In wenigen Schritten zum automatisierten PDE Build

Build My Plug-in!

>> NILS HARTMANN UND GERD WÜTHERICH

Mit dem PDE Build stellt die Eclipse IDE ein mächtiges Build-Werkzeug zur Verfügung, mit dem Plug-ins, Features und RCP-Anwendungen automatisiert gebaut werden können. Allerdings ist das Aufsetzen des automatisierten Builds mit einigen Stolpersteinen versehen. In diesem Artikel wird gezeigt, wie in wenigen Schritten ein lauffähiger PDE Build aufgesetzt und von der Kommandozeile ausgeführt werden kann.

Innerhalb der Eclipse IDE werden Plug-ins in Form von Plug-in-Projekten entwickelt. Für den Export dieser Plug-in-Projekte in deployfähige Plug-ins stellt das Plug-in Development Environment (PDE) den interaktiven „Plug-in Export Wizard" bereit. Um Plug-in-Projekte automatisiert bauen zu können, ohne dass Sie dabei manuell eingreifen müssen, enthält das PDE darüber hinaus den sog. PDE Build, ein automatisiertes Build-System für Plug-ins und weitere Eclipse-Artefakte wie Features und Products. Der automatisierte PDE Build wird z.B. dann relevant, wenn Sie einen Build in ein Continous Integration System wie CruiseControl [1] einbetten möchten.

Obwohl der PDE Build auf Basis von Ant arbeitet, kann er nicht direkt mit einer Standard-Ant-Installation ausgeführt werden. Vielmehr starten Sie dafür eine „headless"-Eclipse-Instanz, die ohne GUI arbeitet. Diese Eclipse-Instanz führt mithilfe des Eclipse-eigenen „antRunners" ein generisches Buildfile aus, das im Normalfall im ersten Schritt die zu bauenden Projekte aus der Versionsverwaltung auscheckt. Darauf aufbauend, generiert der PDE Build eine Reihe weiterer Build-Skripte, die für das Bauen der einzelnen Projekte verantwortlich sind, und führt diese nacheinander aus. Durch die Tatsache, dass der PDE Build nicht mit einer Standard-Ant-Installation, sondern aus Eclipse heraus mithilfe des antRunners ausgeführt wird, können in den Ant-Skripten des PDE Builds spezielle Ant-Tasks verwendet werden, die auf die Eclipse-internen Datenstrukturen zugreifen. So können etwa die Abhängigkeiten zwischen Plug-ins und die daraus resultierenden Klassenpfade ermittelt werden.

Das Beispiel

Das Einrichten eines automatisierten PDE Builds soll hier anhand eines kleinen Beispiels, das aus zwei Plug-in-Projekten und einem Feature-Projekt besteht, demonstriert werden (Abb. 1).

Das *de.example.service*-Plug-in ist über seine importierten Packages einerseits vom *de.example.model*-Plug-in als auch vom *org.eclipse.osgi*-Plug-in ab-

