Serverside OSGi: JEE und OSGi integrieren

OSGi drängt immer weiter in Richtung Enterprise-Anwendungen. So bildet OSGi mittlerweile das Rückgrat der neuesten Generation der Java-EE-Applikationsserver IBM WebSphere und BEA Weblogic. Und auch JBoss hat angekündigt, seinen Microcontainer auf OSGi umzustellen. Damit ist der Applikationsserver an sich zwar modular aufgebaut und kann einfach erweitert werden, aber wie profitiert man von der Modularität und Flexibilität von OSGi in eigenen Java-EE-Anwendungen?

von Dirk Mascher und Björn Wand

Das Java Magazin hat bereits in mehreren Artikeln über OSGi berichtet [1],[2],[3]. Bis jetzt wurde OSGi aber noch nicht im Kontext von Java-Enterprise-(JEE-)Anwendungen vorgestellt. Die Firma Loyalty Partner GmbH mit Sitz in München [4] hat OSGi erfolgreich in ihr Produkt Loyalty Management Solution (LMS), eine JEE-Enterprise-Anwendung, integriert. Der vorliegende Artikel beschreibt einige der Herausforderungen, die sich bei der Integration von OSGi und Java EE ergeben haben, und skizziert Lösungen, die im Laufe des Projekts erarbeitet wurden. Insbesondere wird der Themenkomplex Classloading beschrieben.

Die in diesem Artikel skizzierten Lösungsansätze erheben nicht den Anspruch von Best-Practice-Lösungen für die Einbettung von OSGi [5] in JEE-Anwendungen. Alle in dem genannten Projekt gestellten funktionalen und auch nichtfunktionalen Anforderungen (Stichwort „Performance“) konnten mit der entwickelten Vorgehensweise jedoch erfüllt bzw. deutlich übertroffen werden. Ein großer Vorteil der entwickelten Architektur liegt u.a. darin, dass sukzessive einzelne Teile der Businesslogik einer Anwendung in OSGi Bundles ausgelagert werden können, sodass sich auch für bereits entwickelte Enterprise-Anwendungen ein einfacher Migrationspfad erschließt.

Ohne Anforderungen kein Projekt

Die folgende Auflistung gibt einige der Anforderungen wieder, die die Entscheidung, OSGi einzusetzen, unmittelbar beeinflusst haben.

Quellcode auf CD

- Das Produkt LMS ermöglicht es, beliebige Bonusprogramme mit unterschiedlichen fachlichen Anforderungen zu betreiben. Hierfür ist ein Framework notwendig, das einen flexiblen Mechanismus bereitstellt, der es ermöglicht, konkrete Businessregeln hinter wohldefinierten Schnittstellen „einzuhängen".
- Der Prozess des „Einhängens" soll dynamisch zur Laufzeit möglich sein („Hot Deployment"). Weder der Applikationsserver noch die eigentliche JEE-Enterprise-Applikation soll während des Einhängens durchgestartet werden müssen („Zero Downtime").
- Derart einhängbare Businessregeln bzw. Java-Klassen sollen neben einer eindeutigen Versionsnummer mit einem Gültigkeitszeitraum versehen werden können. Abhängig von einem Zeitstempel, der durch Workflows in der Businesslogik bestimmt wird, soll das Framework die jeweils zu diesem Zeitpunkt gültige Version einer Klasse anziehen und ausführen können.
- Aus Anforderung Nr. 3 leitet sich unmittelbar die technische Anforderung ab, dass von einer Klasse (definiert durch ihren voll qualifizierten Java-Klassennamen) zur Laufzeit mehrere Versionen lad- und instanziierbar sein müssen.

Warum OSGi?

Java-Entwickler, die sich schon einmal intensiver mit dem Thema Classloading auseinandergesetzt haben, wissen, dass sich Anforderung Nr. 4 in Java mithilfe eigener Classloader umsetzen lässt. Eine Klasse wird in Java nicht nur durch ihren voll qualifizierten Namen, sondern zusätzlich auch durch ihren Classloader identifiziert. Mit mehreren Classloadern ist es also möglich, zum selben Zeitpunkt mehrere Class-Objekte von Klassen mit demselben voll qualifizierten Klassennamen isoliert voneinander im Speicher zu halten. Allerdings ist das Schreiben von eigenen Classloadern keine triviale Angelegenheit und birgt durchaus ihre Tücken. Aus diesem Grund sollte man sich, bevor man in die Versuchung kommt einen eigenen Classloader zu schreiben, besser umsehen, ob das, was man vorhat, nicht schon von anderen gemacht wurde. Folgt man diesem Grundsatz und sucht eine existierende Lösung für die Umsetzung von Anforderung Nr. 4, wird man mit großer Wahrscheinlichkeit auf OSGi stoßen.

