

Erste Schritte mit 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

OSGi prägt schon heute vielerorts die Java-Welt und schickt sich an, zu einer universellen Basistechnologie zu werden. So mancher Leser wird diesem bereits seit 2000 existierenden Standard vermutlich erstmalig in Form des Eclipse SDK oder der Eclipse Rich Client Platform begegnet sein, da die Eclipse-Plug-in-Architektur seit Version 3.0 auf OSGi basiert. Damit hat Eclipse zweifelsohne einen bedeutenden Anteil an der aktuellen Popularität dieser Technologie, die jedoch viel mehr zu bieten hat als ein „Modulsystem für Rich Clients“.

Ein Blick zurück

Ursprünglich wurde OSGi nicht mit Blick auf Rich Clients oder gar Server, sondern für so genannte Residential Internet Gateways – Systeme aus der Gebäudetechnik – konzipiert. Derartige Systeme erfordern üblicherweise Fernmanagement sowie Installation von Komponenten im laufenden Betrieb. Mit diesen Anforderungen im Fokus entstand die erste Version der OSGi-Spezifikation, deren Urheber

zahlreiche namhafte Unternehmen sind, die sich 1999 zu diesem Zweck in einer internationalen Organisation zusammenfanden, der OSGi Alliance [1]. OSGi stand ursprünglich als Abkürzung für „Open Service Gateway Initiative“, heute steht der Begriff für sich bzw. als Teil von OSGi Service Platform, Titel der aktuellen Spezifikation (Version 4.1).

Ein dynamisches und serviceorientiertes Modulsystem

Warum ist OSGi eigentlich so erfolgreich? Welchen Eigenschaften ist es zu verdanken, dass OSGi aus der Java-Welt kaum mehr wegzudenken ist? Die OSGi Alliance definiert OSGi als dynamisches Modulsystem für Java. Damit werden zwei der drei aus unserer Sicht entscheidenden Merkmale aufgeführt: Modularisierung und Dynamik.

Modularisierung ist ein „altes Hausmittel“ – nicht nur in der Informatik –, um komplexe Probleme zu lösen. Schließlich lassen sich überschaubare Häppchen leichter verdauen als das große Ganze. Die Objektorientierung im Allgemeinen und Java im Speziellen bieten bereits Möglichkeiten zur Modularisierung, z.B. in Form von Klassen, Paketen, Sichtbarkeitsregeln etc. Allerdings hat sich in der Praxis herausgestellt, dass Java ein Modulkonzept „oberhalb“ von Packages vermissen lässt. OSGi schließt diese Lücke, sodass anstelle der sonst oft anzutreffenden monolithischen Systeme solche treten, die aus verschiedenen Modulen mit wohldefinierten öffentlichen Schnittstellen und Abhängigkeiten

bestehen. Wer meint, Modularisierung sei auch organisatorisch zu erzielen, z.B. durch mehrere Projekte oder durch den Einsatz von komponentenorientierten Build-Systemen wie Maven, der betrachtet nur die halbe Wahrheit. OSGi hingegen adressiert nicht nur den Zeitraum der Entwicklung, sondern insbesondere auch das Laufzeitverhalten: Es wird spezifiziert, wie Module im laufenden Betrieb unter Berücksichtigung ihrer Abhängigkeiten installiert, aktualisiert und entfernt werden können. Gerade diese Dynamik macht OSGi für serverseitige bzw. hochverfügbare Systeme interessant, da Hot Deployment im Standard inklusive ist.

Die dritte wesentliche Eigenschaft von OSGi ist ein serviceorientiertes Programmiermodell: Module können Objekte als Services bereitstellen, indem sie diese an der OSGi Service Registry registrieren. Andere Module können diese Services abrufen und so untereinander kommunizieren. Dies führt zu einer losen Kopplung der beteiligten Module.

