

Immer in Bewegung – Services à la OSGi

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

Nachdem in der fünfteiligen Serie die technischen Grundlagen von OSGi sowie Modularisierung und Laufzeitdynamik unter die Lupe genommen wurden, geht es diesmal im Detail um das OSGi Service Model, der dritten wesentlichen Eigenschaft von OSGi.

Services und lose Kopplung

Im letzten Artikel haben wir Modularisierung als ein Mittel zur Reduktion von Komplexität und letztendlich zur Steigerung von Produktivität, Flexibilität und Qualität in der Softwareentwicklung gepriesen. Allerdings bestehen die Lösungen für die Herausforderungen, mit denen man sich in der Praxis konfrontiert sieht, in der Regel aus mehreren oder gar zahlreichen Modulen. Diese müssen miteinander kollaborieren, sodass nach der Zerlegung in überschaubare Häppchen wieder die Zusammenführung ansteht. Dabei stellt sich natürlich

die Frage, wie man diese gestalten kann, sodass die Vorteile der Modularisierung erhalten bleiben. Schließlich wäre nichts gewonnen, wenn die Module quasi zusammengeschweißt würden, sodass sie untrennbar miteinander verbunden sind. Es gilt also, die Architektur und das Design so zu wählen, dass eine lose Kopplung der Module erzielt wird.

OSGi beantwortet diese Frage mit dem OSGi Service Model (Abb. 1). Das OSGi Framework stellt eine Service Registry zur Verfügung, an der Bundles Services registrieren können. Dabei sind Services Instanzen gewöhnlicher Klassen, also POJOs (Plain Old Java Objects). Zur Registrierung dient das Service Interface, ein „gewöhnlicher“ Java-Typ, in der Regel ein Interface. Dieses wird ebenso verwendet, um Services von der Service Registry zu erfragen. Um der OSGi-Dynamik Rechnung zu tragen, gibt es in Form von *ServiceListeners* die Möglichkeit, auf Registrieren oder De-

registrieren von Services zu reagieren. Wie trägt nun das OSGi Service Model zur losen Kopplung bei? Zum einen abstrahiert die Verwendung von Service Interfaces von der konkreten Implementierung, sodass die Anbieter von Services austauschbar sind. Zum anderen ermöglicht die Indirektion durch die Service Registry, dass Services genutzt werden können, ohne dass das nutzende Bundle mit deren Erzeugung zu tun hätte und somit unabhängig von konkreten Anbietern bleibt.

Beispiel und Entwicklungsumgebung

Wir erweitern heute das Beispiel des „universellen Adressbuchs“ aus dem letzten Artikel um die Verwendung von OSGi Services. Dazu wird das bereits existierende Bundle *com.weiglewilczek.example.osgi.contacts.inmemory* Services registrieren und das neue Bundle *com.weiglewilczek.example.osgi.contacts.shell* diese nutzen. Um Gerechtigkeit walten zu lassen, wird diesmal – wie schon beim „Hello World!“-Beispiel im ersten Teil – Eclipse Equinox und PDE eingesetzt, nachdem wir das letzte Mal die Verwendung von Apache Felix und Bnd gezeigt haben. Für dieses Beispiel benötigt man eine aktuelle Version (3.4.1 zum Zeitpunkt der Erstellung dieses Artikels) des Eclipse SDK mit PDE, z. B. das

