void start(final BundleContext context) {
    System.out.println(MessageFormat.format("[{0}] starting ...",
                                                                          context

        .getBundle().getSymbolicName()));
    final ContactRepository repository = createRepository();
    for (final Contact contact : repository.getAllContacts()) {
      System.out.println(contact);
    }
  }

  public void stop(final BundleContext context) {
    System.out.println(MessageFormat.format("[{0}] stopping ...",
                                                                          context

        .getBundle().getSymbolicName()));
  }

  private ContactRepository createRepository() {
    ...
  }
```

Anzeige

```
Welcome to Felix.
=================

-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (1.2.1)
[   1] [Active     ] [    1] Apache Felix Shell Service (1.0.2)
[   2] [Active     ] [    1] Apache Felix Shell TUI (1.0.2)
[   3] [Active     ] [    1] Apache Felix Bundle Repository (1.2.0)
```

Abb. 3: Installierte Bundles auflisten

```
-> install file:../com.weiglewilczek.example.osgi.contacts.core/com.weiglewilczek.example.
Bundle ID: 4
-> install file:../com.weiglewilczek.example.osgi.contacts.core.inmemory/com.weiglewilczek
Bundle ID: 5
-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (1.2.1)
[   1] [Active     ] [    1] Apache Felix Shell Service (1.0.2)
[   2] [Active     ] [    1] Apache Felix Shell TUI (1.0.2)
[   3] [Active     ] [    1] Apache Felix Bundle Repository (1.2.0)
[   4] [Installed  ] [    1] com.weiglewilczek.example.osgi.contacts.core (1.0.0)
[   5] [Installed  ] [    1] com.weiglewilczek.example.osgi.contacts.core.inmemory (1.0.0)
```

Abb. 4: Bundles installieren

```
-> resolve 5
-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (1.2.1)
[   1] [Active     ] [    1] Apache Felix Shell Service (1.0.2)
[   2] [Active     ] [    1] Apache Felix Shell TUI (1.0.2)
[   3] [Active     ] [    1] Apache Felix Bundle Repository (1.2.0)
[   4] [Resolved   ] [    1] com.weiglewilczek.example.osgi.contacts.core (1.0.0)
[   5] [Resolved   ] [    1] com.weiglewilczek.example.osgi.contacts.core.inmemory (1.0.0)
```

Abb. 5: Bundles auflösen

```
-> start 5
[com.weiglewilczek.example.osgi.contacts.core.inmemory] starting ...
John Doe
Max Mustermann
```

Abb. 6: Bundles starten

```
-> update 5
[com.weiglewilczek.example.osgi.contacts.core.inmemory] stopping ...
[com.weiglewilczek.example.osgi.contacts.core.inmemory] starting ...
Another One
John Doe
Max Mustermann
```

Abb. 7: Bundles aktualisieren

zeugen wir beim Starten eine Instanz des *InMemoryContactRepository* und geben dessen Kontakte nach *System.out* aus.

Die zugehörige *bnd*-Datei muss nun noch um die *Bundle-Activator*-Direktive erweitert werden, die – abgesehen von der Umformatierung in das spezielle Manifest-Format – in das Bundle Manifest übernommen wird:

```
# com.weiglewilczek.example.osgi.contacts.core.
                                    inmemory.bnd
Bundle-Version: 1.0.0
Private-Package: com.weiglewilczek.example.osgi.
                   contacts.core.inmemory.internal
Bundle-Activator: \
 com.weiglewilczek.example.osgi.contacts.core.
                    inmemory.internal.Activator
```

Das resultierende Bundle Manifest sieht nun folgendermaßen aus:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.weiglewilczek.example.
                        osgi.contacts.core.inme
 mory
Bundle-Version: 1.0.0
Import-Package: com.weiglewilczek.example.osgi.
                     contacts.core,org.osgi.
 framework;version="1.4"
Bundle-Activator: com.weiglewilczek.example.osgi.
                      contacts.core.inmemor
 y.internal.Activator
```

Nun ist es an der Zeit, unsere beiden Bundles im OSGi Framework zu installieren, zu starten, zu aktualisieren etc. Dazu starten wir zunächst Apache Felix mittels der Run Configuration *org.apache.felix* aus dem gleichnamigen Projekt. In der Konsole geben wir *ps* ein, um die aktuell installierten Bundles aufzulisten (Abb. 3).

Anschließend verwenden wir das Kommando *install*, um die Bundles zu installieren und listen danach wieder die installierten Bundles auf (Abb. 4).

Unsere beiden neu installierten Bundles befinden sich nun im Zustand *INSTALLED*. Um sie aufzulösen, geben wir das Kommando *resolve* mit der ID des Implementierungs-Bundles ein. Dadurch wird auch das Kern-API-Bundle aufgelöst, um die Abhängigkeiten erfüllen zu können, d.h. unsere beiden Bundles befinden sich nun im Zustand *RESOLVED* (Abb. 5).

Nun starten wir das Implementierungs-Bundle mit dem Kommando *start*. Dadurch wird der Activator instanziiert und dessen *start()*-Methode ausgeführt, sodass die Tracing-Ausgabe sowie die Liste der Kontakte ausgegeben werden (Abb. 6).

Abschließend zeigen wir noch eines der OSGi-Highlights schlechthin: Wir aktualisieren das Implementierungs-Bundle im laufenden Betrieb. Dazu passen wir dessen Activator an, sodass er eine andere Menge von Kontakten enthält, lassen Bnd das Bundle erneut erstellen und führen dann in der Konsole das Kommando *update* aus.

Wie Abbildung 7 zeigt, wird das Bundle zunächst gestoppt, dann transparent aktualisiert und anschließend wieder gestartet. Und tatsächlich finden wir einen neuen Kontakt in der Ausgabe.

Schlussbemerkung und Ausblick

Wir haben die wichtigsten Details des Module und Life Cycle Layer kennengelernt: Bundles sind JAR-Archive mit zusätzlichen Metadaten. Diese werden im Bundle Manifest spezifiziert und deklarieren u.a. die öffentliche Schnittstelle und die Abhängigkeiten eines Bundles. Das OSGi Framework bietet Bundles eine Laufzeitumgebung, wo diese u.a. installiert, gestartet und aktualisiert werden können. Dazu dient ein Management Agent der OSGi-Implementierung, z.B. eine textbasierte Konsole. Im nächsten Teil dieser Artikelserie werden wir die dritte wesentliche Eigenschaft von OSGi unter die Lupe nehmen: Das OSGi Service Model. ■

Heiko Seeberger ist als Technical Director für die Weigle Wilczek GmbH tätig. Sein technischer Schwerpunkt liegt in der Entwicklung von Unternehmensanwendungen mit OSGi, Eclipse RCP, Spring, AspectJ und Java EE. Seine Erfahrungen aus über zehn Jahren IT-Beratung und Softwareentwicklung fließen in die Eclipse Training Alliance ein. Zudem ist Heiko Seeberger aktiver Committer in Eclipse-Projekten, Autor zahlreicher Fachartikel und Redner auf einschlägigen Konferenzen.

Links & Literatur

[1] WeigleWilczek-Examples: www.weiglewilczek.com/examples/

[2] Bnd: www.aqute.biz/Code/Bnd

[3] Apache Felix: felix.apache.org

[3] OSGi Alliance: www.osgi.org

[3] Manifest Format: java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html

[3] OSGi-Spezifikation: www.osgi.org/Specifications/

[3] Seeberger, OSGi in kleinen Dosen, Teil, Java Magazin 12.08