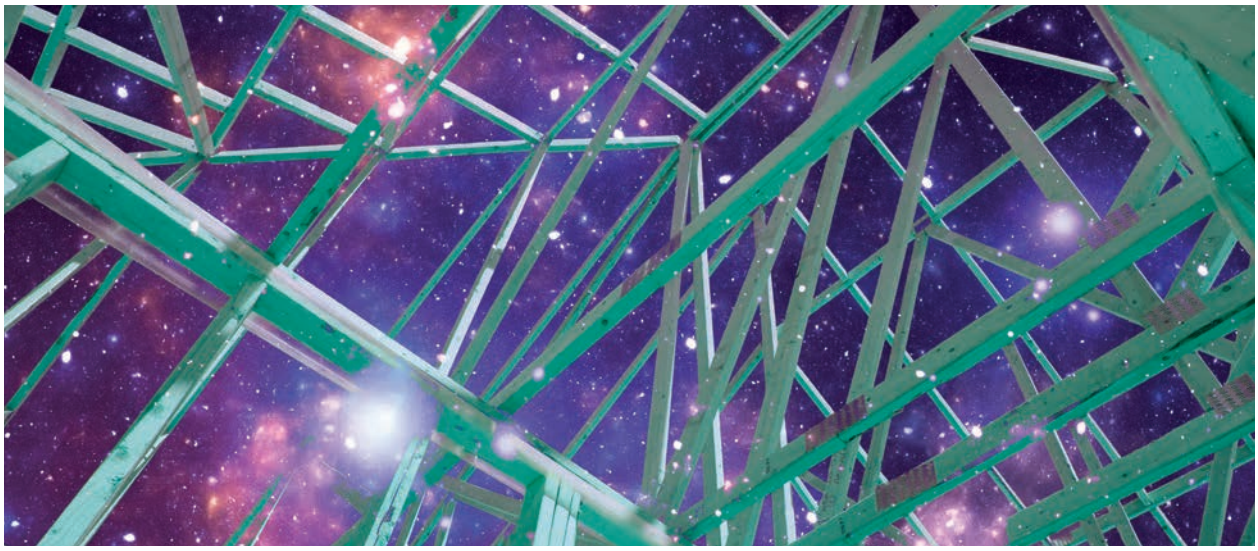

Eclipse Builds mit Maven Tycho

Bau dir dein eigenes Eclipse

Viele Unternehmen setzen auf Eclipse als IDE, reichern diese aber um eigene Plug-ins und ein eigenes Branding an. Wie in [1] beschrieben, war der Prozess zur Erstellung der eigenen Eclipse-Distribution bisher recht umständlich und kompliziert, wird aber mit der Common Build Infrastructure und dem CBI-Build-Prototyp deutlich vereinfacht. Neben der Möglichkeit, den Sourcecode der Eclipse-Plattform selbst zu kompilieren, kann auf Basis bestehender Eclipse-Pakete und Maven Tycho die eigene Eclipse-Distribution mit bekannten Mitteln aus Eclipse selbst heraus gebaut werden.

von Dirk Fauth

Dieser Artikel setzt auf dem Artikel zu Maven Tycho aus der Eclipse Ausgabe 4.11 auf und wiederholt an einigen Stellen die grundlegenden Punkte nochmal. Der Kasten „Hilfreiche Plug-ins“ gibt erste Anhaltspunkte zum richtigen Umgang mit den Maven Builds.

Projektstruktur

Um sich die Arbeit an späterer Stelle zu vereinfachen, ist es sinnvoll, ein Multi-Module-Projekt mit einem *parent*-Projekt anzulegen, über das der Build gestartet wird und die grundlegenden Plug-in-Konfigurationen enthält.

