



Eclipse-RCP-Anwendungen mit Maven und Tycho bauen

Headless, aber trotzdem mit Verstand

Zum Bau von Eclipse-Plug-ins und OSGi-Bundles mit Maven bietet Tycho [1] hilfreiche Maven-Plug-ins und Erweiterungen. Falls Maven mit Build-Server und zentralem Repository bereits für sonstige Projekte verwendet wird, könnten diese Infrastruktur und das dazugehörige Know-how auch für die Eclipse-RCP-Entwicklung genutzt werden. Dieser Artikel stellt als Beispiel die Eclipse-RCP-Anwendung „insurance“ vor, durch das der Build-Prozess mit Maven 3 und Tycho erläutert wird. Um die grafische Benutzerschnittstelle automatisiert zu testen, sind SWTBot-Tests [2] Teil des vorgestellten Maven-Build-Prozesses.

von Kai Spichale

Eine allgemeine Einführung in das Thema Headless Build von Eclipse-RCP-Anwendungen mit Maven und Tycho wurde im Eclipse Magazin 4.10 [3] beschrieben. In diesem Artikel wollen wir dieses Thema anhand einer Beispielanwendung vertiefen. Der vorgestellte Build-Prozess soll zudem bestimmten Anforderungen genügen. Da es sich um die Entwicklung einer Eclipse-RCP-Anwendung handelt, muss gewährleistet werden, dass die Anwendung innerhalb von Eclipse gebaut und gestartet werden kann. Die umfangreichen Funktionen von Eclipse bzw. der Plug-in Development Environment (PDE) [4] sollen also ohne Einschränkungen weiterhin nutzbar sein. Zudem muss ein Headless Build möglich sein, um die Anwendungen auf einem Build- oder

Continuous-Integration-Server automatisiert bauen und testen zu können. Zu guter Letzt muss sichergestellt werden, dass alle Entwickler die gleiche Target-Plattform verwenden. Die in diesem Artikel beschriebene Beispielanwendung insurance ist eine kleine Eclipse-RCP-Anwendung, deren Quellcode Sie auf der Heft-CD finden. Kern der Anwendung ist die Antragserfassung. Dazu werden Informationen zur versicherten Person und zur Versicherung über eine Eingabemaske erfasst. Diese knappe Beschreibung muss uns schon reichen, denn in diesem Artikel soll es vor allem um den Build-Prozess dieser Anwendung gehen.

Manifest-first-Ansatz

Allgemein kann beim Bau von OSGi-Bundles und Eclipse-Plug-ins mit Maven zwischen *POM-first*-Ansatz

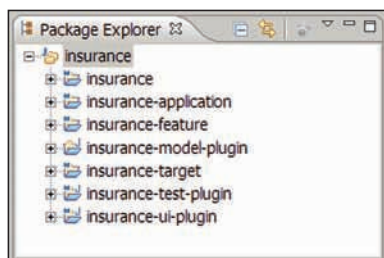


Abb. 1: Struktur der insurance-Anwendung

und *Manifest-first*-Ansatz unterschieden werden. Beim POM-first-Ansatz hilft das *maven-bundle-plugin* von Apache Felix [5], die Manifestdatei des OSGi-Bundles anhand der Definitionen in der *pom.xml* zu generieren. Beim Manifest-first-Ansatz ist es genau umgekehrt. In

beiden Fällen müssen die Projektabhängigkeiten und OSGi-Metadaten nicht redundant definiert werden. Tycho arbeitet nach dem Manifest-first-Ansatz und hat damit einen entscheidenden Vorteil: Die speziellen Eclipse-Projekttypen, die Eclipse zur Plug-in-Entwicklung bietet, können wie gewohnt verwendet werden, lediglich

eine *pom.xml* muss für jedes Projekt hinzugefügt werden. Denn Tycho unterstützt alle PDE-Projekttypen und kann die PDE/JDT-Projekt-Metadaten wiederverwenden. Das bedeutet, dass die Werkzeugunterstützung zur Pflege der Eclipse-Plug-in-Metadaten weiterhin genutzt werden kann. Grundvoraussetzung für diesen Ansatz ist m2eclipse [6], die Maven-Integration für Eclipse. Mit m2eclipse können Maven-Projekte auch innerhalb von Eclipse gebaut werden.

Projektstruktur im Überblick

Abbildung 1 zeigt ein Working Set mit allen Maven- bzw. Eclipse-Projekten der Beispielanwendung. Das Projekt *insurance* ist ein Multi-Module-Projekt und dient dem Bau der gesamten Anwendung. Es enthält grundlegende Plug-in-Konfigurationen, die es durch

