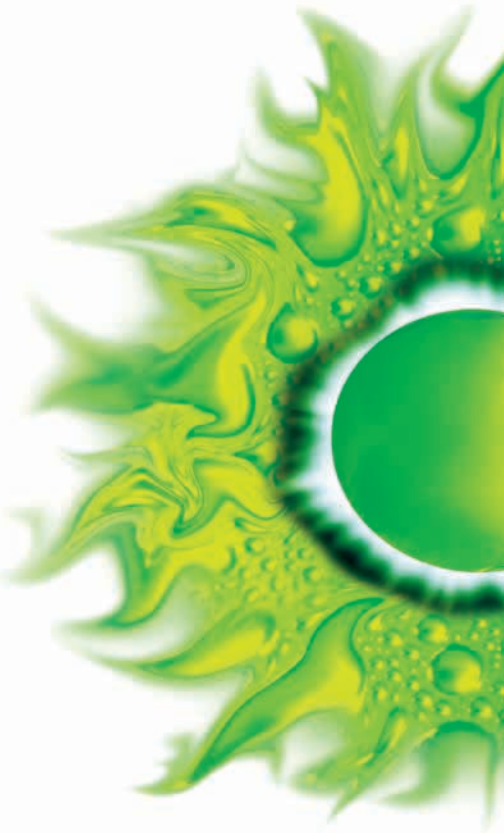
Server-Side Eclipse: Mit Eclipse mehr als Rich Clients entwickeln

Next Step Eclipse

VON MARTIN LIPPERT, BERND KOLB UND GERD WÜTHERICH

Eclipse als Basis zum Bau von IDEs zu verwenden ist längst kalter Kaffee. Eclipse als Plattform für Rich Clients einzusetzen hat sich innerhalb des letzten Jahres ebenfalls etabliert. Was kommt als Nächstes? Die Antwort liegt auf der Hand: serverbasierte Anwendungen! Werfen wir einen genaueren Blick darauf, warum die Eclipse-Technologie im nächsten Schritt die Serverseite erobern wird und welche Möglichkeiten es gibt, Eclipse schon heute für serverbasierte Anwendungen zu nutzen.



Die Eclipse Rich Client Platform hat gezeigt, dass sich weite Teile der Eclipse-Technologie gewinnbringend für normale Business-Anwendungen einsetzen lassen. Wie der Name der Plattform schon vermuten lässt, konzentriert sich RCP allerdings auf die Implementierung von Rich-Client-Anwendungen.

Heutige Anwendungssysteme werden allerdings schon lange nicht mehr als reine Rich-Client-Anwendungen entwickelt. Häufig kommt ein Server zum Einsatz, der dem Klienten Aufgaben abnimmt – wie in einer typischen Multi-Tier-Architektur üblich. Teilweise verzichten größere Systeme sogar völlig auf einen Rich Client – entweder, weil ein ausschließlich Web-basiertes Frontend implementiert wird (mit Web 2.0 erlebt das Web-basierte Frontend ja wieder eine Renaissance) oder weil die Anwendungen überhaupt keine interaktiven Frontends benötigen, da es sich um reine Backend-Anwendungen handelt.

Wieso dann Eclipse?

Sicherlich besteht ein großer Teil der Eclipse Rich Client Platform aus der Workbench und den damit verbundenen Konzepten und Frameworks, um Rich Client UIs zu implementieren. Aber schauen wir einmal unter die UI-Oberfläche eines Systems. Was verbirgt sich dort, wenn man mit der Eclipse-Technologie Anwendungen implementiert?

Das System wird aus Komponenten, den Plug-ins oder OSGi Bundles zusammengesetzt. Die Abhängigkeiten zwischen diesen Komponenten des Systems werden definiert, und es wird exakt deklariert, welcher Teil einer Komponente für andere Teile des Systems sichtbar sein soll. Mithilfe von Eclipse werden diese Abhängigkeiten und Sichtbarkeiten schon während der Entwicklung überprüft, die OSGi Runtime führt diese Prüfung auch zur Laufzeit durch. Die OSGi Runtime ist darüber hinaus auch dafür verantwortlich, dass Komponenten dynamisch (also zur Laufzeit des Systems) hinzugefügt, entfernt oder ausgetauscht werden können. Diese Modularisierungstechnik bildet das Rückgrad jeder auf Eclipse basierenden Anwendung.