Tipp

Während m2e als Version standardmäßig 0.0.1-SNAPSHOT vorgibt, werden neue Eclipse-Plug-ins standardmäßig mit der Version 1.0.0-qualifier versehen. Bei der Erstellung neuer Plug-ins und POM-Dateien muss darauf geachtet werden, dass die Versionen übereinstimmen, da es ansonsten beim Build zu Problemen kommt. Man sollte sich also vorab darüber Gedanken machen, mit welcher Versionierung man arbeiten möchte, und diese entsprechend in *POM* und *MANIFEST* anpassen. An dieser Stelle gibt es leider noch keine automatische Verknüpfung. *SNAPSHOT* und *qualifier* werden von Maven Tycho automatisch miteinander in Verbindung gesetzt.

Hierzu ruft man in Eclipse den Wizard für ein neues Maven-Projekt über New | Other | Maven | Maven auf. Da das *parent*-Projekt nur eine *pom.xml* enthalten wird, reicht es aus, ein einfaches Projekt ohne Archetype zu erzeugen. Auf der Folgeseite müssen Group ID, Artifact ID, Version und Packaging angegeben werden (**Abb. 1**). Zu beachten ist zum einen die Versionierung (Kasten: „Tipp"), zum anderen das Packaging, das auf *pom* gesetzt werden muss. Nach der Erstellung des Projekts über den Wizard kann der *src*-Ordner gelöscht werden, da das Projekt nur aus der *pom.xml* besteht. In der POM (Listing 1) werden die Maven-Plug-ins konfiguriert, die im weiteren Verlauf benötigt werden. Dazu gehören das Tycho-Plug-in selbst, das Plug-in zur Auflösung der Target Platform und das Plug-in für die Testunterstützung. Optional kann auch das Plug-in für die Source-Plug-in-Generierung eingebunden werden.

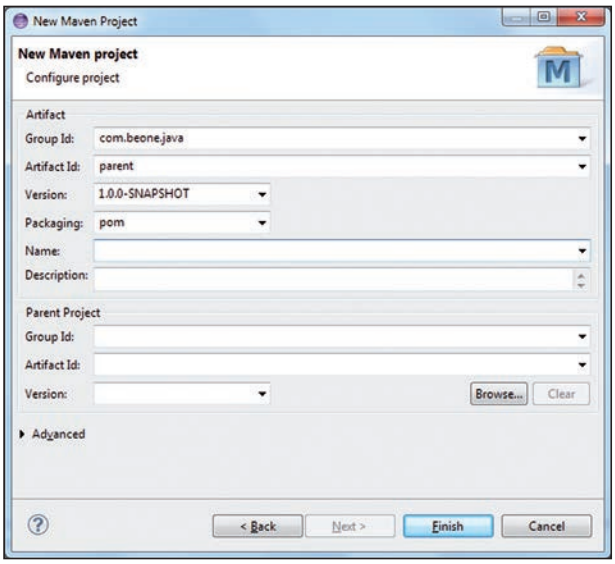

Abb. 1: Maven Project Wizard für das „parent"-Projekt

Listing 1

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.beone.java</groupId>
  <artifactId>parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <packaging>pom</packaging>

  <properties>
    <tycho-version>0.15.0</tycho-version>
  </properties>

  <repositories>
    <repository>
      <id>juno</id>
      <layout>p2</layout>
      <url>http://download.eclipse.org/releases/juno</url>
    </repository>
  </repositories>

  <build>
    <plugins>
      <!-- Plugin to enable tycho -->
      <plugin>
        <groupId>org.eclipse.tycho</groupId>
        <artifactId>tycho-maven-plugin</artifactId>
        <version>${tycho-version}</version>
        <extensions>true</extensions>
      </plugin>

      <!-- Plugin for target platform -->
      <plugin>
        <groupId>org.eclipse.tycho</groupId>
        <artifactId>target-platform-configuration</artifactId>
        <version>${tycho-version}</version>
        <configuration>
          <resolver>p2</resolver>
          <pomDependencies>consider</pomDependencies>
          <environments>
        <environment>
          <os>win32</os>
          <ws>win32</ws>
          <arch>x86</arch>
        </environment>
          </environments>
        </configuration>
      </plugin>