Klassen werden in OSGi immer in Form von Bundles paketiert. Bundles sind dabei im Prinzip nichts anderes als JAR-Dateien mit einem erweiterten Manifest. Das OSGi Framework erzeugt pro Bundle einen separaten Classloader. Interessant ist dabei, dass die einzelnen Classloader keine Hierarchie bilden, sondern erst einmal nichts voneinander wissen. Allerdings kann der Entwickler eines Bundles durchaus Beziehungen zwischen Bundles herstellen. So ist es beispielsweise möglich, dass ein Bundle bestimmte Packages exportiert. Durch das Exportieren eines Packages werden alle Klassen aus diesem Package für andere Bundles sichtbar, allerdings nur für solche Bundles, die explizit ein Interesse an einem exportierten Package über eine korrespondierende Importdirektive ausdrücken. Definiert werden diese Export-/Importbeziehungen über entsprechende *Export-Package* bzw. *Import-Package* Header in den Manifest-Dateien der beteiligten OSGi Bundles. Zur Laufzeit ergibt sich durch diese Beziehungen effektiv ein Netzwerk bzw. ein Geflecht von Classloadern.

Bundles sind gemäß Spezifikation immer versioniert. Die Versionsnummer ist ein Pflichtfeld in dem Manifest von OSGi Bundles. Mithilfe der Versionsnummer ist es möglich, Importbeziehungen auf Package-Ebene zu konkretisieren. Ein „Client Bundle" kann also beispielsweise ausdrücken, dass es Klassen aus dem Package *xyz* benötigt, aber nur in der Version 1.0 bis einschließlich 2.0. Ein weiteres Feature, das man von OSGi quasi geschenkt bekommt, ist die Fähigkeit, neuen Code in ein laufendes System einzubringen, ohne das System dafür stoppen und neu starten zu müssen. OSGi definiert für Bundles einen Lifecycle. Beim Starten und Stoppen eines Bundles kann eigener Code in Form einer speziellen Aktivator-Klasse ausgeführt werden. Die Aktivator-Klasse muss hierfür das Interface *org.osgi.framework.BundleActivator* implementieren und im Manifest des Bundles unter dem Header *Bundle Activator* eingetragen sein. Mithilfe von Aktivatoren ist es beispielsweise möglich, OSGi-Services zu registrieren und über den OSGi-Container verfügbar zu machen. OSGi-Services sind dabei nichts anderes als Instanzen einfacher POJO-Klassen, die in einer globalen Registry unter dem Namen eines bestimmten Service-Interfaces registriert werden. Bei der Registrierung ist es möglich, zusätzliche Metadaten zu hinterlegen, die wiederum von Servicekonsumenten verwendet werden können, um die Suche nach bestimmten Services einzugrenzen.

Alle hier beschriebenen grundlegenden Konzepte von OSGi haben in dem genannten Projekt dazu geführt, auf OSGi als Basistechnologie zu setzen. Bei der Suche nach einer geeigneten OSGi-Implementierung fiel die Wahl auf Equinox [6]. Dafür sprachen mehrere Gründe: Da Equinox das Herzstück der freien Entwicklungsumgebung Eclipse darstellt, wird Equinox von einer großen Entwicklergemeinde produktiv eingesetzt. Dass es sich bei Equinox um ein stabiles Produkt handelt, lag also auf der Hand. Außerdem bietet gerade Eclipse mit seinen PDE Tools eine hervorragende Umgebung für die Entwicklung eigener OSGi Bundles. Es stellte sich also „nur" noch die Frage, wie man Equinox in einem JEE-Applikationsserver am besten einbetten kann.

OSGi und JEE

Jeff McAffer, einer der ursprünglichen Architekten von Eclipse und mittlerweile Leiter der Eclipse-Runtime-Entwicklung, hat zwei Vorgehensweisen beschrieben, wie man Equinox und einen JEE-Container integrieren kann [7]. Allerdings beziehen sich die dort beschriebenen Ansätze nur auf die Integration mit einem Servlet-Container und nicht auf die Integration mit einem EJB-Container. In dem ersten von McAffer beschrieben Ansatz wird Equinox über eine kleine Webanwendung in einen Servlet-Container eingebettet. Die Webanwendung besteht im Wesentlichen nur aus einem Bridge Servlet, das Equinox startet und dann alle Anfragen an Servlets und JSPs, d.h. die eigentliche Webanwendung, weiterleitet. Alle Servlets und

