

Plus CD!

Im Gespräch ▶

Doug Schaefer, Andrew Overholt, Greg Wilkins, Roy Ganor, u.v.m.

eclipse  
MAGAZIN

5.10

Deutschland € 9,80  
Österreich € 10,80, Schweiz sFr 19,20



# eclipse

www.eclipse-magazin.de

## MAGAZIN

>> BONUS-VIDEO AUF CD:



**Ed Merks:**  
The Unbearable Stupidity of Modeling

- >> Scala IDE for Eclipse 2.8.0
- >> Eclipse Helios Classic
- >> PureMVC for Eclipse
- >> Jetty 8.0.0.M0



# Eclipse Helios!

- ✦ Großer Helios-Jahresrückblick
- ✦ Was gibt's Neues in JDT?
- ✦ Modellieren mit Eclipse Helios
- ✦ Reifeprüfung für Xtext
- ✦ Git-Integration in Eclipse
- ✦ Jetty goes OSGi

## Modeling meets Runtime >> 66

redView: Softwarefabrik für UI-Modelle

## Maven, Spring & RCP >> 79

Unter einen Hut gebracht

## CSS Styling in Helios >> 48

Den Sonnengott verschönern

**„Wir sollten das Innovationsnetzwerk um Eclipse weiter ausbauen.“ >> 84**

Jochen Krause zur Debatte: Quo vadis Eclipse?



Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG



Quellcode auf CD!

Enhanced Extension Points: Spring meets Eclipse RCP

# Integration von Spring in Eclipse RCP

Im zweiten Artikel [1] dieser Serie haben wir bereits aufgezeigt, wie durch den Einsatz von Spring [2] und Spring Dynamic Modules [3] die Entwicklung von OSGi Bundles erheblich vereinfacht werden kann. Wollen wir diese Konzepte auch auf Ebene von Eclipse RCP anwenden, gilt es, einige zusätzliche Herausforderungen zu meistern. In diesem Artikel beschreiben wir, wie aus einer Kombination dieser Technologien eine flexible Lösung realisiert werden kann, in der die individuellen Vorteile beider Welten erhalten bleiben.

von Tobias Forster, Ylli Sylejmani, René Eggenschwiler

OSGi bietet mit der Modularisierung und dem serviceorientierten Programmiermodell eine ganze Reihe neuer Möglichkeiten zur Applikationsentwicklung. Durch die Modularisierung ist es möglich, private und öffentliche Bereiche innerhalb eines Java-Archivs (JAR) zu definieren. Kombiniert man dies mit dem serviceorientierten Ansatz, lassen sich die einzelnen Applikationsteile optimal kapseln. Durch genau diese Kapselung der Bundles ist es wesentlich einfacher, mit verschiedenen Versionen von Bibliotheken umzugehen, da jedes Bundle über seinen eigenen Lebenszyklus und Classloader verfügt. Konflikte, die durch unterschiedliche Versionen einer Bibliothek innerhalb des gleichen

Classloaders verursacht werden, gehören somit der Vergangenheit an.

Das Spring Framework auf der anderen Seite bietet durch Dependency Injection ein mächtiges Werkzeug, um Applikationen einfach zu verdrahten. Dadurch wird umfangreicher Infrastrukturcode nahezu überflüssig. Ei-