      <!-- Plugin for source plugin generation -->
      <plugin>
        <groupId>org.eclipse.tycho</groupId>
        <artifactId>tycho-source-plugin</artifactId>
        <version>${tycho-version}</version>
        <executions>
          <execution>
            <id>plugin-source</id>
            <goals>
              <goal>plugin-source</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

      <!-- Plugin for running unit tests -->
      <plugin>
        <groupId>org.eclipse.tycho</groupId>
        <artifactId>tycho-surefire-plugin</artifactId>
        <version>${tycho-version}</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Die Maven-Tycho-Dokumentation ist unter [3] zu finden. Unter Dokumentation sind Beschreibungen zu den einzelnen Packaging Types, eine Reference Card und die Maven-Plug-in-Dokumentation zu finden.

Target Platform

Die Target Platform ist ein Set von Artefakten, aus dem Tycho die Abhängigkeiten für den Build auflöst. Grundsätzlich gibt es zwei Wege, den Inhalt der Target Platform zu definieren:

- über die Angabe von Repositories mit *layout=p2* in der POM, wodurch das gesamte p2 Repository der Target Platform hinzugefügt wird (Listing 1)
- über die Definition einer eigenen Target-Definition, wodurch eine genauere Kontrolle der Abhängigkeiten erreicht wird

Möchte man sich keine Gedanken über Versionen und Abhängigkeiten machen, sollte die Repository-Variante gewählt werden. Tycho kann sich dadurch für die Auflösung sowohl von *included*- als auch *required*-Abhängigkeiten im gesamten Repository bedienen. Aufgrund der vielschichtigen Abhängigkeiten der Eclipse Bundles ist diese Variante zu bevorzugen. Damit die Features und Plug-ins, aus denen die eigene Eclipse-Distribution aufgebaut werden soll, später auch in der für den Build verwendeten Eclipse-Instanz (Running Platform) selbst aufgelöst werden können, sollten die notwendigen Bundles dort installiert sein. Das ist zwar nicht zwingend notwendig, aber so ist die initiale Zusammenstellung der eigenen Eclipse-Distribution einfacher.

Möchte man mehr Kontrolle darüber haben, welche Bundles in welcher Version zur Verwendung kommen, und sollen diese auch unabhängig von der aktuellen Eclipse-Installation ausgewählt werden können, sollte die Target-Definition-Variante gewählt werden. Hierfür kann, wie in Ausgabe 4.11 beschrieben, ein *target-platform*-Projekt mit einer *.target*-Datei angelegt werden. Außerdem muss das *target-platform-configuration*-Plug-in in der parent POM umgestellt werden, sodass es anstatt dem Repository die Target-Definition zur Auflösung der Target Platform verwendet. Seit der Tycho-Version 0.13, bei der das Target-Platform-Konzept überarbeitet wurde, sind folgende Punkte zu beachten:

- Tycho akzeptiert nur Software Sites als Location (z. B. die Eclipse-Juno-Updatesite); darin müssen die gewünschten Pakete wie das EPP Bundle Eclipse IDE for

Listing 2

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
      w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
                        POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>com.beone.java.tycho.example</artifactId>
  <packaging>eclipse-plugin</packaging>
  <parent>
    <groupId>com.beone.java</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../parent</relativePath>
  </parent>
</project>
```

Listing 3

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
      w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
                        POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>com.beone.java.tycho.example.test</artifactId>
  <packaging>eclipse-test-plugin</packaging>
  <parent>
    <groupId>com.beone.java</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../parent</relativePath>
  </parent>
</project>
```