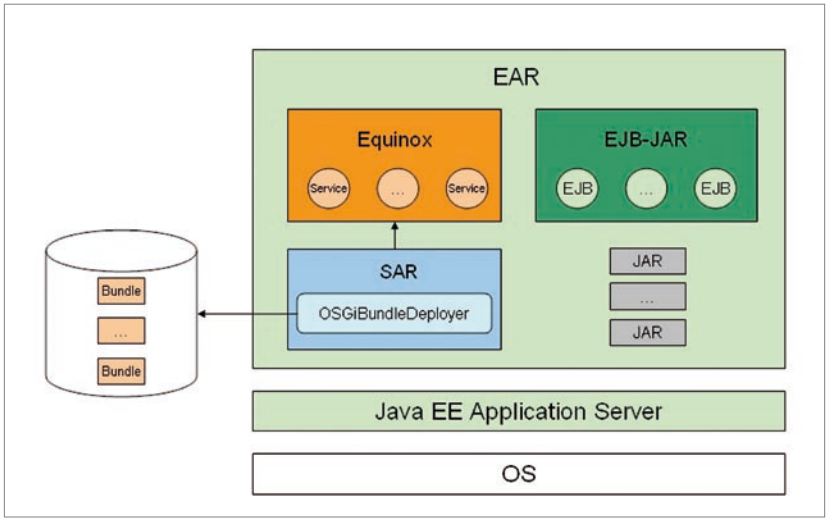

Abb. 1: Integration von Equinox und JBoss

JSPs werden hier nicht in einem Standard-WAR-Archiv, sondern in Form eines OSGi Bundles deployt. Der zweite von McAffer beschriebene Ansatz dreht die Beziehung der Container genau um. Hier ist Equinox der Top-Level-Container, in dem der Servlet-Container in Form eines OSGi Bundles gestartet wird. Auch bei diesem Ansatz wird die eigentliche Webanwendung in Form eines Bundles deployed, sodass sich beide Ansätze aus Sicht des Entwicklers einer Webanwendung letztendlich gleich darstellen. Für das genannte LMS-Projekt erwiesen sich beide Ansätze allerdings als ungeeignet, da in beiden Fällen davon ausgegangen wird, dass die komplette Enterprise-Applikation in Form von OSGi Bundles deployed wird. Genau dies wollte man in dem Projekt allerdings nicht. Stattdessen wurde eine Architektur entwickelt, die nur an bestimmten Stellen den Übergang aus der JEE-Welt in die OSGi-Welt vorsieht. EJBs werden also in Form von EJB-JARs, die in einem EAR-Archiv enthalten sind, deployed und von dem EJB-Container verwaltet. Neben den eigentlichen JEE-Modulen enthält das EAR-Archiv zusätzlich Equniox als OSGi Framework und alle benötigten Third Party Libraries in Form klassischer JAR-Archive. Beim Starten der Enterprise-Anwendung wird, bevor die einzelnen JEE-Module gestartet werden, zuerst Equinox gestartet. Im Fall von JBoss ist dies über einen JBoss-spezifischen MBean-Service möglich (Abb. 1). Im Fall von Bea Weblogic kann hierfür ein Application Lifecycle Listener verwendet werden. Da Bundles austauschbare (hot-deploybare) Logik- bzw. Businessregeln enthalten (Anforderung 2), werden sie nicht statisch in das EAR-Archiv gepackt, sondern unabhängig von dem EAR über eine Datenbank bereitgestellt. Verantwortlich für das Starten von Equinox und für das Installieren und Starten aller Bundles ist die Klasse *OSGiBundleDeployer*.

EJBs, die Businessregeln aus OSGi Bundles verwenden, sind gegen POJO-Interfaces entwickelt. Diese Interfaces, im Projekt *Hooks* genannt, werden von OSGi-Services implementiert, die über Aktivatoren der hot-deploybaren OSGi Bundles registriert und über den OSGi-Container bereitgestellt werden. Der Zugriff auf einen solchen OSGi-Service erfolgt dabei über einen dynamischen Proxy, der alle Interna der OSGi API kapselt und nach außen das Hook-Interface „implementiert". Erzeugt werden diese Proxies, wie in folgendem Codebeispiel, über eine Factory.

```
SampleHook hook = HookProxyFactory.
createProxy(SampleHook.class);
```

Bei jeder Invokation einer Methode auf das *hook*-Objekt findet nun ein Übergang aus dem EJB-Container in den OSGi-Container statt. Handelt es sich bei der Implementierung eines Hooks um eine Businessregel, die über ein Bundle mit einem definierten Gültigkeitszeitraum bereitgestellt wird (Anforderung Nr. 3), muss als weiterer Parameter ein Zeitstempel übergeben werden, der intern von dem dynamischen Proxy verwendet wird, um die für diesen Zeitpunkt gültige Version der Hook-Implementierung zu finden. Beispielhaft sieht ein entsprechender Aufruf der Factory in diesem Fall wie folgt aus:

```
Date timestamp = ... // obtained from somewhere else
SampleHook hook = HookProxyFactory.
createProxy(SampleHook.class, timestamp);
```

Damit der Übergang aus dem EJB-Container in den OSGi-Container über diesen Weg funktioniert, muss das Class-Objekt des Hook-Interfaces im Beispiel *SampleHook.class* sowohl dem Classloader des EJB-Moduls als auch dem Classloader des OSGi Bundles, das die eigentliche Hook-Implementierung bereitstellt, bekannt sein.