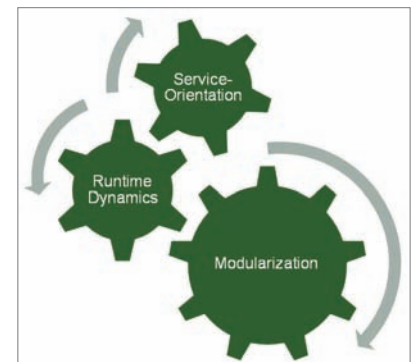


Abb. 1: Herausragende OSGi-Eigenschaften

Artikelserie:

OSGi in kleinen Dosen

Teil 1: Erste Schritte mit OSGi

Teil 2: Immer in Bewegung – Bundles und Lifecycle

Teil 3: Was wünschen Sie? – Services à la OSGi

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

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

Jede dieser drei Eigenschaften, Modularisierung, Laufzeitdynamik und Serviceorientierung, ist an sich nichts Neues, aber OSGi verbindet diese synergetisch, sodass sich die folgenden Nutzenpotenziale ergeben (Abb. 1):

- Erhöhte Flexibilität durch rigorose Trennung von API und Implementierung
- Einsparung von Entwicklungskosten durch Wiederverwendung von Modulen
- Einsparung von Betriebskosten durch standardisiertes Lifecycle Management, z.B. Hot Deployment oder parallelen Betrieb mehrerer Modulversionen
- Hohe Qualitätseffizienz durch gute Testbarkeit aufgrund von loser Kopplung
- Nutzung neuer Möglichkeiten, z.B. in der Softwareverteilung

Doch wie unterstützt OSGi die beschriebenen Eigenschaften? Dazu werfen wir zunächst einen Blick auf die Gesamtarchitektur. Die vorzügliche OSGi-Spezifikation, die in Form von Core Specification und Service Compendium vorliegt,

definiert im Wesentlichen das OSGi Framework sowie die OSGi Standard Services. Das OSGi Framework bzw. eine Implementierung dessen stellt die Laufzeitumgebung für Software auf Basis von OSGi dar. Die Standardservices, z.B. Log Service oder Http Service, standardisieren die Lösung typischer Problemstellungen und können in eigenen Systemen genutzt werden.

Das OSGi Framework

Um gleich im OSGi-Jargon sprechen zu können, führen wir zunächst einen zentralen Begriff ein: Ein OSGi-Modul wird Bundle genannt. Somit bestehen Systeme auf Basis von OSGi aus Bundles, für die das OSGi Framework die Laufzeitumgebung darstellt (Abb. 2). Die Funktionalität des OSGi Frameworks ist auf mehrere Schichten verteilt, die in Abbildung 3 orange dargestellt sind.

Der Module Layer definiert die strukturellen Aspekte rund um Bundles, d.h. die statische Sicht auf das OSGi-Modulkonzept. Bundles sind JAR-Archive und enthalten neben Klassen und Ressourcen mit dem Bundle Manifest eine Datei, die die OSGi-Eigenschaften des Bundles beschreibt. Besonders hervor-

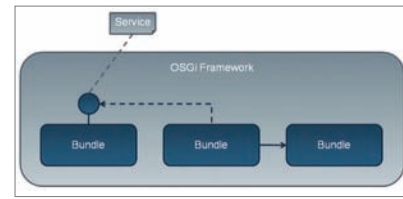


Abb. 2: OSGi-System

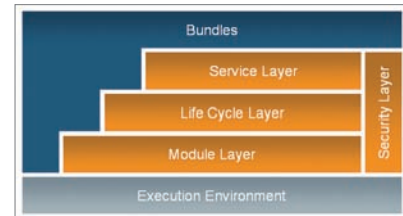


Abb. 3: Schichtenarchitektur des OSGi Frameworks

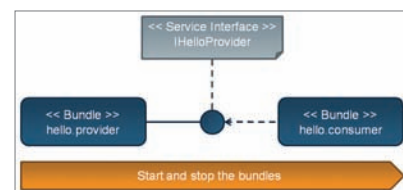


Abb. 4: „Hello World“-Architektur

zuheben ist, dass im Bundle Manifest die Abhängigkeiten und das API, d.h. die öffentliche Schnittstelle, explizit deklariert werden. Das OSGi Framework stellt sicher, dass andere Bundles nur auf

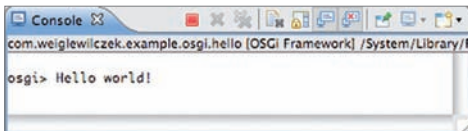


Abb. 5: „Hello World!“-Ergebnis

das API eines Bundles zugreifen können, und das auch nur dann, wenn sie selbst entsprechende Abhängigkeiten deklarieren.