Darüber hinaus nutzt man Extension Points und Extensions, um das System offen und flexibel für Erweiterungen zu gestalten. Schnell benutzt man nicht nur die von der Eclipse-Plattform vordefinierten Extension Points, um das System zu erweitern, sondern definiert eigene, um auch die selbst implementierten Komponenten für Erweiterungen zu öffnen – und oft beziehen sich diese Extension Points nicht mehr nur auf Oberflächen.

Betrachten wir beispielsweise eine fiktive Client-Server-Anwendung, mit der die Beratung und die Beantragung von Finanzprodukten realisiert werden. Während die Beratung weitestgehend innerhalb einer Rich-Client-Applikation realisiert ist, müssen für die Beantragung der Produkte verschiedene serverseitige Dienste aufgerufen werden. Neue Produkte können nun einfach – inklusive ihrer Data Transfer Objects, spezieller Services und dem passendem spezialisierten UI – durch ein neues Plug-in zur Rich Client Application hinzugefügt werden. Wäre es nicht verführerisch einfach, auf dem gleichen Wege dem serverbasierten Teil des Systems dieses neue Produkt bekannt zu machen?

Natürlich benötigt der Server nicht das UI des neuen Produktes. Um die (fachliche) Implementierung des Produktes von der konkreten Darstellung zu trennen, teilen wir die Implementierung des neuen Produktes in zwei Komponenten auf.

Die erste Komponente enthält nur die Core-Implementierung des Produktes (alle Nicht-UI-Teile), die zweite Komponente liefert das spezialisierte UI für das Produkt; eine Design-Richtlinie, die sich in der Vergangenheit bereits für Eclipse-RCP-Anwendungen als nützlich erwiesen hat.

Da das System nicht nur über eine Rich-Client-Oberfläche, sondern auch in Teilen über eine Weboberfläche zu bedienen ist, muss das neue Produkt ebenfalls in diese integriert werden. Dem Server müssen folglich sowohl das neue Core Plug-in also auch der neue UI-Anteil für unsere Webapplikation bekannt gemacht werden.

Im Equinox-Incubator-Projekt Server-Side-Eclipse wurde eine Bridge entwickelt, mit der es möglich ist, die OSGi Runtime von einem Servlet aus anzusprechen und so Webanwendungen aus OSGi Bundles zusammenzusetzen.

Land der Möglichkeiten

Wir sehen, dass die grundlegenden Techniken Eclipse-basierter Anwendungen längst nicht auf den Rich Client begrenzt sind. Aber welche Möglichkeiten existieren bereits heute, um auch für serverseitige Anwendungen oder Anwendungsteile diese Techniken nutzen zu können?

Listing 1

```
package de.kolbware.samples.simpleosgi;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

  public void start(BundleContext context) throws
Exception {
    System.out.println("Hello World!!");
  }

  public void stop(BundleContext context) throws
                                        Exception {
    System.out.println("Goodbye World!!");
  }
}
```

Headless Eclipse

Die OSGi Runtime sowie die Extension Registry von Eclipse sind völlig ohne UI (im so genannten Headless-Modus) sofort einsatzfähig. Das Eclipse-SDK selbst bringt hier schon eine Reihe von Beispielen mit:

- Startet man ausschließlich die OSGi Runtime, wird kein UI angezeigt. Die Runtime selbst lässt sich dabei über eine Konsole steuern.
- Der automatisierte Build-Prozess des Eclipse-Projektes ist als Headless-Eclipse-Anwendung realisiert.
- Des Weiteren kann z.B. der Java Code Formatter ohne UI ausgeführt werden oder auch der EMF-Code-Generator hat eine eigene Eclipse-Applikation, welche ohne UI auskommt.

Mit solchen Headless-Anwendungen lassen sich also bereits einfache batchartige Anwendungen auf Basis der Eclipse-Technologie implementieren, die aus Plug-ins zusammengesetzt werden und sowohl die OSGi Runtime als auch den Extension-Point-Mechanismus nutzen können. Listing 1 zeigt ein einfaches OSGi Bundle, welches über die Kommandozeile gestartet und gestoppt werden kann.