Listing 4

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
      w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
                        POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>com.beone.java.tycho.feature</artifactId>
  <packaging>eclipse-feature</packaging>
  <parent>
    <groupId>com.beone.java</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>
</project>
```

Hilfreiche Plug-ins

Um die Maven Builds nicht nur über die Konsole, sondern aus Eclipse selbst heraus starten zu können, wird ein Maven-Plug-in wie m2e (Maven Integration for Eclipse) benötigt. Dieses kann über die Eclipse Juno Update Site http://download.eclipse.org/releases/juno installiert werden. Weitere Informationen zu m2e sind unter [2] zu finden. Damit die Maven Tycho Packaging Types von m2e aufgelöst werden können, muss außerdem der m2e Connector für Tycho installiert werden. Am einfachsten ist dieser über Window | Preferences | Maven | Discovery | Open Catalog zu finden. In dem darüber geöffneten Dialog den Tycho Configurator auswählen und installieren.


GAMES
DevCon®
29. – 30. Oktober 2012
Rheingoldhalle Mainz
Game Development, Design & Business
CRAFTING THE WORLD OF GAMES
BarCamp + Konferenz
99 €
www.gamesdevcon.de
S&S Media Group

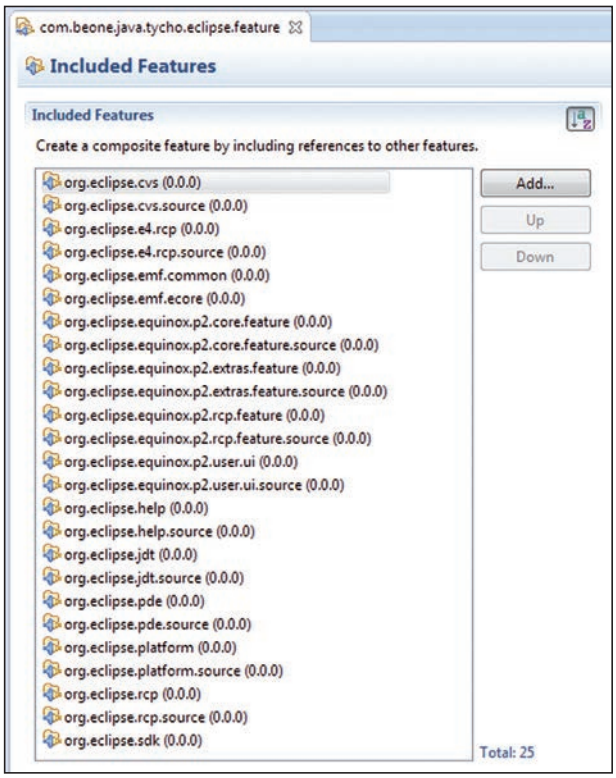

Abb. 2: Verwendete Features auf Basis der Eclipse Java IDE

Java Developers ausgewählt werden, die der eigenen Eclipse-Distribution als Basis dienen sollen.

- Es werden nur die aufgelösten Bundles der Target-Definition hinzugefügt (dies kann über den Reiter Content des Target-Definition-Editoris in Eclipse geprüft werden. Sollten in den später ausgewählten Features für die eigene Eclipse-Distribution Abhängigkeiten vorhanden sein, die nicht über die Pakete in der Target-Definition aufgelöst werden können, wird es während des Builds oder spätestens im fertigen Produkt zu Problemen führen.

Sollen zusätzlich zur Target Platform weitere Abhängigkeiten eingebunden werden, die nicht über ein p2 Repository verfügbar sind (z. B. *GlazedLists*), so können diese Maven-typisch über die *dependencies*-Section dem Build (z. B. in der parent POM) hinzugefügt werden. Das *target-platform-configuration*-Plug-in muss dann, wie in Listing 1, mit dem Konfigurationsparameter *pomDependencies=consider* versehen werden, damit die so konfigurierten OSGi Bundles zur Build-Laufzeit der Target Platform hinzugefügt werden. Die Auflösung dieser Bundles in Eclipse ist je nach Variante unterschiedlich. Für die Repository-Variante reicht es aus, das Bundle in der Running Platform verfügbar zu machen, zum Beispiel durch Ablegen in den *dropins*-Ordner der Eclipse-Installation. Bei der Target-Definition-Variante empfiehlt es sich, eine Directory Location mit den Bundles hinzuzufügen. Da Tycho Directory Locations nicht unterstützt, wird beim Build eine entsprechende Warnung angezeigt, die allerdings ignoriert werden kann, falls die Abhängigkeiten für den Build wie beschrieben eingebunden wurden. Weitere Informationen zu Target Platforms in Tycho unter [4].

Plug-in

Die Erstellung einer eigenen Eclipse-Distribution ist in der Regel mit der Einbindung eigener Plug-ins verbunden. Aufgrund des Manifest-first-Ansatzes ist der Build von Eclipse-Plug-ins mit Maven Tycho sehr einfach. Das Plug-in-Projekt muss nur um eine *pom.xml* erweitert werden, die das zuvor erstellte Parent-Projekt referenziert und den Packaging Type auf *eclipse-plugin* setzt (Listing 2). Bestehende Plug-in-Projekte können mithilfe von m2e einfach in Maven-Projekte umgewandelt werden. Über einen Rechtsklick auf das Projekt Configure | Convert to Maven Project wird ein einfacher Wizard für die Umwandlung geöffnet. Da dieser Wizard

Listing 5