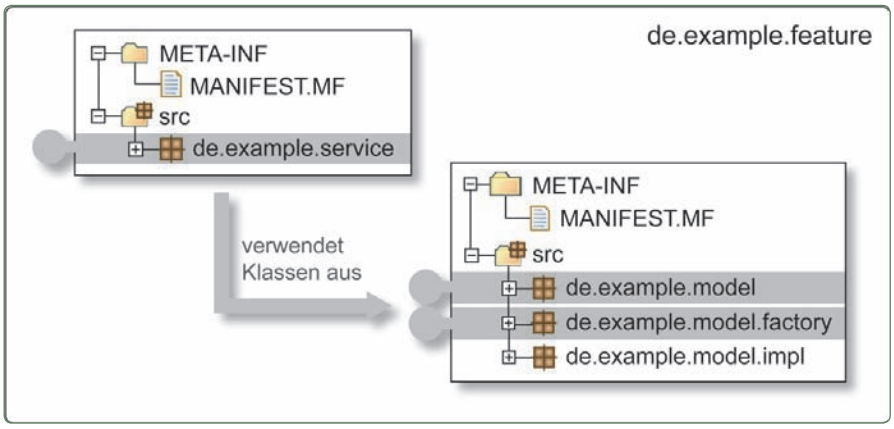

Abb. 1: Abhängigkeiten des Beispiel-Projektes

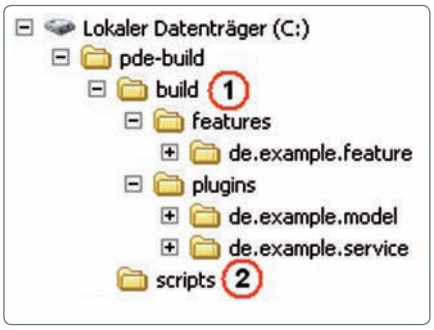

Abb. 2: Verzeichnisstruktur des Build-Projektes

hängig. Das *model*-Plug-in hat keinerlei Abhängigkeiten. Beide Plug-ins werden in dem Feature *de.example.feature* zusammengefasst, da der PDE Build auf oberster Ebene nur mit Features arbeitet. Das komplette Beispielprojekt sowie alle von uns im Rahmen dieses Artikels erstellten Skripte und Konfigurationen finden Sie auf der CD, die diesem Heft beiliegt. Für unser Beispiel verwenden wir den PDE Build aus der Eclipse Version 3.3, das im Verzeichnis C:\ECLIPSE installiert ist. Da sich mit Eclipse 3.3 einige Konfigurationsveränderungen ergeben haben, können die Beispiele leider nicht unverändert mit Eclipse 3.2 verwendet werden.

Vorbereiten der Verzeichnisstruktur

Die vom PDE Build zu bauenden Plug-in-Projekte müssen sich in einem speziellen Verzeichnis, dem Build Directory, befinden, wie in Abbildung 2 (rot markierte 1) gezeigt wird. Das Build-Directory kann frei gewählt werden und enthält die Unterordner PLUGINS und FEATURES, die ihrerseits wieder die zu bauenden Plug-in- und Feature-Projekte enthalten. Der PDE Build arbeitet somit nicht direkt auf den Plug-in-Projekten, die sich in einem Eclipse Workspace befinden. In unserem Beispiel werden die zu bauenden Projekte unter dem Verzeichnis C:\PDE BUILD\BUILD abgelegt.

Im Normalfall sorgt der PDE Build dafür, dass die zu bauenden Plug-in- und Feature-Projekte aus der Versionsverwaltung in die entsprechenden Unterverzeichnisse des Build Directory ausgecheckt werden. Allerdings kann dieser Schritt übersprungen werden, wenn Sie die zu bauenden Projekte selber in den entsprechenden Verzeichnissen bereitstellen möchten. Im Folgenden werden die zu bauenden Plug-ins zunächst händisch aus ihrem Workspace in das Build Directory kopiert. In einem zweiten Schritt wird dann gezeigt, wie Sourcen während der Ausführung des PDE Builds automatisch aus einer Versionsverwaltung ausgecheckt werden können.

Schritt 1.1: Erstellen der Build-Konfiguration

Der PDE Build ist in Form generischer Build-Skripte im Plug-in *org.eclipse.pde.build* implementiert, das Bestandteil der Eclipse-Distribution ist. Für die konkrete Instanziierung eines PDE-Build-Prozesses müssen diese generischen Build-Skripte konfiguriert werden. Das *org.eclipse.pde.build*-Plug-in enthält im Verzeichnis TEMPLATES eine Reihe von Beispielen verschiedener Build-Konfigurationen. Als Grundlage für unsere eigene Konfiguration wählen wir die Dateien im Unterverzeichnis *headless-build*. Bitte kopieren Sie die drei Dateien aus diesem Ordner in das SCRIPTS-Verzeichnis, das sie eben angelegt haben. Die *allElements.xml*-Datei stellt ein Ant Buildfile dar, das als eine Art „Callback“ vom PDE Build verwendet wird. Das Target *allElementsDelegator* in der Datei wird vom PDE Build aufgerufen, um ein weiteres, beliebiges Target für alle Ihre Top-Level-Elemente auszuführen. Damit dieses Target ordnungsgemäß funktioniert, muss lediglich die *ant*-Anweisung angepasst und dort im *property*-Subtask der Name (genauer: die ID) des Features eingetragen werden, das gebaut werden soll. Wenn Sie mehr als ein Feature bauen wollen, müssen Sie den Ant-Eintrag für jedes zu bauende Feature duplizieren (Listing 1).