Die beiden Methoden *start()* und *stop()* sind dabei für den Lebenszyklus eines Bundles zuständig. Die OSGi Runtime wird bereits mit Eclipse ausgeliefert und befindet sich im *org.eclipse.osgi_<version>.jar* im Eclipse-Plug-in-Verzeichnis. Das JAR und damit der minimale Footprint für eine Eclipse-OSGi-Anwendung beträgt ca. 830 kb. Um die OSGi-Laufzeitumgebung von der Kommandozeile zu starten, rufen wir *java -jar org.eclipse.osgi_3.2.0.v20060328.jar -console* auf. Durch den Befehl *ss* (short status) sehen wir, welche Bundles bereits installiert sind und in welchem Zustand sie sich derzeit befinden. In unserem Fall ist nur die Umgebung selbst aktiviert.

Listing 2

```
> java -jar org.eclipse.osgi_3.2.0.v20060328.jar
                                              -console
osgi> ss
Framework is launched.
id   State    Bundle
0    ACTIVE   system.bundle_3.2.0.v20060328
osgi> install file:///c:/temp/sample/simpleOsgi/
                de.kolbware.samples.simpleosgi_1.0.0.jar
Bundle id is 1
osgi> start de.kolbware.samples.simpleosgi
Hello World!!
osgi> stop de.kolbware.samples.simpleosgi
Goodbye World!!
osgi> uninstall de.kolbware.samples.simpleosgi
```

Durch den Befehl *install <URL>* können wir ein neues Bundle installieren und es danach durch *start <bundleID>* aus dem Zustand *INSTALLED* nach *ACTIVE* überführen (Listing 2).

Um nicht bei jedem Start von Hand Bundles installieren und starten zu müssen, bietet die Runtime die Möglichkeit, in einer Konfigurationsdatei mit dem Namen *config.ini* die zu startenden Bundles und ihre Startlevel anzugeben. Die Startlevel werden durch ein @ gekennzeichnet und sind eine Möglichkeit, die Reihenfolge der Initialisierung zu bestimmen. Die Reihenfolge innerhalb eines Levels ist dabei nicht definiert. Soll ein Bundle nach der Installation direkt gestartet werden, kann dies durch *:start* angegeben werden. Ein Beispiel für eine solche Konfigurationsdatei zeigt Listing 3.

Beim Start der Runtime wird der Programmparameter -configuration *<pathToConfigFolder>* angehängt. Als weitere Ausbaustufe des gerade beschriebenen Vorgehens kann man die Equinox Plug-in Runtime noch hinzunehmen. Dies bietet dem Nutzer die Möglichkeit, weitere Eclipse-Funktionalitäten wie z.B. die Eclipse-Runtime, das Jobs API oder eine der vielen anderen (Nicht-UI-)abhängigen Komponenten zu nutzen. Bei diesem Vorgehen unterscheidet sich die De-

Listing 3

```
osgi.noShutdown=true
osgi.bundles=reference\:file\:C\:/temp/sample/
        simpleOsgi/de.kolbware.samples.simpleosgi_
                                    1.0.0.jar/@4\:start
osgi.framework=file\:C\:/eclipse/32m6/eclipse/
        plugins/org.eclipse.osgi_3.2.0.v20060328.jar
```

finition einer Anwendung nicht von der, die beim Erstellen einer RCP-Anwendung gegeben wird. Das Plug-in *org.eclipse.core.runtime* bietet einen Extension Point *applications* an, welcher beim Start ausgewertet wird und als Startpunkt unserer Anwendung dient. Um unsere Anwendung nun starten zu können, verwenden wir das von Eclipse mitgelieferte *startup.jar* oder den OS-spezifischen Launcher. Mit dem Programmparameter *-application <applicationID>* spezifizieren wir, welche Extension angesprochen werden soll. Auch hierbei muss die OSGi-Konfigurationsdatei angegeben werden oder in dem Unterordner *configuration* vorhanden sein.

Hiermit nun aber eine echte serverbasierte Anwendung zu erstellen wäre sehr mühsam, da ein Web- oder Application Server uns bereits viel Arbeit abnimmt, die wir nicht selbst implementieren wollen.