Artikelserie: OSGi in kleinen Dosen

Teil 1: Erste Schritte mit OSGi

Teil 2: Immer in Bewegung – Bundles und Life Cycle

Teil 3: Immer in Bewegung – Services à la OSGi

Teil 4: Alles XML oder was? – Services auf deklarative Weise

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

Quellcode
auf CD

Abb. 1: OSGi Service Model

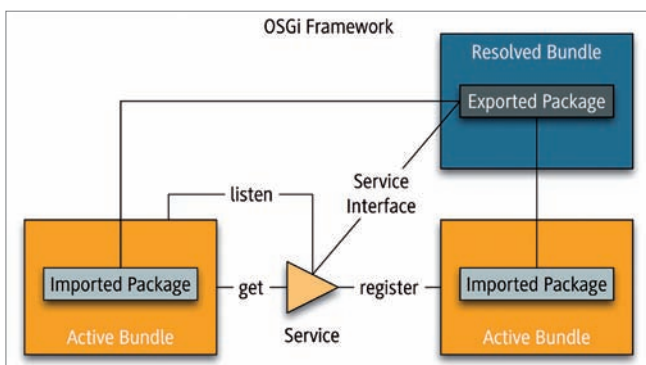
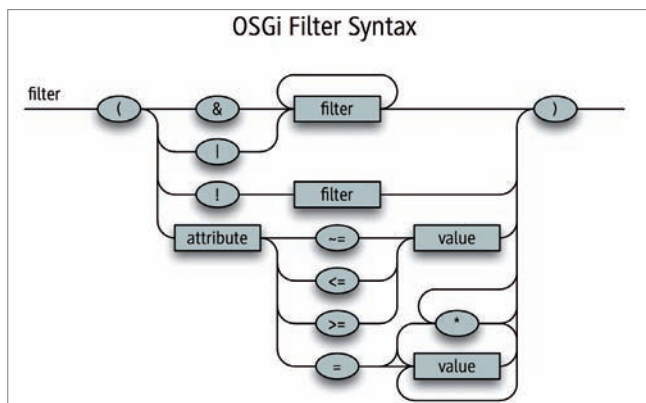


Abb. 2: OSGi Filtersyntax



Anzeige

Package „Eclipse for RCP/Plug-in Developers“ [1]. Den kompletten Sourcecode des Beispiels finden Sie auf der Begleit-CD oder als Download [2].

Services registrieren

Wie bereits erwähnt, werden Services an der Service Registry des OSGi Frameworks registriert. Wie die gesamte Interaktion mit dem OSGi Framework, geschieht auch dies mithilfe des Interfaces *BundleContext*. Die wichtigsten Methoden hierfür lauten:

```
ServiceRegistration registerService(String clazz,
    Object service,
    Dictionary properties);
ServiceRegistration registerService(String[] clazz,
    Object service,
    Dictionary properties);
```

Der erste Parameter ist der voll qualifizierte Name des Service Interfaces bzw. mehrerer Service Interfaces. Dabei ist wichtig, dass der Service, der im zweiten Parameter übergeben wird, das Service Interface bzw. alle Service Interfaces implementiert. Andernfalls werden diese Methoden eine *IllegalArgumentException* werfen. In unserem Beispiel erzeugen wir im Bundle *com.weiglewiczek.example.osgi.contacts.inmemory*

zwei *InMemoryContactRepositories* mit Spieldaten und registrieren diese unter dem Service Interface *ContactRepository* (Listing 1).

Optional können Service Properties in Form von Schlüssel-Wert-Paaren gesetzt werden. Dabei müssen die Schlüssel Strings und die Werte primitive Typen oder solche aus *java.** sein, um implizite Abhängigkeiten zwischen Bundles zu vermeiden. In unserem Beispiel setzen wir die Service Property *contactRepository.name*, deren Wert einen Namen für einen ContentRepository Service repräsentiert.

Das OSGi Framework vergibt analog zu Bundles für jeden Service eine eindeutige numerische ID und setzt diese als Wert der Service Property *service.id*. Die Service Interfaces, unter denen ein Service registriert wird, werden vom OSGi Framework als Wert der Service Property *objectClass* (Tabelle 1) abgelegt.

Damit kommen wir zum Lebenszyklus von Services, der eng mit dem von Bundles verknüpft ist. Nur wenn sich ein Bundle im Zustand *STARTING*, *ACTIVE* oder *STOPPING* befindet, können über dessen *BundleContext* Services registriert werden. Typischerweise

Listing 1

```
// First service
Hashtable<String, Object> properties = new Hashtable<String, Object>();
properties.put(ContactRepository.NAME, "In-memory");
Contact[] contacts = new Contact[] {
    new Contact("John", "Doe"),
    new Contact("Max", "Mustermann")};
context.registerService(ContactRepository.class.getName(),
    new InMemoryContactRepository(contacts), properties);

// Second service
properties = new Hashtable<String, Object>();
properties.put(Constants.SERVICE_RANKING, 1);
properties.put(ContactRepository.NAME, "In-memory-2");
contacts = new Contact[] {
    new Contact("Another", "One")};
context.registerService(ContactRepository.class.getName(),
    new InMemoryContactRepository(contacts), properties);
```

Listing 2

```
ServiceReference[] references =
    context.getServiceReferences(ContactRepository.class,
        getName(), null);

if (references != null) { // Check if any service registered
    for (ServiceReference reference : references) {
        ContactRepository contactRepository = (ContactRepository)
            context.getService(reference);
        if (contactRepository != null) { // Check again!
            System.out.println(MessageFormat.format(
                "All contacts of {0}:",
                reference.getProperty(ContactRepository.NAME)));
            Contact[] contacts = contactRepository.getAllContacts();
            for (Contact contact : contacts) {
                System.out.println(MessageFormat.format("{0} {1}",
                    contact.getFirstName(), contact.getLastName()));
            }
            ...
        }
    }
}
```