Neben dem Target *allElementsDelegator* muss das Buildfile *allElements.xml* für jedes Feature ein *assemble*.feature-id-target enthalten. Sie finden dafür in der Datei bereits ein Beispiel-Target, das Sie bitte für jedes Ihrer Features kopieren und entsprechend umbenennen:

```
<target name="assemble.de.example.feature">
 <ant antfile="${assembleScriptName}" dir="${
                                buildDirectory}"/>
</target>
<target name="assemble.de.example.another.feature">
 <ant antfile="${assembleScriptName}" dir="${
                                buildDirectory}"/>
</target>
```

Die zentrale Konfigurationsdatei des PDE Builds ist die *build.properties*-Datei. Darin wird z.B. festgelegt, wo der PDE Build die Plug-ins findet, die Sie kompilieren möchten. Außerdem legen Sie in dieser Datei fest, wo der PDE Build weitere, „fertige“ Plug-ins findet, die das Buildsystem benutzt, um die Abhängigkeiten des Plug-ins aufzulösen. Im ersten Schritt müssen auch nur diese Properties gesetzt werden. Im Property *buildDirectory* wird das Verzeichnis gesetzt, in dem sich Ihre Plug-ins und Features befinden.

Listing 1

```
<target name="allElementsDelegator">
  <ant antfile="${genericTargets}" target=
                                    "${target}">
   <property name="type" value="feature" />
   <property name="id" value="de.example.feature" />
  </ant>
  <ant antfile="${genericTargets}" target=
                                    "${target}">
   <property name="type" value="feature" />
   <property name="id" value="
            de.example.another.feature" />
  </ant>
</target>
```

Setzen Sie diesen Wert also bitte auf Ihr BUILD-Verzeichnis:

```
buildDirectory=c:/PDE Build/build
```

Beim Entwickeln eines Plug-ins in Eclipse wird eine Target-Plattform benutzt, die diejenigen Plug-ins enthält, gegen die Ihre eigenen Plug-ins kompiliert werden sollen. Standardmäßig ist die Target-Plattform das Eclipse-Installationsverzeichnis selber, Sie können jedoch auch über PREFERENCES | TARGET PLATFORM ein anderes Verzeichnis auswählen, das, analog zum Eclipse-Installationsverzeichnis, über ein PLUGINS- und ein FEATURES-Unterverzeichnis verfügt.

Die Target-Plattform müssen Sie in jedem Fall auch in den *build.properties* für Ihren PDE Build konfigurieren. Dazu verwenden Sie das Property *baseLocation*, das den Pfad zu Ihrer Target Platform enthält. In unserem Beispiel nutzen wir dafür unsere Eclipse-Installation:

```
baseLocation=c:/eclipse
```

Zur konfigurierten Target-Plattform können Sie über den Preferences-Dialog in Eclipse weitere Plug-ins hinzufügen, die sich nicht direkt im Target-Plattform-Verzeichnis befinden. Wenn Sie von dieser Möglichkeit Gebrauch gemacht haben, müssen Sie auch diese Plug-ins in Ihrer *build.properties*-Datei eintragen. Verwenden Sie hierzu die Property *pluginPath*, mit dem Sie dem Buildprozess eine Liste von ganzen Ordnern oder einzelnen Plug-ins hinzufügen können: Wenn Sie einen Ordner angeben, erwartet der PDE Build, dass sich unterhalb des Ordners ein *feature* bzw. *plugins*-Verzeichnis befindet. Alle Features und Plug-ins dieser Unterordner werden der Target-Plattform hinzugefügt. Ein einzelnes Plug-in oder Feature fügen Sie hinzu, indem Sie das Verzeichnis des Plug-ins, resp. die JAR-Datei des Plug-ins, in die Liste aufnehmen. Um die einzelnen Pfade voneinander zu trennen, verwenden Sie den auf Ihrem System gültigen Pfad-Trenner, unter Windows also das Semikolon und unter Unix den Doppelpunkt. Um z.B. das Eclipse Modeling Framework (EMF) sowie das Plug-in *com.myplugin* in Ihre Target-Plattform aufzunehmen, verwenden Sie folgende Property:

```
pluginPath=c:\emf;c:\moreplugins\com.myplugin.jar
```