Web-Container und OSGi

Im nächsten Schritt wollen wir die serverseitigen Anwendungsteile innerhalb eines Webservers (quasi als Application Server) ablaufen lassen. Dabei gibt es grundsätzlich zwei unterschiedliche Ansätze, die beiden Technologien (Webserver und OSGi) zu kombinieren.

1. Die *Servlet-Bridge:* Im Rahmen des Equinox-Incubator-Projekts Server-Side-Eclipse hat Simon Kaegi eine Bridge entwickelt, mit der es möglich ist, die OSGi Runtime von einem Servlet aus anzusprechen und so Webanwendungen aus OSGi Bundles zusammenzusetzen. Der Vorteil dieser Lösung ist, dass der Webserver genau so betrieben werden kann wie eh und je. Dieser Vorteil wiegt besonders schwer, wenn der Webserver zentral administriert wird und auch andere Anwendungen hostet. Diese Servlet-Bridge wird bereits vom Eclipse Communication Framework-(ECF-)Projekt genutzt, um einen zentralen Kommunikationsserver zur Verfügung zu stellen. Diese Lösung setzt allerdings die aktuellen Milestone-Builds der Eclipse-Version 3.2 voraus. Die Servlet-Bridge erlaubt es, Webanwendungen einerseits aus OSGi Bundles zusammenzusetzen und andererseits über die gewöhnliche Java-Servlet-Technologie als Webanwendung zur Verfügung zu stellen. Eine solche Anwendung muss demnach normale Java-Servlets und Webressourcen implementieren. Die Verbindung zum Webserver wird allerdings nicht mehr über eine *web.xml*-Datei hergestellt (oder Ähnliches), sondern über die Servlet-Bridge.

Um die Servlet-Bridge zu nutzen, bietet das Eclipse-Equinox-Incubator-Projekt eine WAR-Datei als Template zum Download an. Das *web.xml* weist hierbei lediglich ein einziges Servlet aus, welches innerhalb der Servlet-Bridge implementiert ist. Dieses Servlet übernimmt hier die Aufgabe des Dispatchers für diejenigen Servlets, welche sich später innerhalb des OSGi-Containers befinden werden. Andere Plug-ins (bzw. OSGi Bundles) können ihre Servlets und Ressourcen mithilfe von zur Verfügung gestellten Extension Points registrieren.

Die Servlet-Bridge ist dabei so implementiert, dass sie mittels eines *IRegistryChangeListener* auf Änderungen an der *ExtensionRegistry* hört. Dadurch ist das dynamische Installieren und Deinstallieren von Bundles und der sich darin befindenden Servlets und Ressourcen möglich. Ein Beispiel hierfür ist bereits im erwähnten WAR-File enthalten und befindet sich in der *plugin.xml* des *org.eclipse.equinox.servlet.ext.jar*, welches auch die Extension Points für die Bridge zur Verfügung stellt (Listing 4).

Listing 4

```
<extension id="testServlet" point="org.eclipse.
                        equinox.servlet.ext.servlets">
 <alias>/sp_test</alias>
 <servlet-class>org.eclipse.equinox.servlet.ext.
                        TestServlet</servlet-class>
</extension>
<extension id="testResource" point="org.eclipse.
                    equinox.servlet.ext.resources">
 <alias>/testresource</alias>
 <base-name>/test</base-name>
 <httpcontext-name>testintest</httpcontext-
                                        name>
</extension>
<extension point="org.eclipse.equinox.servlet.ext.
                                    httpcontexts">
 <httpcontext-name>testintest</httpcontext-
                                        name>
 <path>/test</path>
</extension>
```

Darüber hinaus bietet das einführende Beispiel von Wolfgang Gehner einen weiteren Einblick, wie die Servlet-Bridge eingebunden werden kann (www.infonoia.com/en/content.jsp?d=inf.05.07). Im CVS ist das Incubator-Projekt unter *dev.eclipse.org:/cvsroot/eclipse* im Modul *equinox-incubator* zu finden. Für alle anderen, nicht OSGi-basierten Webapplikationen, die weiterhin außerhalb des Webservers laufen sollen oder müssen, ändert sich nichts. Sie werden nicht durch die OSGi-Webanwendung beeinflusst.