Der Lifecycle Layer bringt die dynamische Sicht ins Spiel, indem er den Lebenszyklus von Bundles im OSGi Framework definiert. So können Bundles beispielsweise zur Laufzeit installiert und wieder deinstalliert werden. Weiterhin können Bundles auch gestartet und gestoppt werden, woraufhin in einem speziellen Objekt, dem *BundleActivator*, entsprechende *start()*- und *stop()*-Methoden aufgerufen werden.

Der Service Layer fügt ein service-orientiertes Programmiermodell hinzu,

sodass Bundles in bestimmten Phasen ihres Lebenszyklus Objekte als Services an der OSGi Service Registry anmelden oder abrufen können. Dabei wird das Serviceinterface, das vom Service implementiert wird, als Schlüssel verwendet. Besonderes Augenmerk muss hierbei der Laufzeitdynamik gewidmet werden: Schließlich können die Bundles, die Services anbieten, jederzeit „kommen und gehen“, womit Servicekonsumenten umgehen können müssen. Daher bietet OSGi Mechanismen, um den Service Lifecycle zu verfolgen, z.B. den im „Hello World!“-Beispiel verwendeten Service-Tracker.

Daneben stellt der Security Layer auf Basis von Java 2 Security spezifische Sicherheitsbelange zur Verfügung. Neben dem Signieren von Bundles spielen darin für OSGi spezifische Permissions eine zentrale Rolle. So kann z.B. festgelegt werden, ob ein Bundle berechtigt wird, bestimmte Services zu verwenden oder das API anderer Bundles zu nutzen.

Unterhalb der Framework-Schichten definiert OSGi in Form von Execution Environments Abstraktionen von Java-Laufzeitumgebungen. Diese können pro Bundle zur Voraussetzung gemacht werden, wobei das OSGi Framework selbst das OSGi/Minimum-1.1 Execution Environment benötigt. Es gibt zahlreiche OSGi-Implementierungen, sowohl kommerzielle Produkte als auch kostenlose Open-Source-Lösungen, z.B. Apache Felix [2] oder Knopflerfish OSGi [3]. Wir werden für das folgende Beispiel Eclipse Equinox [4] verwenden, weil das Eclipse SDK als Entwicklungsumgebung mit dem Plug-in Development Environment (PDE) eine hervorragende Werkzeugunterstützung anbietet, die optimal auf Equinox abgestimmt ist, sodass die ersten Schritte besonders einfach sind. Trotzdem werden wir uns ausschließlich am aktuellen OSGi-Standard orientieren, sodass die Beispiele problemlos auf anderen Implementierungen laufen können.

„Hello World!“

Das Beispiel (Abb. 4) besteht aus zwei Bundles: Das eine bietet beim Starten unter dem Serviceinterface *IHelloProvider* einen Service an, das andere

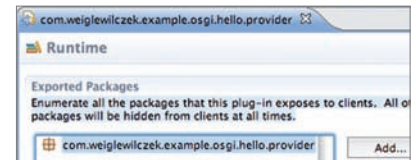


Abb. 6: Deklaration der öffentlichen Schnittstelle

„lauscht“, sobald es gestartet wurde, auf solche Services, um diese aufzurufen und das Ergebnis „Hello World!“ auf der Konsole auszugeben (Abb. 5).

Für dieses Beispiel wird Java 5 und Eclipse SDK Ganymede mit PDE benötigt, z.B. das Package Eclipse for RCP/Plug-in Developers [5]. Um Nebeneffekte zu vermeiden, sollte ein neuer Workspace verwendet werden.