Schritt 1.2: Ausführen des PDE Builds

Nachdem Sie die Konfiguration des PDE Builds wie in Schritt 1.1 vorgenommen haben, können Sie Ihre Plug-ins mit dem

> Nach der Konfiguration kann das Plug-in mit dem PDE Build gebaut werden

PDE Build bauen. Um das Starten des Builds zu vereinfachen, legen Sie eine Batch-Datei *build.bat* im Verzeichnis C:\PDE BUILD\SCRIPTS mit folgendem Inhalt an: (bitte beachten Sie, dass die Namen der beiden JAR-Dateien je nach Eclipse-Version evtl. abweichen können und Sie daher die Namen evtl. anpassen müssen):

```
java -jar
c:\eclipse\plugins\org.eclipse.equinox.launcher_
                                    1.0.0.v20070530.jar
-application org.eclipse.ant.core.antRunner
-f c:\eclipse\plugins\org.eclipse.pde.build_
                        3.3.0.v20070531\scripts\build.xml
-Dbuilder=c:\PDE Build\scripts %*
```

Um den PDE Build auszuführen, müssen Sie jetzt nur noch diese Batch-Datei ausführen. Wenn das Skript beendet ist, werden Sie im Verzeichnis C:\PDE BUILD\BUILD die eben vom PDE Build generierten Skripte finden, daneben das Verzeichnis I.TESTBUILD, in dem sich eine Zip-Datei befindet, die Ihr fertig gebautes Feature sowie die beiden Plug-ins enthält.

Schritt 2.1: Einbindung der Versionsverwaltung

Damit der PDE Build Ihre Sourcen vor dem Bauen auschecken kann, müssen Sie dem Buildsystem mitteilen, in welchen Repositories Sie Ihre Plug-ins abgelegt haben. Dazu verwendet der PDE Build *map*-Dateien, die den in Eclipse verwendeten *Team Project Set*-Dateien ähneln. Eine *map*-Datei enthält eine Liste mit Plug-ins oder Features und die Information, in welcher Versionsverwaltung Sie an welcher Stelle abgelegt sind.

Im Folgenden wird zunächst gezeigt, wie Sie Ihre Projekte aus einem CVS-Repository auschecken können. Am Ende des Artikels werden Sie erfahren, wie Sie vorgehen müssen, um ein Subversion-Repository zu verwenden. Eine *map*-Datei entspricht in ihrem Format einer Java-Properties-Datei. Die einzelnen Einträge haben dabei das folgende Format:

```
{plugin|feature|fragment}@elementId = CVS, tag,
                                cvsroot[, weitere Angaben]
```

Als Schlüssel verwenden Sie also den Typ (*plugin*, *feature* oder *fragment*) und den Namen eines Projekts. Als Wert geben Sie zunächst den Typ der Versionsverwaltung (hier: *CVS*) an, gefolgt vom tag (z.B. *HEAD*) und dem CVSROOT, unter dem Ihr Projekt zu finden ist. Der PDE Build geht dabei davon aus, dass die Verbindung CVSROOT und *elementId* den kompletten Pfad innerhalb Ihrer Versionsverwaltung zu dem jeweiligen Projekt darstellt. Unter weitere Angaben kann z.B. noch ein Passwort angegeben werden. Eine genaue Beschreibung der einzelnen Parameter können Sie der Online-Hilfe von Eclipse entnehmen [2].