Neben der Modularisierung mithilfe von OSGi Bundles ist der Extension-Mechanismus der zweite wichtige Bestandteil der Eclipse-Runtime, den wir nutzen können, um eine gut strukturierte und erweiterbare Architektur zu realisieren.

2. Der *OSGi-basierte Webserver:* Startet man den Webserver selbst auf Basis der OSGi Runtime als ein Bundle, können wie selbstverständlich auch alle Webanwendungen die zugrunde liegende OSGi Runtime nutzen. Das Eclipse-SDK nutzt diese Art, einen Webserver zu verwenden, um das Hilfe-System von Eclipse zu starten. Dies setzt allerdings voraus, dass man die Start-Prozedur des Webservers beeinflussen kann (damit anstelle des Webservers zunächst die OSGi Runtime gestartet wird, die ihrerseits dann den Webserver zum Laufen bringt). In der hier vorgestellten Implementierung wird zunächst der Webserver, in unserem Fall Jetty, in ein eigenes Plug-in gepackt, um ihn auch weiterhin unabhängig zu halten. Für das Starten des Servers ist ein separates Plug-in zuständig, welches auch einen Extension Point für das Hinzufügen von Webapplikationen zur Verfügung stellt. Hierbei sollte auch dafür Sorge getragen werden, dass nachträglich durch OSGi installierte oder deinstallierte Plug-ins korrekt behandelt werden. Dies kann mithilfe eines *IRegistryChangeListener* erreicht werden. Nach dem Start sowie bei Änderungen an der *ExtensionRegistry* wird mithilfe der zur Verfügung gestellten Java-API des Webservers eine Webapplikation deployt. Um das korrekte Laden von Klassen und Ressourcen zu ermöglichen, muss dem eben deployten Kontext ein Classloader zur Verfügung gestellt werden, der sowohl die Klassen des Webservers kennt als auch das Plug-in, welches die Applikation zur Verfügung stellt. Durch den Eclipse-Classloader-Mechanismus sind ebenfalls alle von der Webapplikation abhängigen Plug-ins sichtbar, soweit sie ihre Packages exportieren. Ist diese kleine Hürde zu Beginn überwunden, steht eine sehr mächtige Lösung bereit. Ein Wermutstropfen ist allerdings, dass man nicht immer die Kontrolle über den Webserver hat, was es demnach auch nicht ermöglicht, zunächst OSGi zu starten.

Listing 5

```
public class SimpleRegistryProvider implements
                                    IRegistryProvider {

 private static Object userToken;
 private static Object masterToken;

 private IExtensionRegistry registry;

 public IExtensionRegistry getRegistry() {
  if (registry == null) {
   registry = RegistryFactory.createRegistry(new
    RegistryStrategy(null, null) {
     public void onStart(IExtensionRegistry registry) {
      // create Contributor (Bundle in "normal" Eclipse
      RegistryContributor extensionpointContrib = new
       RegistryContributor("extensionpointContrib",
        "extensionpointContrib", null, null);
      // read the xml
      registry.addContribution(getClass()
       .getResourceAsStream("extensionpoints.xml"),
        extensionpointContrib, false, null, null,
         masterToken);
      }
     }, masterToken, userToken);
    }
    return registry;
   }

   public static void main(String[] args) {
    // create Registry
    IExtensionRegistry registry = new
     SimpleRegistryProvider().getRegistry();
    // search as normal for extensions and handle them
    // ...
   }
  }
```

Der Extension-Mechanismus

Neben der Modularisierung mithilfe von OSGi Bundles (bzw. Plug-ins) ist der Extension-Mechanismus der zweite wichtige Bestandteil der Eclipse-Runtime, den wir nutzen können, um eine gut strukturierte und vor allem erweiterbare Architektur zu realisieren. Wir haben in den vorangegangenen Abschnitten bereits von diesem Mechanismus innerhalb einer laufenden Eclipse-Anwendung Gebrauch gemacht. Mit der kommenden Eclipse-Version 3.2 wird es möglich sein, den Extension-Point-Mechanismus von Eclipse auch ohne eine OSGi Runtime zu nutzen. Das gestattet es uns, in serverseitigen Anwendungen den Extension-Point-Mechanismus zu verwenden und gegebenenfalls beim Deployment des Systems auf die OSGi Runtime zu verzichten.