```
<extension
      id="product"
      point="org.eclipse.core.runtime.products">
   <product
         application="org.eclipse.ui.ide.workbench"
         name="BeOne Tycho Example IDE">
      <property
            name="aboutText"
            value="This is the text shown in the product about dialog.">
      </property>
      <property
            name="aboutImage"
            value="beone_about.png">
      </property>
      <property
            name="appName"
            value="BeOne Tycho Example IDE">
      </property>
      <property
            name="preferenceCustomization"
            value="plugin_customization.ini">
      </property>
      <property
            name="windowImages"
            value="beone_16.png,beone_32.png">
      </property>
      <property
            name="cssTheme"
            value="org.eclipse.e4.ui.css.theme.e4_default">
      </property>
   </product>
```

weder die Tycho Packaging Types kennt, noch eine direkte *parent*-Konfiguration erlaubt, sollte er einfach mit Finish bestätigt und die notwendigen Modifikationen sollten anschließend in der erzeugten *pom.xml* vorgenommen werden. Dabei ist unbedingt darauf zu achten, dass der relative Pfad zum *parent*-Projekt gesetzt ist. Neben der Erzeugung der POM wird außerdem die Verzeichnisstruktur verändert, so ist das *bin*-Verzeichnis nicht mehr relevant und kann gelöscht werden, da Maven standardmäßig in das *target*-Verzeichnis schreibt. Sollten Projektfehler nach der Anpassung der POM auftreten, können diese in der Regel über Maven | Update Project ... gelöst werden.

Testing

Wie von Maven gewohnt, lassen sich auch in Tycho die Tests automatisch bei jedem Build ausführen. Hierzu wurde in Listing 1 das *tycho-surefire-plugin* in der parent POM hinzugefügt. Da Plug-in-Tests in der Regel als Fragmente erstellt werden, unterscheidet sich die Einbindung in den Build-Prozess kaum von der Einbindung von Plug-ins. Lediglich der Packaging Type muss *eclipse-test-plug-in* lauten (Listing 3). Außerdem muss sichergestellt sein, dass alle Abhängigkeiten über die *plugin.xml* aufgelöst werden können. In der Praxis hat es sich bewährt, *org.junit* als erforderliches Plug-in einzutragen, anstatt spezifische Packages zu importieren. Bei Letzterem kann es unter Umständen zu Problemen beim Auflösen von abhängigen Packages der Bibliothek selbst kommen.

Features

Wie eingangs beschrieben, wird die eigene Eclipse-Distribution auf Basis existierender Pakete aufgebaut. Um diese Basis zu definieren, müssen die Pakete, aus denen die Distribution bestehen soll, konfiguriert werden. Hierzu erzeugt man sich ein Feature und inkludiert die Eclipse-Features, die als Basis dienen sollen (**Abb. 2**). Der Einfachheit halber kann bei der Definition dieses Features eine Eclipse-Installation verwendet werden, die der Zieldistribution entspricht, um die Auswahl der notwendigen Features zu vereinfachen. Alternativ können die Pakete, soweit bekannt, auch manuell in die *feature.xml* eingetragen werden. Zusätzlich zu den enthaltenen Features müssen die folgenden Plug-ins explizit im Plug-ins-Bereich aufgelistet werden, um spätere Abhängigkeiten und Konfigurationen zu ermöglichen:

