

# Services auf deklarative Weise

Die OSGi Service Platform (OSGi) hat sich zu einem sehr bedeutenden Standard im Java-Umfeld entwickelt. Also wird es für den engagierten Java-Entwickler allerhöchste Zeit, sich damit näher auseinanderzusetzen.

von Heiko Seeberger

In den ersten drei Artikeln dieser Serie über die technischen Grundlagen von OSGi wurden die drei wesentlichen OSGi-Prinzipien kennengelernt: Modularisierung, Laufzeitdynamik und Services. Aufbauend auf diesem Fundament bieten OSGi Declarative Services (DS) einen Ansatz zur Vereinfachung des Umgangs mit dynamischen Services.

## Wozu ein deklarativer Ansatz?

Das OSGi Service Model aus dem letzten Artikel ermöglicht Kollaboration zwischen Bundles, ohne diese starr aneinanderzukoppeln. Damit ein Bundle Services anbieten oder nutzen kann, muss es gestartet werden und einen

BundleActivator bereitstellen. Dessen Methode *start()* bietet Zugriff auf den BundleContext, mit dem z. B. Services an der OSGi Service Registry registriert oder ServiceTracker zu Beobachtung und Nutzung von Services erzeugt werden können. Dieser programmatische Ansatz ist zwar sehr flexibel und für „kleine“ Systeme hervorragend geeignet, kann jedoch die folgenden beiden Nachteile mit sich bringen:

- Hohe Komplexität
  - Lange Start-up-Dauer und hoher Speicherverbrauch
- Erstens kann der Code, der zum Umgang mit dynamischen Services nötig

ist, rasch eine hohe Komplexität erreichen. Wenn beispielsweise ein Service andere zwingend benötigt, dann müssen die erforderlichen Services beobachtet und der abhängige je nach deren Verfügbarkeit registriert bzw. deregistriert werden. Diese Komplexität kann sich insbesondere bei „größeren“ Projekten nachteilig bemerkbar machen. Zweitens werden beim programmatischen Ansatz Services gleich beim Starten der Bundles erzeugt und registriert, ganz unabhängig davon, ob sie überhaupt jemals benötigt werden. Gerade bei Systemen, die aus zahlreichen Bundles bestehen, kann dies leicht zu einer allzu hohen Start-up-Dauer führen. Natürlich stellt auch der möglicherweise unnötige Speicherverbrauch für Systeme, die mit den verfügbaren Ressourcen schonend umgehen müssen, ein ernsthaftes Problem dar.

Die Declarative Services Specification [1] aus dem Service Compendium der OSGi-Spezifikation adressiert genau diese beiden Nachteile: Durch ein deklaratives Service Component Model wird die Handhabung von OSGi Services vereinfacht, und durch die Möglichkeit von Delayed Components können Start-up-

## 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 à la OSGi

**Teil 4: Services auf deklarative Weise**

Teil 5: Hier wird „Service“ groß geschrieben – Ausgewählte OSGi-Standardservices

Quellcode  
auf CD

Dauer und Speicherverbrauch optimiert werden.

Bevor wir uns im Detail mit OSGi Declarative Services beschäftigen, sei erwähnt, dass es noch weitere vergleichbare Ansätze gibt. Ein sehr populärer Vertreter ist Spring Dynamic Modules [2], das gemäß der aktuellen Early-Draft-Version der OSGi-Spezifikation 4.2 [3] als RCF 124 Blueprint Service standardisiert werden wird. Weitere Ansätze sind z. B. Apache Felix iPOJO [4] und Guice Peaberry [5]. All diesen gemeinsam ist, dass sie keine proprietären Mechanismen zur Kollaboration verwenden, sondern auf dem OSGi Service Model aufbauen. Dadurch sind sie untereinander und auch mit dem programmatischen Ansatz vollständig kompatibel, sodass in einem System durchaus mehrere dieser Ansätze gleichzeitig verwendet werden können.

### Beispiel und Entwicklungsumgebung

Das etablierte Beispiel des „universellen Adressbuchs“ aus den letzten Artikeln wird zum Teil auf OSGi Declarative Services umgestellt. Dazu wird das bereits existierende Bundle *com.weiglewilczek.example.osgi.contacts.shell* angepasst, und zwar wird ein eigenes Kommando für die Equinox Console hinzugefügt, indem das Interface *CommandProvider* implementiert und als Service registriert wird. Über die Methode *\_listAll()*, deren Signatur zwingend mit einem Unterstrich beginnen muss, wird das Kommando mit dem abgeleiteten Namen *listAll* für die Equinox Console verfügbar gemacht. Wie der Name erahnen lässt, sollen für alle verfügbaren *ContactRepository*s alle *Contacts* ausgegeben werden. Alles, was hierbei mit Services zu tun hat, wird mittels OSGi Dynamic Services umgesetzt. Dazu benötigen wir einige weitere Bundles in unserer Target Platform:

- *org.eclipse.osgi.services*: API für OSGi Standard Services, u. a. DS
- *org.eclipse.equinox.ds* und *org.eclipse.equinox.util*: Equinox-Implementierung für DS

Wie immer finden Sie den kompletten Sourcecode des Beispiels einschließlich

der kompletten Target Platform auf der Begleit-CD oder als Download [6].