## Artikelserie: Maven-gestützte Eclipse-Plug-in-Build-Umgebung

Teil 1: Automatischer Build von Eclipse-RCP-Applikationen

Teil 2: Automatisiertes Testen von OSGi Bundles

Teil 3: Automatisieren von Eclipse Product Builds

**Teil 4: Integration von Spring in Eclipse RCP**

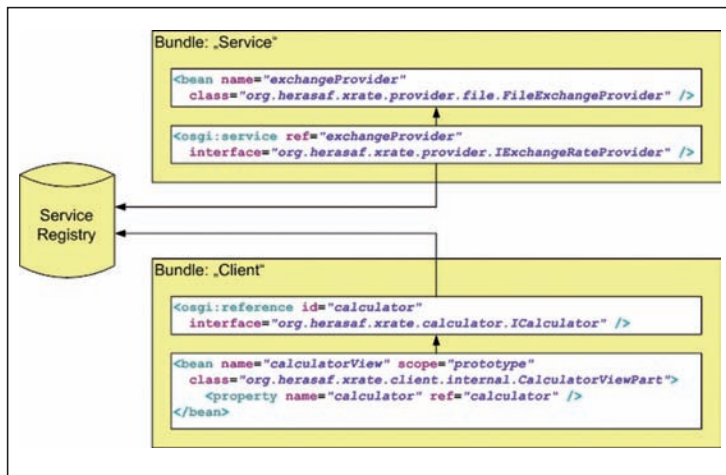


Abb. 1: Verdrahtung mit Spring und Spring Dynamic Modules

**Listing 1: Tracker zum Laden eines OSGi Services**

```
public class CalculatorTracker {
    private ServiceTracker serviceTracker;

    public CalculatorTracker(final BundleContext bundleContext) {
        serviceTracker = new ServiceTracker(bundleContext, ICalculator.class
            .getName(), null);
        serviceTracker.open();
    }

    public ICalculator getCalculator(int timeout) {
        ICalculator calculator = null;
        try {
            calculator = (ICalculator) serviceTracker.waitForService(timeout);
        } catch (InterruptedException e) {
            // do nothing
        }
        return calculator;
    }

    ...
}
```

**Listing 2: Nutzung des Trackers zum Laden des OSGi Services**

```
ICalculator calculator = null;
...
CalculatorTracker tracker = new CalculatorTracker(XRatePlugin
    .getDefault().getBundleContext());
try {
    calculator = tracker.getCalculator(5000);
} finally {
    tracker.close();
}
...
```

gene Factories und ähnliche Konzepte werden nur noch selten benötigt.

Durch den Einsatz von Spring Dynamic Modules als Bindeglied zwischen dem Spring Framework und der OSGi-Umgebung lassen sich beide Welten einfach kombinieren. Musste man bis anhin eine aufwändige Referenzierung von OSGi Services selbst implementieren (Listing 1 und Listing 2), vereinfacht der Einsatz von Spring und Spring Dynamic Modules den Umgang mit OSGi Services fundamental.

Der gezeigte Tracker wird nicht mehr länger benötigt, da Registrierung und Referenzierung der OSGi Services vollständig von Spring Dynamic Modules übernommen werden. Dies geschieht alles über die Spring-Konfiguration, wie aus **Abbildung 1** ersichtlich ist. In den Klassen muss der Entwickler lediglich noch die entsprechenden Setter-Methoden für die zu injizierenden Instanzen definieren.

**Extension Points**

Mit der vorgestellten Lösung können OSGi Bundles einfach verdrahtet werden. Betrachtet man jedoch Eclipse RCP, stellt man fest, dass mit dem Extension-Point-Konzept ein anderer Ansatz zur Anwendung kommt. Der Einsatz von *Extension Points* ermöglicht das Einhängen von eigenen Implementierungen und eigenem Verhalten in die Komponenten des Eclipse RCP Frameworks. Damit einher geht der Verlust der Kontrolle über den Lebenszyklus der Klasse, da weder der Aufrufzeitpunkt bekannt ist, noch, ob diese Klassen überhaupt aufgerufen werden. Dies wird vollständig durch das Framework bestimmt. Die Herausforderung besteht nun darin, diese beiden Welten so zu kombinieren, dass durch Spring erzeugte Instanzen in den Lebenszyklus des Eclipse RCP Frameworks eingebettet werden können.