Der Provider

Zunächst erstellen wir das Bundle *com.weiglewilczek.example.osgi.hello.provider*. Wir werden in der kommenden Folge auf übliche Namenskonventionen eingehen. PDE unterstützt die Entwicklung von OSGi Bundles unter anderem durch Plug-in-Projekte. Diese sind, vereinfacht dargestellt, Java-Projekte, deren Build-Klassenpfad den deklarierten Abhängigkeiten der Bundles entspricht. Auf diese Weise können Laufzeitfehler aufgrund fehlender oder fehlerhaft deklarierter Abhängigkeiten schon zur Entwicklungszeit vermieden werden.

Plug-in-Projekte werden mit einem eigenen New-Wizard angelegt. In unserem Beispiel wird auf der ersten Wizard-Seite *com.weiglewilczek.example.osgi.hello.provider* als Projektname eingetragen und an *OSGi framework* mit Ausprägung *standard* selektiert. Auf der zweiten Wizard-Seite können die Vorbelegungen mit einer Ausnahme übernommen werden: Dem vorgeschlagenen Namen für *Activator* wird ein *internal*-Package-Segment vor dem Klassennamen eingeschoben. Nach Fertigstellung wurde ein neues Projekt angelegt, das im Wesentlichen die *Activator*-Klasse sowie das Bundle Manifest (mehr dazu in der kommenden Folge) enthält.

Als Nächstes erstellen wir im Package *com.weiglewilczek.example.osgi.hello.provider* (diesmal ohne *internal*) das Serviceinterface *IHelloProvider* mit der Methode *sayHello()*:

Listing 1

```
public class Activator implements BundleActivator {

    private ServiceTracker helloProviderTracker;

    public void start(final BundleContext context) {
        helloProviderTracker = new ServiceTracker(context,
            IHelloProvider.class.getName(),
            new ServiceTrackerCustomizer() {

                public Object addingService(final ServiceReference reference) {
                    final IHelloProvider helloProvider = (IHelloProvider)
                        context.getService(reference);
                    System.out.println(helloProvider.sayHello());
                    return helloProvider;
                }

                public void modifiedService(final ServiceReference reference,
                    final Object service) { // Nothing to be done!
                }

                public void removedService(final ServiceReference reference,
                    final Object service) { // Nothing to be done!
                }
            });
        helloProviderTracker.open();
    }

    public void stop(final BundleContext context) {
        if (helloProviderTracker != null) helloProviderTracker.close();
    }
}
```

```
public interface IHelloProvider {
    String sayHello();
}
```

Dazu implementieren wir im Package `com.weiglewilczek.example.osgi.hello.internal.provider` (diesmal wieder mit *internal*) die Klasse `OsgiHelloProvider`:

```
public class OsgiHelloProvider implements
    IHelloProvider {

    public String sayHello() {
        return "Hello world!";
    }
}
```

In der Methode `Activator.start()` erzeugen wir eine Instanz des `OsgiHelloProviders` und registrieren diese als OSGi Service unter dem Serviceinterface `IHelloProvider`:

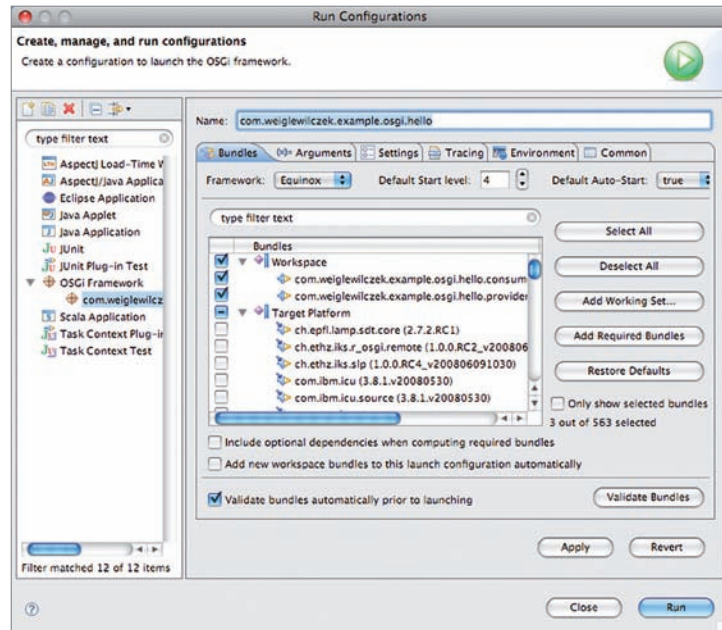
```
public void start(final BundleContext context) {
    context.registerService(
        IHelloProvider.class.getName(),
        new OsgiHelloProvider(), null);
}
```