### Service Components

Wie sieht nun der deklarative Ansatz des Service Component Models aus? Abbildung 1 zeigt exemplarisch zwei aktive Bundles, die je eine Service Component enthalten. Da Declarative Services, wie erwähnt, auf dem OSGi Service Model aufsetzen, gilt analog, dass Bundles gestartet werden müssen, damit ihre Service Components erzeugt werden. Allerdings ist kein *BundleActivator* mehr erforderlich, und die tatsächliche Erzeugung kann verzögert erfolgen, doch dazu später mehr.

Service Components bestehen aus einer XML-Beschreibung (Component Description) und einem Objekt (Component Instance). Sie können OSGi Services sowohl bereitstellen als auch referenzieren. Die Component Description enthält alle Informationen über die Service Component, z. B. den Klassennamen für die Component Instance, die Service Interfaces für die bereitgestellten Services etc.

So weit, so gut, doch wer erzeugt eigentlich die Service Components? Dazu kommt ein besonderes Pattern zum Einsatz: Das Extender Model [7]. Darin gibt es ein spezielles Bundle, das alle anderen Bundles beobachtet, analysiert und anhand von gewissen Kriterien Aktionen im Namen dieser Bundles durchführt. Im konkreten Fall handelt es sich beim Extender Bundle um die so genannte Service Component Runtime (Abb. 2). Diese prüft für alle gestarteten Bundles, ob deren Bundle Manifest den Manifest Header *Service Component* enthält, der eine oder auch mehrere Component Descriptions referenziert, z. B.:

```
Service-Component: OSGI-INF/component.xml
```

Aufgrund der Informationen in den Component Descriptions erzeugt die Service Component Runtime die Service Components für diese „DS-powered“ Bundles. Eine minimale Component Description enthält nur deren Namen, der global eindeutig sein muss, sowie den Klassennamen der Component Instance:

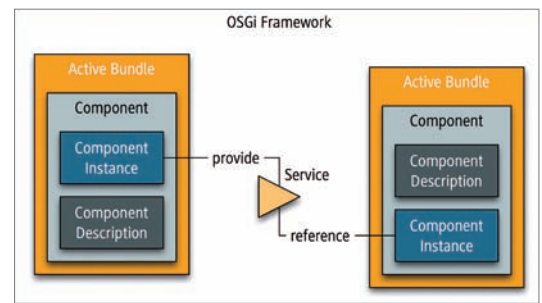


Abb. 1: OSGi Service Component Model

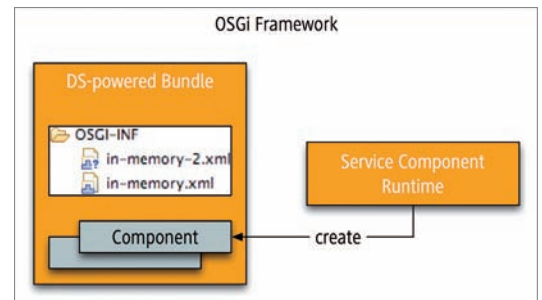


Abb. 2: Service Component Runtime

```
<component name="com.weiglewilczek...shell">
  <implementation class="com.weiglewilczek...
                                     Component" />
</component>
```

Oft gibt es pro Bundle nur eine Service Component, und dann ist es ein gängiges Pattern [8], wie in unserem Beispiel die Klasse mit *Component* und die Component Description mit *component.xml* zu bezeichnen.

### Services bereitstellen

Eine solche minimale Service Component ist natürlich ohne praktische Relevanz. Interessant wird es, sobald die Service Component einen Service bereitstellt. Dazu dient in der Component Description das Element *service* mit einem oder mehreren Subelementen *provide* für die Angabe von Service Interfaces:

```
<component immediate="false" ...>
  ...
  <service>
    <provide interface="org.eclipse...
                                     CommandProvider" />
  </service>
</component>
```

Service Components, die einen Service bereitstellen, sind standardmäßig De-

layed Components. Das bedeutet, dass sie nicht gleich erzeugt werden, sondern an ihrer Stelle ein Proxy in der Service Registry registriert wird. Erst wenn der bereitgestellte Service erstmalig verwendet werden soll, wird eine Delayed Component erzeugt. Dieses Verhalten kann über das Attribut *immediate* des Elements *component* modifiziert werden.