In den derzeit verfügbaren 3.2-Meilenstein-Builds der Eclipse-Plattform sind die wesentlichen dafür benötigten Refactorings bereits abgeschlossen. So ist es zum jetzigen Zeitpunkt zwar nötig, das 830 kb große OSGI JAR mit auszuliefern, jedoch nur noch, um die Klassenhierarchie konsistent zu halten. Um den Extension-Point-Mechanismus von Eclipse standalone zu benutzen, muss lediglich ein *IRegistryProvider* implementiert werden. Dieser ist dafür zuständig, die *plugin.xml* bzw. ihr Äquivalent zusammenzusammeln und der *ExtensionRegistry* hinzuzufügen. Danach kann jeder, der Zugriff auf die zentrale *ExtensionRegistry* hat, Plug-ins für einen Extension Point anfragen und auswerten (Listing 5).

Web-Container und Eclipse-UI

Über die reine Verwendung der OSGi Runtime und des Extension-Point-Mechanismus hinaus geht das neue Eclipse Rich AJAX Platform-Projekt-Proposal. In diesem Projekt wird daran gearbeitet, eine Eclipse-ähnliche Workbench mittels AJAX-Technologien für Webanwendungen zu erstellen. Damit wird dem Ent-

wickler die Möglichkeit gegeben, nicht nur die gleiche Modularisierungs- und Komponenten-Technologie der Eclipse Rich Client Platform zu nutzen, sondern auch ähnliche APIs und Programmiermodelle für die Entwicklung von Webanwendungen einzusetzen. Das Projekt-Proposal findet sich unter www.eclipse.org/proposals/rap/. Eine Demo der Technologie kann man im Web unter rap.innoopract.com/webworkbench finden. (Siehe auch Interview auf S. 32.)

Java EE, Application Server und Spring

Reicht ein Web-Container nicht für die serverseitige Anwendung als Ablaufumgebung aus und muss stattdessen ein vollwertiger Java EE Application Server her, stellt sich die nächste Frage: Kann die OSGi Runtime in einem kompletten Application Server genutzt werden?

In Mailing-Listen behaupten zwar immer wieder Teilnehmer, dass sie bereits JBoss oder andere Application Server mit OSGi zusammen zum Laufen gebracht hätten. Fertig nutzbaren Code scheint es aber bisher nirgends zu geben. Dennoch steigt das Interesse an einer OSGi-Integration in Application-Servern ständig. Auf Anfrage des Eclipse Healthcare-Projektes hat sich anscheinend auch das Geronimo-Projekt die Aufgabe gestellt, einen OSGi-Container zu implementieren. Das Problem sind die meist ausgefeilten Classloading-Techniken der im Application Server laufenden Container, da diese oft nicht mit dem Classloading der OSGi-Implementierung harmonieren. Dennoch ist es selbstverständlich auch hier möglich, die bereits im Abschnitt Web-Container und Extension Points beschriebene Technik anzuwenden, soweit es das Classloading erlaubt.

Der Einsatz von schwergewichtigen Application-Servern für größere serverseitige Java-Anwendungen ist längst nicht mehr unumstritten. Vor allem das Spring Framework (www.springframework.org) geht hier einen anderen Weg und fördert die leichtgewichtige Konstruktion von serverseitigen Enterprise-Anwendungen mit Java, ohne dass ein Application Server zum Einsatz kommen muss.

Da das Spring Framework selbst keine Hilfe bietet, um große Anwendungen sauber zu modularisieren, scheint eine Integration mit OSGi vielversprechend. Erste implementierte Prototypen zeigen, dass sich beide Technologien nicht nur recht einfach miteinander kombinieren lassen, sondern sich auch hervorragend ergänzen. So lassen sich mit OSGi Bundles Spring-Konfigurationen in Module verteilen, während der Spring-Dependency-Injection-Mechanismus über Modulgrenzen hinweg einsetzbar bleibt.