**Spring Framework Utility: Klein, aber oho!**

Genau diese Lücke schließt das Spring Framework Utility [4] von Martin Lippert [5]. Es erlaubt die Verwendung von Spring Beans aus dem *ApplicationContext* in den *Extension Points* von Eclipse RCP. Damit können Klassen, die RCP-Code beinhalten, einfach mit Spring instanziiert werden. Mit der Klasse *SpringExtensionFactory*, die durch das Spring Framework Utility zur Verfügung gestellt wird, können die Spring Beans anschließend direkt im *Extension Point* referenziert werden. Dies erfolgt über die Angabe des vollständigen Klassennamens der *SpringExtensionFactory*, gefolgt von einem Doppelpunkt und dem Namen der zu referenzierenden Spring Bean. Üblicherweise würde hier ansonsten der vollständige Name der RCP-Klasse eingetragen. **Abbildung 2** zeigt die Verwendung der *SpringExtensionFactory*.

Die Verbindung von Spring und Eclipse RCP ist möglich, indem sich das Spring Framework Utility anstelle der eigentlich vorgesehenen Klasse in Eclipse RCP einhängt. Wird der *Extension Point* aufgerufen, lädt die *SpringExtensionFactory* den *ApplicationContext* des jeweiligen Bundles und gibt die Bean mit dem angege-



benen Namen zurück. Der Zugriff auf den *Application Context* des jeweiligen Bundles ist möglich, da Spring Dynamic Modules diesen nach der Erzeugung als Service in der OSGi Service Registry registriert.

### Foreign Exchange Calculator Client – Das Beispiel

Wir wollen dies nun am Beispiel des Foreign Exchange Calculator Clients zeigen und ihn auf das Spring Framework Utility umstellen. Den Foreign Exchange Calculator kennen wir bereits aus den bisherigen Artikeln dieser Serie. **Abbildung 3** beschreibt die Struktur der Beispielapplikation.

Im Client benutzen wir ein Objekt vom Typ *ICalculator* für die Umrechnung von Schweizer Franken in Fremdwährungen. Dieses Objekt wird als Service vom Bundle *Calculator* zur Verfügung gestellt. Um die Servicereferenz zu erhalten, haben wir bisher auf reine OSGi-Funktionalität (*ServiceTracker*) gesetzt.

Mit der Umstellung auf die neue Lösung können wir den bisher verwendeten *ServiceTracker* wie auch den dazu nötigen Aufruf in der View (*CalculatorViewPart*) vollständig entfernen. Stattdessen fügen wir in der View ein neues Attribut vom Typ *ICalculator* und die entsprechende Setter-Methode ein. Das Spring Framework kann somit die entsprechende Instanz injizieren. Damit werden die Abhängigkeiten zu OSGi weiter reduziert.

Mit dieser Anpassung haben wir sämtliche Änderungen im Code vorgenommen, welche für die Umstellung nötig sind. Als Nächstes wird die Spring-Konfiguration als XML unter *META-INF/spring* erstellt. Neben der Referenzierung des *ICalculator* Service über Spring Dynamic Modules muss auch eine Instanz von *CalculatorViewPart* erstellt werden, in die wir den Service injizieren. Listing 3 zeigt die dafür benötigten Konfigurationen.

Die eigentliche Integration in den *Extension Point* von Eclipse RCP nehmen wir in der Datei *plugin.xml* vor. Hier ersetzen wir unter dem *Extension Point* *org.eclipse.ui.views* den Klassennamen *org.herasaf.xrate.client.internal.CalculatorViewPart* durch den Aufruf der *SpringExtensionFactory* und dem Namen der zu ladenden Bean aus dem *Application Context*. Diese Änderung ist in Listing 4 ersichtlich.