- *javax.el*
- *org.eclipse.core.runtime*
- *org.eclipse.equinox.ds*
- *org.eclipse.equinox.event*

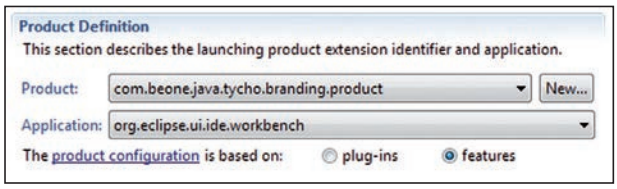

Abb. 3: Featurebasierende Produktdefinition

Die eigenen Plug-ins werden in einem separaten Feature zusammengefasst. Natürlich ist es auch möglich, sie dem oben erstellten Feature hinzuzufügen. Aufgrund des Feature Brandings und im Sinne der Kapselung ist eine saubere Trennung allerdings sinnvoller.

Die beiden Features müssen nach der Erstellung über Eclipse (New | Feature Project) und der beschriebenen Konfiguration der enthaltenen und abhängigen Plug-ins/Features ebenfalls in Maven-Projekte umgewandelt werden. Dabei ist wieder die Konfiguration des *parent* zu beachten, der Packaging Type ist diesmal *eclipse-feature* (Listing 4).

Branding

Das Branding der eigenen Eclipse-Distribution ist meistens ein zentraler Punkt. Hier können die Standardmechanismen des Eclipse-Produkt-Brandings, wie unter [5] beschrieben, verwendet werden. Dies kann sowohl in einem bestehenden als auch in einem neuen, nur für das Branding zuständigen Plug-in passieren. Wichtig ist dabei, dass in der *plugin.xml* der Extension Point *org.eclipse.core.runtime.products* eingebunden und mit einer eindeutigen ID versehen wird, über die das darunter definierte Produkt später referenziert werden kann (Listing 5). Als *application* wird für das Produkt *org.eclipse.ui.ide.workbench* gesetzt. Für Juno ist es erforderlich, die Property *cssTheme* für das Styling zu setzen, da ansonsten die Themes nicht geladen werden.

Das Branding der eigenen Eclipse-Distribution ist meistens ein zentraler Punkt.

Neben dem Produkt-Branding ist es auch möglich, das zuvor erstellte Feature der eigenen Plug-ins mit einem Branding zu versehen, sodass es im About-Dialog aufgeführt wird. Das Feature Branding kann erstellt werden, wie unter [6] beschrieben, wobei darauf zu achten ist, dass die notwendigen Dateien *about.ini, about.mappings* und die 32x32-Grafik für den Dialog in den *build.properties* des Plug-in-Projekts, das das Branding übernimmt, mit aufgenommen sind.

Produkt

Um am Ende ein lauffähiges Produkt zu bekommen, wird noch ein weiteres einfaches Projekt benötigt. Hierfür wird mit New | Project ein neues Projekt angelegt und darin mit New | Product Configuration die auf Features basierende Produktkonfiguration auf Basis des zuvor erstellten Produkts erstellt (**Abb. 3**). Dieser Kon-

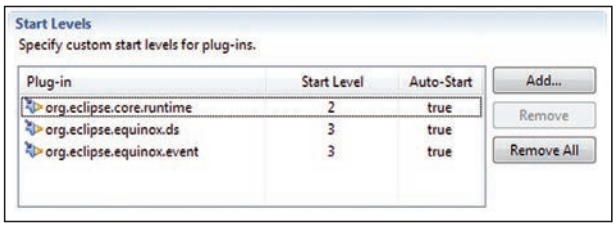

Abb. 4: Start-Level-Konfiguration zentraler Bundles

figuration fügt man die beiden zuvor erzeugten Features hinzu.

Um sicherzustellen, dass die Eclipse-Distribution gestartet werden kann, müssen in der Konfiguration der Produktdefinition, die Plug-ins *org.eclipse.core.runtime*, *org.eclipse.equinox.ds* und *org.eclipse.equinox.event* auf Auto-Start gesetzt werden, wie in [7] beschrieben (**Abb. 4**). Damit wird sichergestellt, dass diese zentralen Bundles vor allen anderen geladen werden.