In unserem Beispiel stellt die Service Component im Bundle *com.weiglewilczek.example.osgi.contacts.shell* den Service *CommandProvider* zur Verfügung. Um ganz nach POJO-Prinzip die eigentlichen Domänenobjekte frei von OSGi-API zu halten, bietet es sich an, nicht die Domänenobjekte selbst als Component Instances zu verwenden, sondern Wrapper einzusetzen [8]. In unserem Beispiel ist das „Domänenobjekt“ die Klasse *ListAllCommand*, die Abhängigkeiten auf die registrierten *ContactRepository Services* hat. Diese Serviceabhängigkeiten werden von der Service Component verwaltet, also von der Klasse *Component*. So kann *ListAllCommand* frei von OSGi-API bleiben, wengleich in diesem einfachen Beispiel natürlich spezifischer Code für die Equinox Console enthalten ist.

### Services referenzieren

Ihre volle Stärke entfalten Service Components, wenn sie OSGi Services referenzieren. Dabei gibt es zahlreiche Optionen, z. B. Look-up- oder Event-Strategie, statische oder dynamische Policy etc. Für diese Details sei auf die Spezifikation verwiesen. In allen Fällen wird zum Referenzieren von OSGi Services das Element *reference* verwendet:

```
<component ...>
...
<reference name="contactRepositories"
  interface="com.weiglewilczek...
  ContactRepository"
  cardinality="1..n"/>
</component>
```

Das Attribut *interface* spezifiziert das Service Interface und mit dem optionalen Attribut *target* könnte auch noch ein Filter-Ausdruck angegeben werden. Besondere Bedeutung genießt die Kardinalität, die mittels *cardinality*

spezifiziert wird. Damit wird zwischen einfachen und mehrfachen sowie optionalen und obligatorischen Referenzen unterschieden. Wenn eine Referenz mittels „1..1“ oder „1..n“ als obligatorisch deklariert wird, wird die Service Component frühestens dann aktiviert, wenn die Referenz erfüllt werden kann, d. h. wenn ein passender Service registriert wurde. Weiter wird, falls die Service Component einen Service anbietet, dieser bzw. dessen Proxy erst dann in der Service Registry registriert, wenn die Referenz erfüllt wird. Umgekehrt wird die Service Component deaktiviert und ein allfälliger Service deregistriert, wenn die Referenz nicht mehr erfüllt werden kann. Über das Attribut *name* wird ein lokaler Name spezifiziert, über den im Code der Component Instance auf die Referenz zugegriffen werden kann, je nach Kardinalität mittels *ComponentContext.locateService()* oder *ComponentContext.locateServices()*. Der dafür erforderliche *ComponentContext* wird in der optionalen Methode *activate()* übergeben, die nicht Bestandteil eines Interfaces ist, sondern per Reflection gefunden wird und gemäß Spezifikation *public* oder *protected* sein muss. In diesem Beispiel wird eine obligatorische Referenz auf mehrere *ContactRepository Services* verwendet. Das bedeutet, dass unsere Component erst dann ak-

tiviert wird, wenn mindestens ein *ContactRepository Service* vorhanden ist. Gleichzeitig wird unsere Component auch erst dann als *CommandProvider Service* registriert. Insgesamt vereinfacht sich der Code durch die Verwendung von OSGi Declarative Services (Listing 2) im Vergleich zum programmatischen Ansatz (Listing 1) erheblich.

### Schlussbemerkung und Ausblick

Die Spezifikation der OSGi Declarative Services bietet noch etliche Details, z. B. zum Lebenszyklus von Service Components, zu Referenzen, zu Factory Components etc. Doch schon diese kurze Einführung zeigt, wie wir von den Vereinfachungen im Umgang mit OSGi Services profitieren können, die das Service Component Model bietet. Abschließend soll nochmals ausdrücklich betont werden, dass OSGi Declarative Services ohne Einschränkung mit dem programmatischen Ansatz oder auch mit anderen OSGi-Komponentenmodellen kombiniert werden können, sodass deren Einsatz keinerlei Einschränkung für die Flexibilität des Gesamtsystems darstellt. Im nächsten Artikel werden weitere ausgewählte OSGi Standard Services betrachtet und damit diese Einführung in die Grundlagen von OSGi abgeschlossen. ■



**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] OSGi-Spezifikation: [www.osgi.org/Specifications/](http://www.osgi.org/Specifications/)
- [2] Spring Dynamic Modules: [www.springsource.org/osgi/](http://www.springsource.org/osgi/)
- [3] OSGi 4.2, Early Draft 2: [www.osgi.org/download/osgi-4.2-early-draft2.pdf](http://www.osgi.org/download/osgi-4.2-early-draft2.pdf)
- [4] Apache Felix iPOJO: [felix.apache.org/site/apache-felix-ipojo.html](http://felix.apache.org/site/apache-felix-ipojo.html)
- [5] Guice Peaberry: [code.google.com/p/peaberry/](http://code.google.com/p/peaberry/)
- [6] WeigleWilczek-Examples: [www.weiglewilczek.com/examples/](http://www.weiglewilczek.com/examples/)
- [7] Extender Model: [www.osgi.org/blog/2007/02/osgi-extender-model.html](http://www.osgi.org/blog/2007/02/osgi-extender-model.html)
- [8] Buch „Equinox and OSGi“: [equinoxosgi.blogspot.com](http://equinoxosgi.blogspot.com)