Für unser Beispiel-Projekt legen Sie bitte im *build*-Ordner ein Verzeichnis MAPS an und erstellen dort die Datei *de.example.map*:

```
plugin@de.example.model=CVS,HEAD,:pserver:
                anonymous@localhost:/cvs-repository
plugin@de.example.service=CVS,HEAD,:pserver:
                anonymous@localhost:/cvs-repository
feature@de.example.feature=CVS,HEAD,:pserver:
                anonymous@localhost:/cvs-repository
```

Nach dem Anlegen der *maps*-Datei müssen Sie dem PDE Build noch mitteilen, dass Ihre Sourcen nun aus der Versionsverwaltung ausgecheckt werden sollen. Dazu kommentieren Sie das Property *skipFetch* in der *build.properties*-Datei mit einem „#“ aus (es genügt nicht, dieses Property auf *false* zu setzen):

```
#skipFetch=true
```

Bevor Sie jetzt einen erneuten Build starten, löschen Sie aus dem BUILD-Ordner das PLUGINS- und FEATURES-Verzeichnis, um sicherzustellen, dass die Sourcen auch tatsächlich aus dem Repository ge-

laden werden. Wenn Sie nun den Build ausführen, werden Sie sehen, dass der PDE Build Ihre drei Projekte aus dem CVS-Repository auscheckt und in das C:\PDE BUILD\BUILD-Verzeichnis kopiert. Anschließend werden die Projekte wie gewohnt gebaut, sodass Sie nach dem Ende des Builds unter C:\PDE BUILD\BUILD\I.TESTBUILD die zip-Datei mit den fertigen Features und Plug-ins finden.

Schritt 2.2: Versionierung der map-Dateien

In der *map*-Datei legen Sie nicht nur fest, welche Plug-ins und Features zusammen gehören und gemeinsam gebaut werden sollen, sondern auch deren jeweilige Version. Während in unserem Beispiel alle Projekte vom „HEAD" stammen, können die Projekte natürlich auch mit jedem anderen Tag oder Branch versehen werden. Um sich diese Informationen zu merken, versionieren wir künftig auch die *map*-Datei. Wenn Sie im Nachhinein dann z.B. einmal eine Fehlerbehebung eines Ihrer Plug-ins bauen und ausliefern müssen, können Sie anhand der versionierten *maps*-Datei feststellen, in welcher Konstellation das Plug-in ursprünglich gebaut wurde, und auf dieser Grundlage dann die Fehlerbehebung erstellen. Nachdem Sie die Fehlerbehebung erstellt haben, vertaggen Sie die veränderten Projekte, passen die Versionsnummer in der *map*-Datei an und vertaggen auch diese. Mit diesem Tag starten Sie dann Ihren PDE Build, der die neue Version Ihrer Projekte baut. Um den PDE Build anzuweisen, vor dem Starten Ihre *map*-Datei auszuchecken, kommentieren Sie zunächst in der *build.properties*-Datei die *skipMaps*-Property aus:

```
#skipMaps=true
```

Anschließend setzen Sie die folgenden Properties. Vorlagen dafür finden Sie in der *build.properties*-Datei, Sie müssen diese nur entsprechend anpassen:

```
mapsRepo=:pserver:anonymous@localhost:
                                    /cvs-repository
mapsRoot=de.example.maps
mapsCheckoutTag=HEAD
```

Mit der Property *mapsRepo* wird festgelegt, in welchem Repository Sie Ihre *map*-Datei abgelegt haben, und die *mapsRoot*-Property zeigt auf den Pfad innerhalb Ihres Repositories, an dem sich die *map*-Datei befindet. Die Version, mit der die *map*-Datei ausgecheckt werden soll, legen Sie mit *mapsCheckoutTag* fest.

Da der PDE Build jetzt sowohl Ihre Source-Projekte als auch die benötigten *map*-Files direkt aus der Versionsverwaltung lädt, können Sie nun das komplette BUILD-Unterverzeichnis aus dem PDE BUILD-Verzeichnis entfernen. Wenn Sie danach mit dem *build.bat*-Skript den Build starten, wird zunächst Ihre *map*-Datei ausgecheckt. Dann werden Ihre Projekte ausgecheckt und der Build gestartet. Neben dem Auschecken der *map*-Datei bietet Ihnen der PDE Build auch die Möglichkeit, Ihre *map*-File nach dem Build zu vertaggen – z.B. mit der aktuellen Buildnummer. Sie können später dann anhand dieses Tags nachvollziehen, welche *map*-Datei und damit welche Versionen Ihrer Projekte Sie bei dem Build verwendet haben. Setzen Sie dazu zunächst die Property *tagMaps*:

```
tagMaps=true
```