Listing 1

```
<plugins>
  <plugin>
    <groupId>org.sonatype.tycho</groupId>
    <artifactId>tycho-maven-plugin</artifactId>
    <extensions>true</extensions>
  </plugin>
  <plugin>
    <groupId>org.sonatype.tycho</groupId>
    <artifactId>maven-osgi-compiler-plugin</artifactId>
    <configuration>
      <source>1.6</source>
      <target>1.6</target>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.sonatype.tycho</groupId>
    <artifactId>target-platform-configuration</artifactId>
    <configuration>
      <resolver>p2</resolver>
      <target>
        <artifact>
          <groupId>org.insurance</groupId>
          <artifactId>insurance-target</artifactId>
          <version>1.0.0-SNAPSHOT</version>
          <classifier>helios</classifier>
        </artifact>
      </target>
      <ignoreTychoRepositories>true</ignoreTychoRepositories>
    </configuration>
  </plugin>
</plugins>
```

Listing 2

```
<plugins>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
```

```
<artifactId>build-helper-maven-plugin</artifactId>
<executions>
  <execution>
    <id>attach-artifacts</id>
    <phase>package</phase>
    <goals>
      <goal>attach-artifact</goal>
    </goals>
    <configuration>
      <artifacts>
        <artifact>
          <file>insurance.target</file>
          <type>target</type>
          <classifier>helios</classifier>
        </artifact>
      </artifacts>
    </configuration>
  </execution>
</executions>
</plugin>
</plugins>
```

Listing 3

```
<plugins>
  <plugin>
    <groupId>org.sonatype.tycho</groupId>
    <artifactId>maven-osgi-test-plugin</artifactId>
    <configuration>
      <useUIHarness>true</useUIHarness>
      <product>org.insurance.plugin.product</product>
      <application>org.insurance.plugin.application</application>
    </configuration>
  </plugin>
  <dependency>
    <groupId>org.insurance</groupId>
    <artifactId>insurance-application</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
</dependencies>
</plugins>
```



Project Inheritance an die anderen Projekte vererbt, um den Konfigurationsaufwand zu verringern und Redundanzen zu vermeiden. Der eigentliche Quellcode der Anwendung befindet sich in den Projekten *insurance-ui-plugin* und *insurance-model-plugin*. Wie die Namen dieser beiden Projekte andeuten, handelt es sich hierbei um Eclipse-Plug-ins. Gründe für eine Aufteilung des Quellcodes kann es viele geben, doch an dieser Stelle soll uns nur die Definition der Abhängigkeit von *insurance-ui-plugin* auf *insurance-model-plugin* interessieren. Obwohl es sich um Maven-Projekte handelt, wird die Abhängigkeit nicht in *pom.xml* definiert, sondern in der OSGi-Manifestdatei von *insurance-ui-plugin*. Das Projekt *insurance-feature* enthält die Definition des Eclipse-Features, das die beiden Plug-ins beinhaltet. Die Product Configuration ist im Projekt *insurance-application* enthalten. Mithilfe dieses Projekts werden für verschiedene Plattformen ausführbare Programme erzeugt. Die SWTBot-Tests sind in *insurance-test-plugin* enthalten. Die Target Platform [4] wird in *insurance-target* definiert.

Maven-Konfiguration

Wie schon angedeutet wurde, befindet sich ein Großteil der notwendigen Konfiguration, der an die anderen Projekte vererbt wird im Projekt *insurance*. Listing 1 zeigt die Konfiguration folgender Maven-Plug-ins:

1. *tycho-maven-plugin*
2. *maven-osgi-compiler-plugin*
3. *target-platform-configuration*

Mit dem ersten Maven-Plug-in wird Tycho „aktiviert“, sodass die speziellen Packaging Types von Tycho verwendet werden können. Die verschiedenen Packaging Types von Tycho werden in [7] beschrieben. Während der Ausführung des Build-Prozesses berechnet Tycho entsprechend der OSGi-Regeln die Projektabhängigkeiten und fügt sie dem Maven-Project-Modell hinzu. Tycho verwendet hierfür die OSGi- und Eclipse-Metadaten. Zur Kompilierung des Quellcodes wird das zweite Maven-Plug-in verwendet. Es handelt sich dabei um einen speziellen JDT-Compiler, der die eingeschränkte Package-Sichtbarkeit von OSGi-Bundles berücksichtigt. Mit dem dritten Maven-Plug-in wird die Target Platform konfiguriert. Sie ist, vereinfacht ausgedrückt, die Eclipse-Installation, auf deren Basis Plug-ins entwickelt und ausgeführt werden. Der p2-Resolver benutzt die Plug-ins der Target Platform, um alle Abhängigkeiten, die nicht Teil des Projekts sind, aufzulösen. Falls nur ein einzelnes Plug-in und nicht das gesamte Projekt gebaut wird, kann der p2-Resolver auch die Plug-ins im lokalen Maven Repository verwenden.