Bei der Umwandlung dieses Projekts in ein Maven-Projekt ist neben der Konfiguration des *parent* und der Anpassung des Packaging Types auf *eclipse-repository* noch die Konfiguration für die Erstellung des *p2* Repositories und die Erzeugung des Archiv-Files zu beachten (Listing 6).

Um den gesamten Build über eine zentrale Stelle zu starten, müssen die erzeugten Maven-Projekte noch in der parent POM als Module eingetragen werden. Anschließend kann über Run as | Maven build ... aus Eclipse heraus oder über die Konsole aus dem *parent*-Projekt heraus der Maven Build gestartet und nach erfolgreichem Build die eigene Eclipse-Distribution im *target*-Verzeichnis des Produktprojekts begutachtet werden.

Dirk Fauth ist Senior Consultant bei der BeOne Stuttgart GmbH und seit mehreren Jahren im Bereich der Java-Entwicklung tätig. Er war in Projekten im Umfeld von JSF, Spring und Eclipse RCP tätig und ist aktiver Committer im Eclipse-Nebula-NatTable-Projekt.

Links & Literatur

[1] http://it-republik.de/jaxenter/artikel/Bauen-mit-vereinten-Kraeften-4780.html

[2] http://www.eclipse.org/m2e/

[3] http://www.eclipse.org/tycho/

[4] http://wiki.eclipse.org/Tycho/Target_Platform

[5] http://www.vogella.com/articles/EclipseRCP/article.html#product

[6] http://wiki.fernuni-hagen.de/eclipse/index.php/Branding_eines_Features

[7] http://wiki.eclipse.org/Eclipse4/RCP/FAQ#Why_won.27t_my_application_start.3F

Listing 6

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
      www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
        maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
                                                          4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>com.beone.java.tycho.distribution</artifactId>
  <packaging>eclipse-repository</packaging>
  <parent>
    <groupId>com.beone.java</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../parent</relativePath>
  </parent>

  <build>
    <plugins>
    <!-- Create a p2 repository -->
      <plugin>
        <groupId>org.eclipse.tycho</groupId>
        <artifactId>tycho-p2-repository-plugin</artifactId>
        <version>${tycho-version}</version>
        <configuration>
          <includeAllDependencies>true</includeAllDependencies>
        </configuration>
      </plugin>

      <!-- Create product ZIP files -->
      <plugin>
        <groupId>org.eclipse.tycho</groupId>
        <artifactId>tycho-p2-director-plugin</artifactId>
        <version>${tycho-version}</version>
        <executions>
          <execution>
            <id>materialize-products</id>
            <goals>
              <goal>materialize-products</goal>
            </goals>
          </execution>
          <execution>
            <id>archive-products</id>
            <goals>
              <goal>archive-products</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```