Um das Vertaggen nach erfolgreichem Build auszuführen, fügen Sie dem *postBuild*-Target in der *customElements.xml*-Datei einen Aufruf des *tagMapFiles*-Target hinzu:

```
<target name="postBuild">
 <antcall target="gatherLogs" />
 <antcall target=»tagMapFiles»/>
</target>
```

Mit dem Property *mapsTagTag* können Sie das Tag festlegen, mit dem Ihre *map*-Datei versehen wird. Diese Property ist mit der *buildId* vorbelegt, sodass es ausreicht, Ihre Build-Id (z.B. eine Versionsnummer) beim Start des Builds mit anzugeben, um Ihre *map*-Datei damit zu vertaggen:

```
Build.bat –DbuildId=Release_1_0_0
```

Schritt 2.3: Initialisierung des Build-Prozesses

Bislang haben Sie die Artefakte, die durch einen Buildlauf erzeugt wurden, vor dem Ausführen des nächsten Builds von Hand gelöscht. Um Ihnen diese Arbeit abzunehmen, erweitern wir das *preSetup*-Target in der *customTargets.xml*-Datei. Dieses Target wird vom PDE Build vor dem Auschecken der *map*-Datei ausgeführt und kann dazu genutzt werden, beliebige Initialisierungen etc. vorzunehmen. Wir nutzen dieses Target, um künftig das komplette Build-Verzeichnis zu löschen. Damit stellen wir sicher, dass der PDE Build in jedem Fall die *map*-Datei und alle Source-Projekte aus der Versionsverwaltung auscheckt. Fügen Sie dazu den folgenden *delete*-Aufruf innerhalb des *preSetup*-Targets ein:

```
<target name="preSetup">
<delete dir="${buildDirectory}"/>
</target>
```

Analog zu dem *preSetup*-Target sind in der *customTargets.xml*-Datei für die verschiedenen Phasen des Buildprozesses weitere *Callback*-Targets definiert, die Sie nutzen können, um eigene Funktionalität zu implementieren.

Exkurs: Arbeiten mit Subversion

Nachdem wir bislang gezeigt haben, wie Sie den PDE Build mit einem CVS-Repository verwenden, soll im Folgenden noch die Verwendung von Subversion demonstriert werden. Damit Sie aus dem PDE Build heraus auf ein Subversion Repository zugreifen können, müssen Sie das *Subversion PDE Build plugin* installieren, da der PDE Build von Haus aus keine Subversion-Unterstützung bietet.

Listing 2

```
<target name="getMapFiles"
depends="checkLocalMaps" unless="skipMaps">
 <property name="mapsCheckoutTag" value="HEAD" />
 <svn javahl="false">
  <export srcurl="${mapsRepo}"
    revision="${mapsCheckoutTag}"
    destpath="${buildDirectory}/maps" force="true"/>
 </svn>
</target>
...
<target name="tagMapFiles" if="tagMaps">
 <svn javahl="false">
  <mkdir url="${mapsRepo}/tags/${mapsTagTag}"
   message="Created by PDE Build ${buildId}"/>
  <copy srcUrl="${mapsRepo}/
                ${mapsCheckoutTag}/${mapsRoot}"
   destUrl="${mapsRepo}/tags/${mapsTagTag}/
                                ${mapsRoot}"
   message="Tagged from PDE Build ${buildId}"/>
 </svn>
</target>
```

Bitte laden Sie dazu das Plug-in von der Projekt-Homepage auf Sourceforge herunter [3] und kopieren die in der zip-Datei enthaltenen Verzeichnisse PLUGINS und FEATURES in Ihr Eclipse-Verzeichnis nach C:\ECLIPSE.

In der *customTargets.xml*-Datei müssen Sie zunächst die Targets anpassen, die mit den *map*-Files arbeiten, also *getMapFiles* und *tagMapFiles*, um diese von CVS auf Subversion umzustellen. Hier muss jetzt der *svn*-Task, der mit dem *Subversion PDE Build plugin* ausgeliefert wird, benutzt werden (Listing 2).