Listing 3

```
private final class ContactRepositoryTracker extends ServiceTracker {
    @Override
    public Object addingService(ServiceReference reference) {
        ContactRepository contactRepository = (ContactRepository)
            super.addingService(reference);
        if (contactRepository != null) { // Check again!
            System.out.println(MessageFormat.format("All contacts of {0}:",
                reference.getProperty(ContactRepository.NAME)));
            Contact[] contacts = contactRepository.getAllContacts();
            for (Contact contact : contacts) {
                System.out.println(MessageFormat.format("{0} {1}",
                    contact.getFirstName(), contact.getLastName()));
            }
            ...
        }
    }
}
```

geschieht das beim Starten, d. h. in der Methode *BundleActivator.start()*. Mithilfe der beim Registrieren zurückgegebenen *ServiceRegistration* können Services wieder deregistriert werden. Allerdings ist das „manuelle“ Deregistrieren oft gar nicht nötig, weil das OSGi Framework dies automatisch vornimmt, wenn ein Bundle gestoppt wird.

Services nutzen

Zur Nutzung von Services sind zwei Schritte erforderlich. Zunächst wird unter Angabe eines Service Interfaces eine oder mehrere *ServiceReferences* erfragt, danach werden mithilfe dieser der oder die tatsächlichen Services abgerufen. Die hierfür wichtigsten Methoden des *BundleContext* lauten:

```
ServiceReference getServiceReference(String clazz);
ServiceReference[] getServiceReferences
    (String clazz, String filter);
Object getService(ServiceReference reference);
```

Der Grund für die gerade aufgeführte Mehrdeutigkeit ist einfach zu erklären: Es können unter demselben Service Interface beliebig viele Services registriert werden, wohlgermerkt auch gar keiner. Der Nutzer kann daher a priori nicht wissen, wie viele Services registriert sind, und muss mit dieser inhärenten Mehrdeutigkeit umgehen. Die Methode *getServiceReferences()* ermöglicht durch die Verwendung eines Filters, die Ergebnismenge einzuschränken. Sie liefert für alle passenden Services eine *ServiceReference* zurück. Anders die Methode *getServiceReference()*. Hier wendet bereits das OSGi Framework eine Heuristik an, die die Ergebnismenge auf einen Service einschränkt, sofern überhaupt passende registriert sind. Dabei wird der Service zurückgeliefert,

dessen Property *service.ranking* (Tabelle 1) den höchsten Wert hat. Falls dies zu keiner eindeutigen Entscheidung führt, wird der Service mit der kleinsten ID verwendet.

In unserem Beispiel erstellen wir ein neues Bundle *com.weiglewilczek.example.osgi.contacts.shell*, das beim Starten alle registrierten *ContactRepository* Services aufruft, sodass wir *getServiceReferences()* verwenden. Beide Methoden bedürfen der Prüfung auf null, denn dies ist der Rückgabewert, auch für *getServiceReferences()*, falls kein Service zur Anfrage passt. Anschließend kann *getService()* aufgerufen werden, wobei eine zuvor zurückgelieferte *ServiceReference* als Parameter übergeben wird. Aufgrund der Dynamik von OSGi muss unbedingt nochmals auf null geprüft werden (Listing 2), denn es könnte ja vorkommen, dass im Moment zwischen der Abfrage der *ServiceReference* und des Service dieser deregistriert wurde.

In unserem Beispiel geben wir zum einen den Wert der Service Property *contactRepository.name* aus, also den Namen des *ContactRepository*. Anschließend geben wir die Namen aller enthaltenen Contacts aus. Um das Beispiel auszuführen, legen wir eine OSGi Framework Run Configuration an, nehmen unsere Bundles sowie deren Abhängigkeiten auf und starten zuerst *com.weiglewilczek.example.osgi.contacts.inmemory* und danach *com.weiglewilczek.example.osgi.contacts.shell*. Diese Startreihenfolge ist wichtig, da sowohl das Registrieren als auch das Konsumieren der Services im Beispiel beim Starten erfolgt. Wenn die Reihenfolge umgedreht wird, werden wir keinerlei Ausgabe sehen. Dieses Verhalten ist höchst problematisch, denn bei einem dynamischen modularen System kann

Service Property	Bedeutung
objectClass: String[]	Service Interfaces der Registrierung, vom OSGi Framework gesetzt
service.id: Long	Eindeutige Service ID, vom OSGi Framework gesetzt
service.ranking: Integer	Wenn mehrere Services zu einer Anfrage über <i>getServiceReference()</i> passen, dann wird derjenige mit dem höchsten Wert zurückgeliefert, der Default entspricht 0

Tabelle 1: Wichtige Standard-Service-Properties

die Startreihenfolge kaum kontrolliert werden.