Classloading reloaded

Eine Möglichkeit, die Class-Objekte der Hook-Interfaces sowohl dem Classloader des EJB-Moduls als auch dem Bundle Classloader bekannt zu machen, besteht darin, die Hook-Interfaces ebenfalls in ein OSGi Bundle zu packen und dieses statisch mit in das EAR zu inkludieren. OSGi Bundles sind in der Regel auch als ganz normale JAR-Archive zu verwenden. Zeigt ein entsprechender *Classpath*-Eintrag in dem Manifest des EJB-Moduls also auf das OSGi Bundle mit den Hook-Interfaces, sind die einzelnen Class-Objekte der Hooks für den Classloader des EJB-Moduls sichtbar. Um zu vermeiden, dass das OSGi Bundle mit den Hook-Interfaces einmal statisch als normales JAR in das EAR gepackt wird und zusätzlich noch über die Datenbank bereitgestellt werden muss, wurde der bereits beschriebene *OSGiBundleDeployer* so entworfen, dass er nicht nur Bundles aus der Datenbank (nach-)laden kann, sondern auch „statische" Bundles aus einem fest definierten Verzeichnis des EAR-Archivs installieren und starten kann. Somit ist es möglich, dass Bundles mit Hook-Implementierungen über Standard-OSGi-*Import Package*-Direktiven in ihren Manifest-Dateien auf die Class-Objekte der Hook-Interfaces zugreifen können.

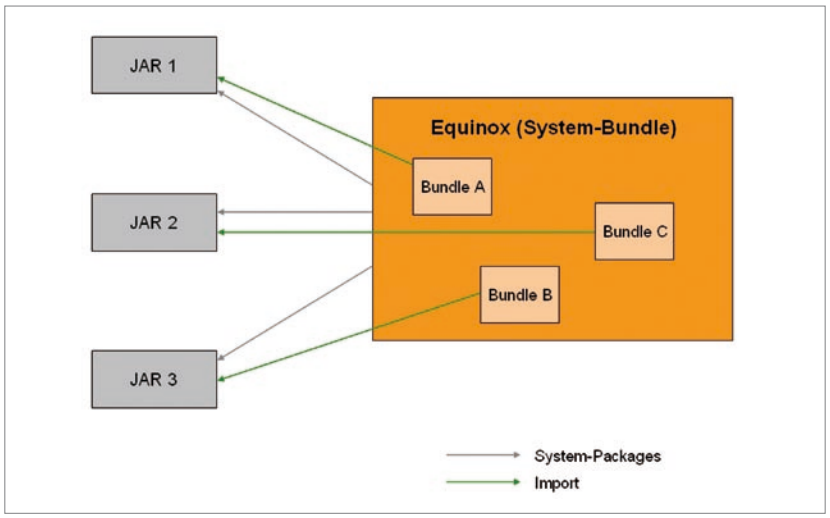

Abb. 2: Zugriff auf System Packages

Eine weitere Möglichkeit, das Problem mit den Hook-Interfaces zu lösen, besteht darin, sie nicht (zusätzlich) in ein OSGi Bundle, sondern tatsächlich nur in ein normales JAR-Archiv zu packen. Aus Sicht des EJB-Moduls ändert sich bei dieser Vorgehensweise nichts. Die Hook-Class-Objekte werden dem Classloader des EJB-Moduls wie vorher über Standard-*Class Path*-Manifest Referenzen bekannt gemacht. Um den Classloadern der OSGi Bundles mit den Hook-Implementierungen allerdings Zugriff auf die Class-Objekte der Hook-Interfaces zu geben, muss ein „OSGi-Hintertürchen" geöffnet werden: System-Packages.

Damit die Class-Objekte der Hook-Interfaces über System-Packages exportiert werden können, muss der Parent Classloader des OSGi Frameworks sie finden können. Im LMS-Projekt wird das OSGi Framework über die Klasse *OSGiBundleDeployer* gestartet, d.h. der Classloader des Moduls, das die Klasse *OSGiBundleDeployer* enthält, muss Zugriff auf alle über System-Packages definierte Klassen haben. Im Fall von JBoss wird der *OSGiBundleDeployer* als MBean Service über ein JBoss-spezifisches SAR-Modul deployed. Damit der Classloader des SAR-Moduls Zugriff auf die Hook-Interfaces bekommt, muss in dem Manifest des SAR-Archivs eine *Class Path*-Referenz auf das JAR-Archiv mit den Hook-Interfaces enthalten sein (Abb. 3). Im Falle von Bea WebLogic können JARs, die System-Packages enthalten, alternativ auch in das Weblogic-spezifische */APP-INF/lib*-Verzeichnis innerhalb des EAR gelegt werden.