Stellen Sie jetzt Ihre *map*-Datei auf Subversion um. Das Format einer Zeile ähnelt dem Format der CVS-*map*-Dateien:

```
{plugin|feature|fragment}@elementId = SVN, tagPath,
                    repositoryUrl, preTagPath, postTagPath
```

Der Schlüssel ist identisch mit dem Schlüssel in CVS-*map*-Dateien. Als Wert geben Sie zunächst die Art der Versionsverwaltung, in unserem Fall als Subversion, mit dem Wert „SVN“ an. Die weiteren Angaben benötigt das Subversion Plug-in, um einen kompletten URL für den Zugriff des Repositories zu erstellen. Dazu hängt es den *preTagPath*, den *tagPath* und den *postTagPath* an den repositoryUrl. Der Eintrag *plugin@example=SVN,trunk,svn://localhost/,source,example* führt so zum Repository-URL *svn://localhost/source/trunk/example*. Weitere Dokumentation und Beispiele dazu finden Sie auf der Projekt-Home des Subversion PDE Plug-ins. Um auf das Beispiel-Projekt zuzugreifen, passen Sie die *map*-Datei wie folgt an:

```
plugin@de.example.model=SVN,trunk,svn://localhost/
                                    ,,de.example.model
plugin@de.example.service=SVN,trunk,svn://localhost/
                                    ,,/de.example.service
feature@de.example.feature=SVN,trunk,svn://localhost/
                                    ,,de.example.feature
```

In der *build.properties*-Datei müssen Sie nun noch das Subversion Repository für die *map*-Datei konfigurieren:

```
mapsRepo=svn://localhost
#mapsRepo=:pserver:anonymous@localhost:/
                                    cvs-repository
mapsRoot=de.example.maps
mapsCheckoutTag=trunk
```

Damit ist Ihr Build auf Subversion umgestellt. Wenn Sie das „build“-Skript jetzt ausführen, werden sowohl Ihre *map*-Datei als auch die Source-Projekte aus dem Subversion-Repository ausgecheckt.

Fazit

Wir haben Ihnen in diesem Artikel einen ersten Einstieg in den PDE Build aufgezeigt. Sie sind damit jetzt in der Lage, Ihre Plug-ins und Features direkt aus der Versionsverwaltung auszuchecken und zu bauen. Der PDE Build bietet Ihnen aber noch sehr viel mehr Möglichkeiten, auf die wir im Rahmen dieses Artikels nicht eingehen konnten. Sie können beispielsweise auch ganze RCP-Anwendungen bauen oder innerhalb des Buildprozesses automatische Tests ausführen lassen. Nähere Informationen dazu bietet Ihnen u.a. der Artikel „Build and Test Automation for plug-ins and features“ auf der Eclipse-Homepage [4] oder auch die Online-Hilfe von Eclipse. Zum Schluss möchten wir Sie noch auf das Open-Source-Tool *pluginbuilder* hinweisen [5]. Dieses Eclipse Plug-in bietet Ihnen mit grafischen Wizards und Editoren eine sehr einfache und elegante Möglichkeit, einen PDE Build einzurichten.

Nils Hartmann arbeitet als Softwareentwickler und -architekt in Hamburg. Neben der Konzeption und Entwicklung von JEE-Anwendungen beschäftigt er sich mit der Entwicklung von Eclipse Plug-ins. Er ist Mitbegründer des Open-Source-Projektes ant4eclipse (ant4eclipse.sf.net).Kontakt: nils@teha-systems.de.

Gerd Wütherich ist selbständiger Softwarearchitekt. Er verfügt über langjährige Erfahrung in der Realisierung großer, javabasierter Anwendungssysteme. Darüber hinaus ist er regelmäßiger Referent auf Konferenzen und Autor verschiedener Fachartikel. Weitere Informationen unter www.gerd-wuetherich.de. Kontakt: gerd.wuetherich@comdirect.de.

>>Links & Literatur

[1] cruisecontrol.sourceforge.net

[2] help | plug-in development environment guide | tasks | advanced pde build topics

[3] sourceforge.net/projects/svn-PDE Build/

[4] www.eclipse.org/articles/Article-PDE-Automation/automation.html

[5] www.pluginbuilder.org