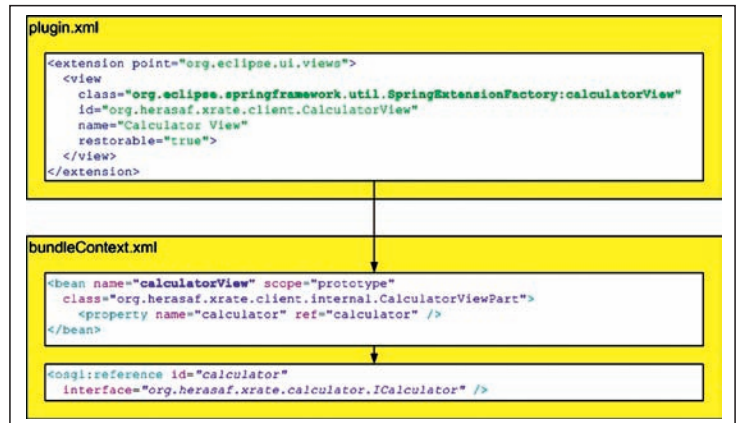


Abb. 2: Verdrahtung mit Spring und Spring Dynamic Modules

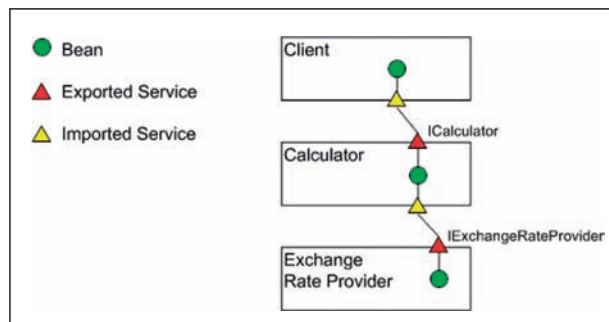


Abb. 3: Aufbau des Währungsrechners in drei Bundles

### Auswirkungen auf den Maven-Build-Prozess

Damit haben wir die nötigen Anpassungen im Bundle vorgenommen. Als Letztes muss noch sichergestellt werden, dass der Maven-Build-Prozess [6] die Änderungen auch berücksichtigt. Dafür sind Anpassungen in der Projektkonfiguration (POM) des Client-Bundles nötig. In den Properties für das *Bundle Plugin for Maven* [7] können nun die OSGi-Imports entfernt werden. Stattdessen muss aber ein Import mit dem Pfad *org.eclipse.springframework.util* eingefügt werden, damit der Zugriff auf die *SpringExtensionFactory* möglich ist. Zusätzlich muss eine weitere Dependency für das Spring Framework Utility hinzugefügt werden. Diese ist nötig, damit Maven beim Build die entsprechenden Klassen findet und das Bundle automatisch in die Target Platform kopiert.

#### Listing 3: Verdrahtung des Clients mit Spring und Spring Dynamic Modules

```

<bean name="calculatorView" scope="prototype"
  class="org.herasaf.xrate.client.internal.CalculatorViewPart">
  <property name="calculator" ref="calculator" />
</bean>

<osgi:reference id="calculator"
  interface="org.herasaf.xrate.calculator.ICalculator" />

```

#### Listing 4: Integration des Spring Beans in den Extension Point

```

<extension point="org.eclipse.ui.views">
  <view
    class="org.eclipse.springframework.util.SpringExtensionFactory:calculatorView"
    id="org.herasaf.xrate.client.CalculatorView"
    name="Calculator View"
    restorable="true">
  </view>
</extension>

```



## Eclipse Buckminster, b3 & p2

Eclipse Magazin: Hi Thomas, hi Henrik! Wie würdet ihr das Buckminster-Projekt in einigen Worten beschreiben?

**Thomas Hallgren:** Buckminster ist ein Werkzeug zum Assemblieren, Builden und Veröffentlichenden von Anwendungen. Es nutzt vorhandene Eclipse-Technologien wie die Eclipse Resource Platform, Project Builder, Team Provider, das p2-Provisionierungsframework und den PDE Build. Im Gegensatz zu den meisten anderen Build-Tools baut Buckminster den Workspace, d. h. es führt den genau gleichen Build in einer Headless-CI-Umgebung durch wie in der Eclipse IDE. Das bedeutet, dass die Features und Bundles, die in der IDE gebaut werden, von Buckminster ohne zusätzliche Konfiguration headless gebaut werden können. Ein weiterer Vorteil ist, dass der Build auf Warnmeldungen und Fehler reagieren kann, die vom Manifest Compiler, Schema Compiler und anderen Standard-Eclipse-Buildern verursacht werden.