Mit der bis hier skizzierten Lösung ist es also möglich, aus dem EJB-Container heraus Services aufzurufen, die innerhalb des OSGi-Containers laufen. Was passiert aber innerhalb einer Hook-Implementierung? Was, wenn eine Hook-Implementierung auf eine Third Party Library zugreifen muss? Und was

OSGi System Packages

In Kapitel 3.8.5 der OSGi Service Platform Core Specification Version 4.1 [5] ist das System Property *org.osgi.framework.system.packages* beschrieben. Über dieses Property können Packages definiert werden, die von einem speziellen Bundle, dem System-Bundle, exportiert werden. Die Syntax des Properties entspricht dabei exakt der Syntax des *Export Package* Manifest Header von Standard-OSGi-Bundles. Alle Packages, die so als System-Packages definiert sind, müssen über den Parent Classloader des OSGi Frameworks aufgelöst werden können. Ist dies der Fall, können Klassen aus System-Packages von anderen OSGi Bundles verwendet werden, sofern die betreffenden Packages in den Manifesten der Bundles über normale *Import Package*-Direktiven importiert werden (Abb. 2).

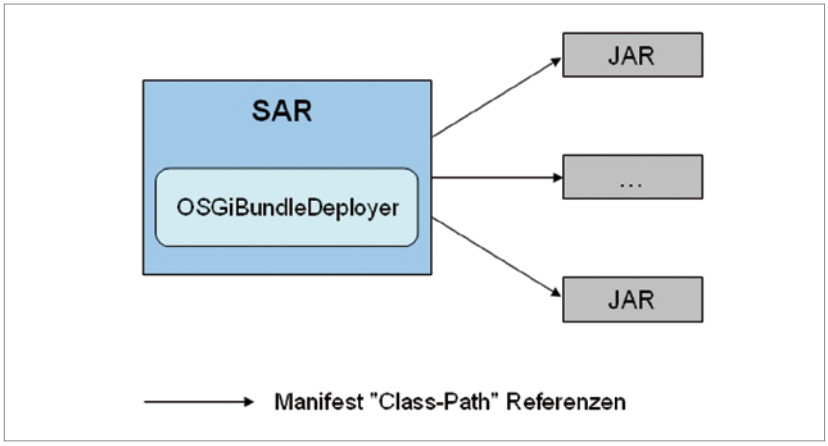

Abb. 3: Manifest Classpath-Referenzen auf normale JAR-Archive

ist zu tun, wenn diese Third Party Library ihrerseits über dynamisches Class-Loading auf Klassen aus dem ursprünglichen OSGi Bundle zugreifen muss? Für genau diese Fragen müssen Antworten gefunden werden, wenn man versucht, innerhalb der Implementierung von OSGi-Services auf Frameworks wie Hibernate [8] zuzugreifen. Da Hibernate in dem genannten Projekt das gesetzte Persistenz-Framework war, musste also ein flexibler Lösungsansatz gefunden werden, um Hibernate und OSGi zu integrieren [9].

In einem ersten Ansatz würde man sicher versuchen, alle verwendeten Third Party Libraries (inklusive Hibernate) in OSGi Bundles umzuwandeln, d.h. zu jeder Library ein passendes OSGi Bundle Manifest entweder von Hand zu schreiben oder von Tools wie dem *bnd*-Tool [10] generieren zu lassen. Problematisch ist allerdings, dass man im Fall von Hibernate je nach Anwendung bzw. Kontext ein separates Bundle Manifest schreiben müsste, da jede Anwendung ja ihre eigenen über Hibernate persis-tierten Klassen mitbringt. Spätestens wenn zur Laufzeit ein zusätzliches Bundle gestartet werden soll, das neue persistierbare Klassen mitbringt, wird selbst dieser Workaround an seine Grenzen stoßen. Ein Ausweg aus diesem Dilemma könnte der Equinox-spezifische Buddy-Mechanismus sein, über den es möglich ist, einem Bundle B, das von Bundle A aufgerufen wird, implizit Zugriff auf alle Klassen aus Bundle A zu geben – ohne dass hierfür explizit *Import Package*-Definitionen in Bundle B notwendig sind.

Context Classloader

Seit Java 1.2 gibt es in der Java API den so genannten *Context Classloader*. Jeder Thread ist in Java mit einem Context Classloader assoziiert. Die Assoziation von Thread und Context Classloader findet dabei in der Regel beim Erzeugen des Threads (bzw. beim Zuweisen eines freien Threads aus dem Thread Pool eines JEE-Applikationsservers) statt. In Standalone-Java-Applikationen (bzw. in Single-Threaded-Anwendungen) entspricht der Context Classloader immer dem System-Classloader, d.h. die Verwendung des Context Classloaders und *Class.forName* zum Suchen einer Klasse liefern hier immer dasselbe Ergebnis. In JEE-Anwendungen ist dies jedoch nicht so, da JEE-Applikationsserver für jedes Deployment-Archiv (zumindest für jedes Top-Level-Deployment-Archiv) eigene Classloader erzeugen. Eine Library, die im Klassenpfad des Applikationsservers liegt und aus einer JEE-Anwendung heraus aufgerufen wird, könnte mit *Class.forName* also nicht auf Klassen zugreifen, die Teil der aufrufenden JEE-Anwendung sind. Verwendet die Library (z.B. log4j) hingegen den Context Classloader des aktuellen Threads, kann die Library sehr wohl auf derartige Klassen (z.B. eigene log4j Appender) zugreifen. Aus diesem Grund wird der Context Classloader quasi von allen Java Libraries für dynamisches Classloading verwendet.