Listing 6

```
<beans>
 <bean id="beanA"
    class="de.[…].BeanA">
 </bean>

 <bean class="org.springframework.osgi.service.
                                    OsgiServiceExporter">
  <property name="exportBeans">
   <list>
    <value>beanA</value>
   </list>
  </property>
 </bean>
</beans>
```

Listing 7

```
<beans>
 <bean id="beanB" class="com.[…].BeanB">
  <property name="beanA">
   <ref local=" beanProxy"/>
  </property>
 </bean>

 <bean id="beanProxy"
 class="org.springframework.osgi.service.
                        OsgiServiceProxyFactoryBean">
  <property name="serviceType"><value>
                com.[…].BeanA </value></property>
  <property name="beanName"><value>beanA</
                                    value></property>
 </bean>
</beans>
```

Prinzipiell kann dabei zwischen zwei unterschiedlichen Ansätzen unterschieden werden. Die wesentliche Frage dabei ist, wo der (Spring-)ApplicationContext angesiedelt ist, der letztendlich für die Erzeugung der Beans und deren Verknüpfung verantwortlich ist. So ist beispielsweise eine Integration zwischen Spring und OSGi denkbar, bei der im Gesamtsystem genau ein ApplicationContext existiert, dem über den Extension-Mechanismus Bean-Definitionen aus den einzelnen Bundles hinzugefügt werden können. Nachteil einer solchen Lösung ist jedoch, dass der potenziellen Dynamik von OSGi-Applikationen nicht Rechnung getragen wird: Der ApplicationContext wird genau einmal beim Systemstart initialisiert, Änderungen im Lebenszyklus der einzelnen Bundles während der Laufzeit bleiben unberücksichtigt.

Ein intelligenterer Ansatz zur Integration von Spring und OSGi kann innerhalb des Spring-Sandbox (via CVS-Zugriff unter: pserver:anonymous@cvs.sourceforge.net:/cvsroot/springframework im Verzeichnis *spring/sandbox*) begutachtet werden. In der dort enthaltenen Lösung, die Bestandteil von Spring 2.1 werden soll, besitzt jedes Bundle, welches Bean-Definitionen bereitstellt, einen eigenen ApplicationContext. Um die Bundle-übergreifende Injizierung von Referenzen zu ermöglichen, können Bundles ihre Spring Beans als OSGi-Services exportieren (Listing 6). Gleichzeitig können die so bereitgestellten Beans über einen Proxy-Mechanismus in andere Beans injiziert werden (Listing 7).

Wird das fremde Bundle zur Laufzeit deaktiviert, dann sorgt der Proxy für eine entsprechende Fehlerbehandlung. Neben dem Dependency-Injection-Mechanismus lassen sich in diesem Szenario alle Technologien nutzen, die in Spring zur Verfügung stehen, beispielsweise Verteilung oder Persistenz. Gleichzeitig können über die OSGi-Runtime alle Bundles, aus denen eine Anwendung aufgebaut ist, zur Laufzeit hinzugefügt, aus ihr entfernt oder ausgetauscht werden. Diese Eigenschaften machen die Verbindung von Spring und OSGi zu einer sehr leistungsfähigen Basis für Server-Applikationen. Darüber hinaus lassen sich beispielsweise mit dem Extension-Point-Mechanismus spezielle, selbstdefinierte Erweiterungspunkte definieren, deren Extensions vom Anbieter des Extension Points automatisch dem Spring Framework unter einer dezidierten Konfiguration bekannt gemacht werden. Wir werden in einem zukünftigen Artikel einen detaillierten Blick auf die vielversprechende Kombination der Eclipse-Runtime mit dem Spring Framework werfen.

Ausblick

Die Eclipse-Technologie ist längst nicht mehr nur für die Entwicklung von Client-Anwendungen interessant. Wir haben gezeigt, dass auch serverseitige Anwendungen von der Eclipse-Plattform profitieren können. Die unterschiedlichen Möglichkeiten demonstrieren, dass vielfältige Einsatzszenarien denkbar sind und bereits von unterschiedlichen Projekten genutzt werden. Dass die Eclipse-Technologie auch die Serverseite erobern wird, steht für uns mittlerweile völlig außer Frage.