Abschließend deklarieren wir noch das Package, das das Serviceinterface als Bestandteil der öffentlichen Schnittstelle enthält. Dies erfolgt im Bundle Manifest, das wir durch Doppelklick im Package Explorer mit dem Manifest-Editor – einem weiteren PDE-Werkzeug – öffnen. Im Reiter `Runtime` fügen wir das Package `com.weiglewilczek.example.osgi.hello.provider` zu den Exported Packages hinzu (Abb. 6).

Der Consumer

Nun erstellen wir das Bundle `com.weiglewilczek.example.osgi.hello.consumer`. Dazu gehen wir analog zum Provider vor. Der Consumer enthält mit dem `Activator` nur eine Klasse. Listing 1 zeigt, wie wir in dessen Methode `start()` einen `ServiceTracker` (mehr dazu in kommenden Folgen) für obiges Serviceinterface anlegen. Der `ServiceTrackerCustomizer` ruft in der Methode `addingService()` den vom Provider registrierten `IHelloService` auf und gibt das Ergebnis auf der Konsole aus. Um den ServiceTracker und das vom Provider angebotene Serviceinterface nutzen

Abb. 7:
OSGi
Framework
Run
Configu-
ration



zu können, müssen diese Abhängigkeiten im Bundle Manifest eingetragen werden. Dazu verwenden wir wiederum den Manifest-Editor und fügen im Reiter `Dependencies` die entsprechenden Packages hinzu.

„Hello World!“ in Aktion

Um dieses Beispiel ablaufen zu lassen, können wir eine weitere hilfreiche Funktion des PDE nutzen: Es ist nicht nötig, Bundles im korrekten Zielformat zu erstellen und in einem OSGi Framework zu installieren. Vielmehr können Plug-in-Projekte direkt aus der Entwicklungsumgebung heraus gestartet werden. Dazu wird eine Run Configuration vom Typ `OSGi Framework` benötigt. Über das Menü `RUN | RUN CONFIGURATIONS ...` gelangen wir in einen Dialog, in dem wir eine neue `OSGi Framework` Run Configuration anlegen (Abb. 7). Dabei ist der Reiter `Bundles` besonders wichtig, denn hier legen wir fest, welche Bundles verwendet werden sollen. Die Voreinstellungen bewirken, dass die verwendeten Bundles nicht nur installiert, sondern auch gleich gestartet werden, was wir für unser Beispiel auch benötigen. Wir wählen unsere beiden Beispiel-Bundles und lassen mittels `Add Required Bundles` alle abhängigen Bundles automatisch ergänzen. Ein Klick auf `RUN` lässt uns „Hello World!“ in Aktion erleben (Abb. 5).

Ausblick

In dieser ersten Folge haben wir einen Überblick über OSGi gegeben, die Architektur des OSGi Frameworks betrachtet und Modularisierung, Laufzeitdynamik und Serviceorientierung als wesentliche Eigenschaften von OSGi herausgestellt. Die nächste Folge steht unter dem Motto „Immer in Bewegung – Bundles und Lifecycle“ und beleuchtet im Detail das Modulkonzept und die Laufzeitdynamik von OSGi. ■



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] www.osgi.org/
- [2] felix.apache.org/
- [3] www.knopflerfish.org/
- [4] www.eclipse.org/equinox
- [5] www.eclipse.org/downloads