Der nicht unerhebliche Verwaltungsoverhead für die Umwandlung aller direkt und indirekt benötigten Libraries in OSGi Bundles hat im genannten Projekt letztendlich den Ausschlag dafür gegeben, einen anderen Lösungsansatz zu suchen – ein Lösungsansatz, bei dem Third Party Libraries so verwendet werden können wie sie sind, nämlich als normale JARs. Einzige Voraussetzung für die gefundene Lösung ist, dass alle Libraries, die auf dynamischem Classloading basieren, hierfür den Context Classloader des aktuellen Threads verwenden – eine Voraussetzung, die praktisch keine Einschränkung darstellt, da Libraries, die gängigerweise in JEE-Anwendungen zum Einsatz kommen (wie z.B. Hibernate), immer den Context Classloader zum dynamischen Laden von Klassen verwenden.

Die im Projekt gefundene Lösung besteht darin, während der Invokation von Hooks zur Laufzeit den Context Classloader des aktuellen Threads so umzubiegen, dass er auf alle Klassen und Packages, die innerhalb des Bundles mit der Hook-Implementierung sichtbar sind, zugreifen kann. Der Context Classloader stützt sich also auf den Classloader des Bundles, das gerade aufgerufen wird. Listing 1 zeigt den entscheidenden Code der Implementierung dieser *BundleBackedClassLoader*-Klasse. Damit der Lösungsansatz funktioniert, darf der *BundleBackedClassLoader* nicht, wie normalerweise in Java üblich, zuerst an den Parent Classloader delegieren, sondern muss in der *findClass*-Methode unmittelbar die *loadClass*-Methode des aktuellen Bundles aufrufen. Mit dem *BundleBackedClassLoader* als Context Classloader des aktuellen Threads kann ein OSGi-Service nun ohne Weiteres Libraries wie Hibernate verwenden (vorausgesetzt, alle Packages der Libraries sind als System-Packages durch das System-Bundle exportiert, die Libraries sind als normale JARs in dem EAR enthalten und der Parent Classloader des Moduls, das das OSGi Framework gestartet hat, kann über Manifest-*Classpath*-Referenzen auf die JARs zugreifen). Außerdem kann eine Library wie Hibernate nun ohne Weiteres Klassen aus dem aufrufenden Bundle laden (Abb. 4).

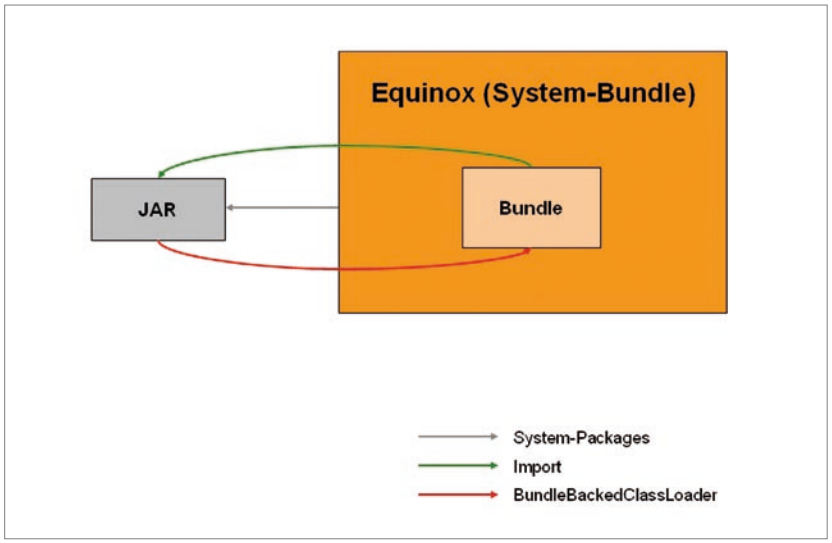

Abb. 4: BundleBackedClassLoader

Als letztes Puzzlestück muss jetzt nur noch sichergestellt werden, dass der Context Classloader automatisch bei der Ausführung von Hook-Implementierungen auf eine Instanz des *BundleBackedClassLoader* umgehängt wird. Da in dem LMS-Projekt bereits Aspektorientierung eingesetzt wurde, lag es nahe, hierfür einen Aspekt mithilfe von AspectJ [11] zu schreiben (Listing 2).

Um einen einfachen Aufhänger für entsprechende Pointcuts zu bekommen und um im Code zu verdeutlichen, dass innerhalb der Implementierungen der Hook-Methoden etwas Besonderes passiert, wurde im genannten Projekt die Entscheidung getroffen, alle Methoden des Hook-Interfaces in der Hook-Implementierung über die Annotation *@HookMethod* zu markieren (Listing 3).