**EM:** Welche Verbesserungen hat Buckminster im Helios-Release zu bieten?

**Hallgren:** Die wichtigsten Ergänzungen sind Git-Support, unsere neue vollständige Dokumentation sowie die Unterstützung für die Headless-Ausführung von JUnit und EcEmma Code Coverage. Eine Liste der neuen Features findet sich auch in Henrik Lindbergs Blog unter: <http://henrik-eclipse.blogspot.com/2010/05/buckminster-36-new-noteworthy.html>.

Auch wenn es streng genommen nicht zum Buckminster-Projekt gehört, denke ich, dass auch Johannes Utzig's Buckminster Plug-in für Hudson CI eine Erwähnung wert ist. Das Plug-in hat viele Verbesserungen erfahren und stellt nun eine sehr nützliche Ergänzung für Hudson dar.

**EM:** Was ist der Unterschied zwischen Buckminster und anderen Build-Lösungen bei Eclipse, insbesondere das b3-Projekt?

**Lindberg:** Eclipse b3 ist als Nachfolger sowohl von Buckminster und den PDE Build gedacht (und potenziell auch für andere Build-Technologien). Es umfasst eine Menge von Konzepten, die von diesen Technologien abgedeckt werden, und vereint diese in einem einzigen Modell. Das Modell wird durch eine sehr mächtige DSL auf Basis von Xtext bereitgestellt. Buckminster ist ein reifes und bewährtes Build-System, das seit 2006 im Einsatz ist. Eclipse b3 ist hingegen noch in der Anfangsphase der Entwicklung. Zurzeit verfügt b3 noch nicht über eine eigene Engine für die Build. Unser Plan ist es, Buckminster als Build Engine für das erste Release von b3 zu nutzen.

### Porträt



**Thomas Hallgren** ist Co-Lead und Hauptarchitekt des Eclipse-Buckminster-Projekts sowie Committer in den Eclipse-Projekten b3 und p2. Thomas ist Mitbegründer von Cloudsmith Inc. und Mitglied des Eclipse Architecture Councils.



**Henrik Lindberg** ist Gründer und CTO von Cloudsmith mit über 20 Jahren Berufserfahrung als Entwickler innovativer Infrastruktur- und Anwendersoftware. Er ist Leiter der Eclipse-b3- und Buckminster-Projekte und Committer bei Eclipse p2 und EPP.

Was wir hauptsächlich betonen möchten, ist, dass der Build-Prozess von Buckminster genau so funktioniert wie der in der Eclipse IDE. Was Buckminster leistet, ist das Auffinden und die Materialisierung von Komponenten sowie die Orchestrierung verschiedener Aktionen. Weder Buckminster noch b3 stellen einen Ersatz für PDE selbst dar – lediglich für den veralteten und unpopulären Headless PDE Build. Wir denken, dass dies wichtig ist, da es immer noch die „ursprüngliche Logik“ ist, die für das Zusammenfügen der Einzelteile sorgt.

**EM:** Auf der b3-Projektseite kann man lesen, dass Eclipse b3 aus den beiden Komponenten b3 Engine und b3 Site Aggregator besteht. Können Sie die beiden Komponenten kurz beschreiben?

**Lindberg:** b3 nutzt das Eclipse Modeling Framework (EMF) und Xtext. Es umfasst Repository-Verwaltung, Materialisierung, Discovery, Auflösung und den Build. Die Konzepte ähneln denen von Buckminster, sind aber in vielen Fällen generalisiert und bieten zusätzlich eine Trennung von Aufgabenbereichen. Aus der Perspektive des Anwenders sind die wichtigsten Features:

- Einfachheit! Die Dinge, die jeder typischerweise bei der Durchführung eines Builds benötigt, sind sehr einfach umzusetzen. Große Sorgfalt wurde auch darauf verwendet, zusätzliche Bearbeitungsvorgänge so einfach wie möglich zu halten.
- Der b3-Editor ist ein vollständiger Eclipse-Texteditor mit Syntaxfarb- hervorhebung, Codevollständigkeit, Quick Fixes und vielem mehr.
- Zusätzlich zu der Tatsache, dass b3 eine DSL für Build-bezogene Konstrukte ist (die „Build-Einheiten“, ihre Abhängigkeiten und die Aktionen, die den Build-Prozess durchführen), ist es auch möglich, die tatsächliche Build-Logik in b3 zu schreiben (ohne auf Java oder ANT zurückgreifen zu müssen). Aber auch das Aufrufen von Java-Code oder ANT-Skripten ist problemlos möglich.
- Die Einfügung zusätzlicher Prozesse und Modifikationen der Build-Logik ist mittels einer „Before/After/Around“-Semantik in der Logik genauso möglich, wie die Anpassung Build-bezogener Konstrukte (d. h. Manipulation von Dependencies, Filter, Properties etc.)