Martin Lippert ist Senior-IT-Berater bei der it-agile GmbH. Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und Eclipse-Technologie und ist Committer im Eclipse-Equinox-Incubator-Projekt. Kontakt: martin.lippert@it-agile.de.

Bernd Kolb ist freiberuflicher Berater und Coach. Er ist (Mit-)Autor diverser Artikel sowie eines Buches und Sprecher auf Konferenzen. Seine Schwerpunkte liegen auf modellgetriebener Softwareentwicklung sowie Eclipse-Technologien. Kontakt: b.kolb@kolbware.de.

Gerd Wütherich arbeitet als Software-Architekt bei der comdirect bank AG, wo er sich u.a. mit dem Einsatz von Spring und Eclipse-Technologie beschäftigt. Er ist Autor verschiedener Fachartikel und Mitbegründer des Open-Source-Projektes ant4eclipse (ant4eclipse.sf.net). Kontakt: gerd.wuetherich@comdirect.de.

Server-Side Eclipse & More

Eine Reihe von Eclipse-Projekten nutzt bereits oder plant den Einsatz der Eclipse-Runtime-Technologie für serverseitige Anwendungen. Hier ein kurzer Überblick:

- ECF: Das Eclipse Communication Framework realisiert bereits einen Kommunikations-Server, der die Servlet-Bridge aus dem Eqnuinox-Incubator-Projekt nutzt, um OSGi auf dem Server innerhalb eines Webservers ablaufen zu lassen.
- Open Healthcare: Das Open Healthcare-Projekt möchte seine serverseitigen Anteile gerne innerhalb eines Application Server wie Geronimo einsetzen. Passend dazu planen Entwickler des Geronimo-Projektes, einen OSGi-Container für Geronimo zu implementieren.
- Rich AJAX Platform: Die Rich AJAX Platform soll es ermöglichen, ähnlich wie mit der Generic Workbench der Rich Client Plattform Webanwendungen zu implementieren. Dazu wird es ebenfalls nötig, die OSGi Bundles der Anwendung innerhalb eines Webservers ablaufen zu lassen. Auch hier könnte die Servlet-Bridge verwendet werden.
- Eclipse Component Framework: Innerhalb dieses Projektes soll eine Komponenten-Infrastruktur implementiert werden, um typische Business-Anwendungen zu implementieren, ähnlich wie die Rich Client Platform den Entwickler eine solide und erprobte Architektur für Rich Clients bietet. Natürlich setzt auch das Eclipse Component Framework den Einsatz von OSGi auf dem Server voraus.
- Corona: Im Corona-Projekt soll eine SOA-artige serverbasierte Infrastruktur entwickelt werden, die die Realisierung von kollaborativen Systemen unterstützen soll. Beispielsweise soll es möglich werden, in einem verteilten Entwickler-Team Projekte gemeinsam zu benutzen. Deshalb soll auch eine serverseitige und OSGi-basierte SOA-Plattform zur Verfügung gestellt werden.

Server-Side Eclipse, die 2. – Ihre Meinung ist gefragt

Wie es in der nächsten Ausgabe weitergehen soll, können Sie mitbestimmen. Wir haben unter www.eclipse-magazin.de/serverside eine Umfrage eingerichtet, bei der Sie unter den folgenden drei Themenausrichtungen wählen können:

- Spring kombiniert mit OSGi und Extension Registry: Wir betrachten genauer, welche Möglichkeiten es gibt, serverbasierte OSGi-Anwendungen oder Anwendungsteile zu implementieren und dabei gleichzeitig vom Spring-Framework zu profitieren, inklusive Dependency Injection und der Kombination von Spring-Beans und dem Extension-Point-Mechanismus.
- Eclipse OSGi kombiniert mit einem Webserver: Welche Möglichkeiten gibt es, OSGi-basierte Anwendungen in einen Webserver zu deployen? Wie kann man einen Webserver auf OSGi-Basis starten?
- Die Eclipse Extension Registry mit und ohne OSGi: Die Extension Registry lässt sich mit und ohne OSGi nutzen, um Systeme zu flexibilisieren. Wir betrachten im Detail, wie sich Anwendungen mit dem Extension-Point-Mechanismus flexibilisieren lassen, sowohl auf Basis von OSGi als auch ohne OSGi.