Innerhalb der Methode *doSomething,* der beispielhaften Hook-Implementierung aus Listing 3, können jetzt beispielsweise ohne Probleme Objekte mit Hibernate persistiert werden, deren Klassen entweder in demselben Bundle enthalten sind oder von dem Bundle importiert werden. Damit der *ClassloadingAspect* das Bundle, in dem die *doSomething*-Methode liegt, bestimmen kann, müssen alle Hook-Implementierungen von der Basisklasse *HookImpl* abgeleitet sein. Diese definiert im Wesentlichen nur eine getter- und eine setter-Methode für ein *org.osgi.framework.BundleContext*-Objekt. Injiziert wird der BundleContext beim Erzeugen des Serviceobjekts über eine Service Factory. Die Service Factory implementiert das Interface *org.osgi.framework.ServiceFactory* und wird beim Registrieren des Services über den Aktivator des Bundles erzeugt.

Hat man in eigenen Projekten nicht die Möglichkeit, ein AOP-Framework einzusetzen, kann man über das Konzept der *ServiceFactory* alternativ eine Factory-Implementierung registrieren, die nicht nur das eigentliche Serviceobjekt, erzeugt, sondern auch einen dynamischen Proxy, der das Service-Interface implementiert und bei jeder Invokation an das eigentliche Serviceobjekt delegiert. Damit hat man innerhalb der *InvocationHandler*-Implementierung des Proxy wie in einem AspectJ *around* Advice die Möglichkeit, jeweils vor und nach der Invokation einer Methode einzugreifen, um beispielsweise den Context Classloader umzubiegen.

Entwicklung mit den Eclipse-PDE-Tools

Eclipse bietet mit seiner Plugin Development Environment (PDE) nicht nur eine hervorragende Umgebung für die Entwicklung von Eclipse-Plug-ins sondern ganz allgemein für die Entwicklung von OSGi Bundles. Insbesondere der Manifest-Editor und der Manifest-Builder, der die Konsistenz des Manifests nach jeder Änderung verifiziert und gefundene Fehler zum Beispiel in nicht auflösbaren Import-Beziehungen sofort anzeigt,

Listing 1

```
public class BundleBackedClassLoader extends ClassLoader {
 private Bundle bundle;

 public BundleBackedClassLoader(Bundle bundle) {
  this.bundle = bundle;
 }
 ...
 protected Class<?> findClass(String name)
                                        throws ClassNotFoundException {
  return bundle.loadClass(name);
 }
 ...
 protected Class<?> loadClass(String name, boolean resolve)
  throws ClassNotFoundException {

  Class clazz = findClass(name);
  if (resolve) {
   resolveClass(clazz);
  }
  return clazz;
 }
}
```

Listing 2

```
public aspect ClassloadingAspect {

 public pointcut hookMethodInvocations(HookMethod hookMethod):
        execution(* *(..)) && @annotation(hookMethod);

 Object around(HookMethod hookMethod) : hookMethodInvocations
                                                (hookMethod) {

  ClassLoader oldClassLoader = Thread.currentThread().
                                        getContextClassLoader();
  try {
   Bundle bundle = getBundleContext(thisJoinPoint).getBundle();
   ClassLoader newClassLoader = new BundleBackedClassLoader(bundle);
   Thread.currentThread().setContextClassLoader(newClassLoader);
   return proceed(hookletMethod);
  } finally {
   Thread.currentThread().setContextClassLoader(oldClassLoader);
  }
 }

 private BundleContext getBundleContext(JoinPoint joinPoint) {
  HookImpl hook = (HookImpl)joinPoint.getTarget();
  return hook.getBundleContext();
 }
}
```

Listing 3

```
public class SampleHookImpl extends HookImpl implements SampleHook
{

 @HookMethod
 public void doSomething() {
  ...
 }

}
```

sind eine große Unterstützung bei der Entwicklung eigener OSGi Bundles.

Leider werten die PDE-Tools zur Entwicklungszeit jedoch keine System-Packages aus, sodass direkte Abhängigkeiten zu Klassen und Interfaces, die zur Laufzeit von dem System-Bundle über den System-Package Mechanismus zur Verfügung gestellt werden, erst einmal nicht aufgelöst werden können. Dieses Problem kann jedoch relativ einfach mithilfe eines „Wrapper"-Bundles gelöst bzw. umgangen werden. Das „Wrapper"-Bundle, das entweder als Projekt innerhalb des Eclipse Workspace angelegt werden kann oder über die Target-Platform bereit gestellt wird, referenziert über den Bundle-Classpath Manifest Header alle entsprechenden JAR-Archive und kann die relevanten Packages aus diesen Archiven damit ganz normal mithilfe des Export-Package Headers exportieren.

Im Rahmen des genannten Projekts sind eine Reihe weiterer Eclipse Plugins entstanden, die den Entwickler beim Schreiben von Hook-Implementierungen zusätzlich unterstützen.