**EM:** Und was hat es mit dem b3 Aggregator auf sich?

**Hallgren:** Der b3 Aggregator nutzt eine Reihe von p2-Repositories als Input und erzeugt ein neues p2-Repository als Output. Er ermöglicht eine feingranulare Kontrolle darüber, welche Artefakte das resultierende Repository enthalten sollen. Da der p2-Planner bei der letzten Entscheidung darüber zum Einsatz kommt, welche Artefakte kopiert werden sollen, ist das resultierende Repository immer kohärent. Alle Artefakte können ohne Konflikte installiert werden. Eine umfassende Liste möglicher Anwendungsfälle findet sich auf der b3-Aggregator-Wiki-Seite.

**EM:** Was sind die nächsten Schritte für Build und Provisioning in Eclipse?

**Hallgren:** Als nächsten Schritt planen wir, Buckminster weiter zu verbessern und es zum Backend für b3 zu machen. Das wird die Art und Weise, einen Build-Prozess aufzusetzen, sehr vereinfachen, da b3 es erlaubt, alles in einem Modell zu definieren, möglicherweise sogar in einem einzigen File. Wir wollen einen einheitlichen Weg etablieren, um Artefakte zusammenzubauen und in p2-Repositories zu veröffentlichen.



## Tipps und Tricks

In der Praxis ergibt sich durch den Einsatz des Spring Framework Utility das eine oder andere Problem, das man aber mit ein paar Kniffen in den Griff bekommt. Benutzt man die *SpringExtensionFactory* beispielsweise in *Extension Points*, die kurz nach dem Start der Applikation aufgerufen werden, kann es zu Fehlern kommen. Der Grund dafür ist, dass zu diesem Zeitpunkt noch nicht sämtliche Bundles gestartet werden, um gegenüber dem Benutzer eine schnelle Reaktion in Form eines Splash Screens zu ermöglichen. Davon betroffen sind auch die Bundles von Spring Dynamic Modules. Würden zuerst sämtliche Bundles geladen, träte eine zu große Verzögerung ein, die den Benutzer verwirren könnte. Sobald die Applikation jedoch zu laden beginnt, sind die meisten Bundles aktiviert und dieses Problem tritt nicht mehr auf. Zu diesen *Extension Points* gehört zum Beispiel der häufig verwendete *org.eclipse.core.runtime.applications*, der eine eigenständige Applikation definiert. Wir empfehlen deshalb, bei solch früh aufgerufenen *Extension Points* auf den Einsatz von Spring zu verzichten.

Je nach Einsatz der Beans in den *Extension Points* ist es sinnvoll, bei jedem Aufruf eine neue Instanz zu erzeugen. Beispielsweise soll ein Editor jedes Mal neu instanziiert werden, wenn der entsprechende *Extension Point* aufgerufen wird, da wir ansonsten mehrere geöffnete Fenster oder Register mit dem genau gleichen Inhalt haben. Da Spring standardmäßig die Beans unter dem Scope „singleton“ erzeugt, besteht über den gesamten Lebenszyklus des Bundles nur eine einzige Instanz. Möchte man dieses Verhalten ändern, kann bei den entsprechenden Beans der Scope explizit auf *prototype* gesetzt werden. Dadurch wird bei jeder Anfrage der Bean eine neue Instanz erzeugt. Je nach Anwendung kann es jedoch vorteilhaft sein, immer die gleiche Instanz zu nutzen. Es muss daher von Fall zu Fall unterschieden werden, welches Verhalten benötigt wird.