Service Properties und Filter

Wir haben bereits eine Möglichkeit kennengelernt, wie man Service Properties nutzen kann, und zwar als Informationsträger. Eine weitere Möglichkeit von besonderer Bedeutung ist die Verwendung in Filtern, um die Ergebnismenge beim Abrufen von *ServiceReferences* einzuschränken. Wie bereits beschrieben, verwendet das OSGi Framework die Service Property *service.ranking* dazu, beim Aufruf von *getServiceReference()* einen eindeutigen Treffer zu ermitteln. Natürlich können wir auch beliebige eigene Service Properties definieren und diese in Filtern nutzen.

Wie sieht nun ein Filter aus? OSGi verwendet dazu ein besonderes Format: Die „String Representation of LDAP Search Filters“ [3]. Die Syntax beruht auf der polnischen Notation, bei der zuerst die Operatoren und danach die Operanden geschrieben werden. Abbildung 2 visualisiert die Filtersyntax in einem Syntaxdiagramm. Im Folgenden zwei Beispiele:

```
(objectClass=com.weiglewilczek*)
(&(objectClass=com.weiglewilczek*)(service.
    ranking>=10))
```

Unser Beispiel ist zu einfach, um Filter im Code zu verwenden. Aber die Equinox Console ermöglicht bei der Verwendung des Kommandos die Angabe eines Filterausdrucks. Wenn wir das Beispiel starten und den ersten oben aufgeführten Beispielfilter eingeben, werden genau die beiden Services ausgegeben (Abb. 3).

Services und Dynamik

OSGi ist ein dynamisches System und dies gilt insbesondere für Services. Konsumenten müssen mit dieser inhärenten Dynamik umgehen. Dafür bietet das OSGi Framework die Möglichkeit, auf *ServiceEvents* zu reagieren, also insbesondere auf das Registrieren und Deregistrieren. Über den *BundleContext* können *ServiceListeners* angemeldet werden, die entweder alle *ServiceEvents*

```
osgi> services (objectClass=com.weiglewilczek*)
{com.weiglewilczek.example.osgi.contacts.core.ContactRepository}={
  Registered by bundle: initial@reference:file:../../com.weiglewil
  Bundles using service:
    initial@reference:file:../../com.weiglewilczek.example.osgi.co
{com.weiglewilczek.example.osgi.contacts.core.ContactRepository}={
  Registered by bundle: initial@reference:file:../../com.weiglewil
  Bundles using service:
    initial@reference:file:../../com.weiglewilczek.example.osgi.co
```

Abb. 3: Anwendung von Filtern in der Equinox Console

oder eine über Filter eingeschränkte Untermenge erhalten.

```
void addServiceListener(ServiceListener listener);
void addServiceListener(ServiceListener listener,
    String filter);
```

Aufgrund von Nebenläufigkeit kann der Umgang mit diesen *ServiceListeners* recht diffizil sein. Soll beispielsweise eine jederzeit aktuelle Liste von Services eines bestimmten Typs vorgehalten werden, so besteht die Gefahr, Duplikate zu erzeugen oder einzelne Services auszulassen. Daher spezifiziert das OSGi Service Compendium mit dem *ServiceTracker* eine Hilfsklasse, die mögliche Race Conditions und andere Schwierigkeiten berücksichtigt und damit die Beherrschung der Servicedynamik vereinfacht. Wir empfehlen, in der Regel nicht direkt mit *ServiceListeners* zu arbeiten, sondern den *ServiceTracker* zu verwenden. Dieser bietet nicht nur die gerade beschriebene Möglichkeit der Serviceliste, sondern u. a. auch Methoden, um auf Registrieren und Deregistrieren zu reagieren oder eine bestimmte Zeit auf einen Service zu warten.

```
public Object[] getServices()
public Object addingService(ServiceReference
```



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] Eclipse SDK: www.eclipse.org/downloads
- [2] WeigleWilczek-Examples: www.weiglewilczek.com/examples/
- [3] String Representation of LDAP Search Filters: www.ietf.org/rfc/rfc1960.txt