Fazit und Ausblick

Durch den Einsatz von OSGi und die Integration in JEE konnten alle eingangs erwähnten Anforderungen, die an das Produkt der Firma Loyalty Partner GmbH gestellt wurden, erfüllt werden. Bundles mit alternativen Hook-Implementierungen können jederzeit dynamisch deployt werden, ohne dass die eigentliche JEE-Enterprise-Anwendung gestoppt werden muss. Dieser Mechanismus kann nicht nur für spezielle Kampagnen, sondern auch z.B. für das Einspielen von Bugfixes oder Patches verwendet werden und funktioniert aufgrund der Tatsache, dass die Bundles über eine Datenbank zur Verfügung gestellt werden, auch in einem Cluster. Der einzige Fall, in dem ein Hot-Deployment nicht funktioniert, ist, wenn ein Bundle neue Abhängigkeiten zu Third Party Libraries mitbringt, die in dem EAR-Archiv noch nicht enthalten und im OSGi-Framework noch nicht als System Packages definiert sind. Dieser Fall stellt jedoch eine echte Ausnahme dar und wurde deshalb in Kauf genommen.

Leider ist es nicht möglich, im Rahmen dieses Artikels auf die ebenfalls sehr interessanten Themen Laufzeitmanagement und Unit Testing einzugehen. Der Vollständigkeit halber sei aber erwähnt, dass auch für diese Themenkomplexe in dem beschriebenen Projekt Lösungen erarbeitet wurden. Beispielsweise wird beim Start des OSGi Frameworks eine JMX MBean installiert, über die u.a. neue Bundles installiert und gestartet werden können.

Der Einzug von OSGi in die Java-Enterprise-Welt scheint nicht mehr aufzuhalten zu sein. Dafür spricht nicht nur, dass die neuesten Generationen der großen JEE-Applikationsserver bereits auf OSGi basieren bzw. bald auf OSGi basieren werden, sondern z.B. auch, dass Projekte wie Spring bereits eigene Unterprojekte für die Integration mit OSGi bereitstellen. Für Anwendungen, die bereits auf Spring basieren, bietet das Projekt „Spring Dynamic Modules" [12] eine interessante Alternative für den Einstieg in OSGi. Für das genannte LMS-Produkt kam „Spring OSGi" (wie das Projekt anfangs genannt wurde) jedoch nicht in Frage, da es zu Beginn des LMS-Projekts in einer zu frühen Phase war und sich außerdem einige der genannten Anforderungen nicht out-of-the-box hätten umsetzen lassen.

Die Autoren sind auf die ersten Ergebnisse der OSGi Enterprise Expert Group (EEG) [13] gespannt. Die OSGi EEG hat sich zum Ziel gesetzt, OSGi für die Verwendung in Enterprise-Anwendungen zu erweitern. Allerdings ist bis zur Fertigstellung dieses Artikels noch kein Termin für eine erste Version einer Spezifikation der OSGi EEG, geschweige denn einer ersten Implementierung bekannt, sodass davon auszugehen ist, dass vorerst proprietäre Lösungen, wie sie dieser Artikel skizziert, gefunden werden müssen, um von der Flexibilität und Modularität von OSGi in Java-Enterprise-Anwendungen profitieren zu können. ■

Dirk Mascher arbeitet als selbstständiger Architekt und Entwickler im Java- und JEE-Umfeld. Als externer Berater und Architekt war er für die Integration von OSGi in das Produkt LMS der Firma Loyalty Partner GmbH verantwortlich. 1997 hat er von C/C++ auf Java gewechselt, seit 1999 entwirft und entwickelt er komplexe Systeme im Java-Enterprise-Umfeld.

Björn Wand arbeitet für die Firma Loyalty Partner GmbH in der Produktentwicklung. Seit drei Jahren liegt sein Fokus im Java- und JEE-Umfeld. Davor war er fünf Jahre bei einem großen Telekommunikationsunternehmen als C++-Entwickler in verschiedenen Projekten rund um Abrechnungssysteme tätig.

Links & Literatur

[1] Sven Haiges: OSGi Serie, in: Java Magazin 7-8.2005
[2] Sven Haiges: Happy Birthday, OSGi!, in: Java Magazin 4.2006
[3] Mirko Jahn: OSGi applied Serie, in: Java Magazin 8-9.2007
[4] Loyalty Partner GmbH: www.loyaltypartner.com
[5] OSGi Spezifikationen: www.osgi.org/Specifications
[6] Equinox: www.eclipse.org/equinox
[7] Jeff McAffer, Simon Kaegi: Eclipse, Equinox and OSGi: www.theserverside.com/tt/articles/article.tss?l=EclipseEquinoxOSGi
[8] Hibernate: www.hibernate.org
[9] Peter Kriens: OSGi and Hibernate: www.osgi.org/blog/2007/06/osgi-and-hibernate.html
[10] bnd Tool: www.aqute.biz/Code/Bnd
[11] AspectJ: www.eclipse.org/aspectj
[12] Spring Dynamic Modules for OSGi Service Platforms: www.springframework.org/osgi
[13] OSGi EEG: www.osgi.org/EEG/HomePage