## Fazit

Beim Spring Framework Utility handelt es sich um eine kleine, aber sehr nützliche Bibliothek. Der größte Vorteil liegt in der Möglichkeit, Bundles über alle Schichten hinweg mit Spring verknüpfen zu können, angefangen bei OSGi Services bis hin zu Klassen, welche in den *Extension Points* von Eclipse RCP benutzt werden. Durch den Einsatz dieser Bibliothek lässt sich vielfach OSGi-naher Infrastrukturcode durch eine einfache Konfiguration ersetzen. Dies spart Wartungsaufwand und hilft, Abhängigkeiten zu anderen Bundles zu reduzieren.

## Rückblick

Blicken wir zurück auf die vorgestellten Lösungen in dieser Artikelreihe, so ist erkennbar, dass der Aufbau eines integrierten Maven-Build-Prozesses für eine OSGi-basierte Entwicklungsumgebung nach wie vor eine Herausforderung darstellt. Dabei stellen sich dem Entwickler grundsätzliche Probleme in den Weg, wie die

unterschiedlichen Konzepte der eingesetzten Werkzeuge und ihre fehlende Kompatibilität. Viele dieser Probleme können jedoch mit einfachen Kniffen und Anpassungen gelöst werden. Dadurch wird die Nutzung einer automatischen Build-Umgebung wesentlich vereinfacht und unterstützt den Entwickler, anstatt ihn zu behindern. Dennoch sind Verbesserungen in der Kombination von OSGi, Eclipse RCP und Maven unbedingt nötig, soll die Akzeptanz bei den Entwicklern erhöht werden – dies, obschon die Stabilität und Reife von OSGi weit fortgeschritten ist. Ankündigungen der Entwicklergemeinschaft, weitere und stärker integrierte Plug-ins für Maven, wie beispielsweise Tycho [8], zu entwickeln, lassen aber darauf hoffen, dass eine einfache und integrierte Lösung schon bald Realität werden könnte.

Auf der Webseite von HERAS [9] können sämtliche Lösungen und weiterführende Informationen aus und zu den Artikeln dieser Serie heruntergeladen werden.



**Tobias Forster** arbeitet als Software Engineer bei Zühlke Engineering in Schlieren (Schweiz). Seine Schwerpunkte liegen in der Entwicklung von Unternehmensanwendungen mit Java EE, OSGi, Eclipse RCP, Spring und Maven.



**Ylli Sylejmani** arbeitet als Software Engineer im Institut für Interne Technologien und Anwendungen an der Hochschule für Technik in Rapperswil (Schweiz). Mit Erfahrung im Bereich der Eclipse-Technologien, SOA, Maven und Spring ist er für die Wartung und Weiterentwicklung des Open-Source-Projekts HERAS zuständig.



**René Eggenschwiler** ist Softwarearchitekt und Projektleiter bei Siemens IT Solutions, Professional Services in der Schweiz. Zudem ist er eines der Kernmitglieder im Open-Source-Projekt HERAS. Er entwickelt Lösungen in den Bereichen Enterprise Computing, SOA und Security. Dabei hilft ihm seine mehrjährige Erfahrung mit Spring, Maven, Hibernate und JEE.

## Links & Literatur

- [1] Automatisiertes Testen von OSGi Bundles: Eclipse Magazine 03/2010
- [2] Spring Framework: <http://www.springsource.org/>
- [3] Spring Dynamic Modules: <http://www.springsource.org/osgi/>
- [4] Spring Framework Utility: <http://martinlippert.blogspot.com/2008/10/new-version-of-spring-extension-factory.html>
- [5] Webseite von Martin Lippert: <http://www.martinlippert.com/>
- [6] Maven: <http://maven.apache.org>
- [7] Apache Felix: <http://felix.apache.org/>
- [8] Sonatype Tycho: <http://tycho.sonatype.org/>
- [9] Eclipse Rich Client Platform Reference Architecture bei HERAS: <http://www.herasaf.org/> (engineering@herasaf.org)