Die Maven-Konfiguration von *insurance-ui-plugin* und *insurance-model-plugin* ist minimalistisch. Neben der *ArtifactId* und der Referenz auf das Parent-POM muss nur der Packaging Type *eclipse-plugin* angegeben werden. Gleiches gilt für *insurance-feature* und *insu-*

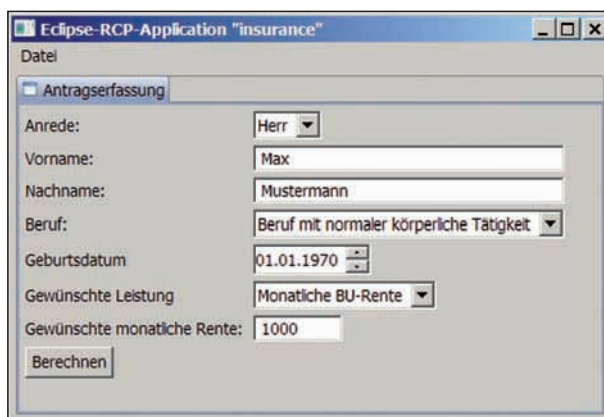


Abb. 2:
Die Oberfläche wird automatisch durch die Tests bedient

rance-application, für die als Packaging Type *eclipse-feature* bzw. *eclipse-application* dienen.

Die in Listing 2 gezeigte Konfiguration des Maven-Plug-ins *build-helper-maven-plugin* ist für *insurance-target* notwendig. Im Wesentlichen wird nur die Datei *helios.target* ausgewählt, in der die Target Platform definiert ist. Die Definition der Target Platform wird im folgenden Abschnitt noch genauer erläutert. Doch bevor es damit weitergeht, werfen wir noch einen Blick auf Listing 3. Darin ist die Konfiguration des Maven-Plug-ins *maven-osgi-test-plugin* gezeigt, das für die Oberflächentests in *insurance-test-plugin* notwendig ist. Die Konfiguration enthält eine Abhängigkeit auf *insurance-application* sowie die Product- und Application-ID der Anwendung.

Definition der Target Platform

Eine Target Platform ist eine Menge von Plug-ins, auf deren Basis die zu entwickelnde Anwendung gebaut und ausgeführt wird. Die zur Entwicklung verwendete Eclipse-Umgebung (IDE) dient standardmäßig als Target Platform. Damit kann aber nur schwer sichergestellt werden, dass verschiedene Entwickler die gleichen Plug-in-Versionen verwenden. Änderungen an der IDE durch Updates oder Neuinstallationen könnten Einfluss auf die zu entwickelnde Anwendung haben. Daher ist es ratsam, zwischen IDE und Target Platform zu unterscheiden. Eine separate Target Platform kann im einfachsten Fall durch Kopieren der IDE oder durch Download und Entpacken des RCP SDK erzeugt werden. Empfehlenswert ist jedoch die Verwendung einer Target Definition, mit der die Target Platform während des Builds erzeugt wird.

Die Target Definition *insurance.target* der Beispielanwendung wurde mit dem Target-Definition-Editor der PDE [4] erzeugt. Zur Auswahl der notwendigen Plug-ins können u. a. Verzeichnisse, Installationen und Software Sites angegeben werden. Für die Beispielanwendung wurden die Software Sites von Eclipse 3.6 und SWTBot angegeben, sodass die notwendigen Plug-ins automatisch durch den Eclipse-Provisierungsmechanismus p2 heruntergeladen werden. Mit der Option *Include all environments* muss man sich nicht mehr um Deltapacks kümmern, wenn die RCP-Anwendung für verschiedene



Plattformen gebaut werden soll. Sofern die Target Definition unter Versionskontrolle steht, ist sichergestellt, dass alle Entwickler die gleiche Target Plattform verwenden.

Bei diesem Ansatz wird man leider abhängig von externen p2-Updatesites, die eventuell nicht immer erreichbar sind und die notwendigen Plug-ins vielleicht nicht für alle Zeit bereithalten werden. Zwar müssen die Plug-ins und Features nur einmal ins lokale Maven Repository heruntergeladen werden, aber bei jedem Build werden die p2-Repository-Metadaten heruntergeladen, sodass der Build-Prozess länger dauert als notwendig. Aus diesen Gründen empfiehlt es sich, eine lokale p2-Updatesite zu verwenden, die die Target Plattform beinhaltet. Beispielsweise unterstützt Nexus Professional Eclipse-p2-Repositories.

Automatisiertes UI-Testen

Der beschriebene Build-Prozess wird erst durch automatisierte Tests komplett. Zur Qualitätssicherung der Beispielanwendung wollen wir SWTBot-Tests einsetzen. SWTBot wird uns dabei als „Klick-Roboter“ [8] dienen, d.h. die Oberfläche wird durch Aufrufe des SWTBot-API bedient. Wie dieses API eingesetzt werden kann, um robuste Oberflächentests zu entwickeln, wurde bereits im Eclipse Magazin [8], [9] beschrieben. Die Probleme [9], die es mit JUnit4 und Eclipse 3.5 gab, sind mit Eclipse 3.6 behoben worden. Daher werden wir JUnit 4 verwenden und darin das SWTBot-API aufrufen. Die JUnit-Tests erhalten folgende Annotation:

```
@RunWith(SWTBotJUnit4ClassRunner.class)
```

Wie wir in Listing 3 bereits gezeigt haben, kann die Ausführung der Tests mit geringem Aufwand automatisiert werden. Jetzt wird auch verständlich, warum der Parameter *useUIHarness* auf *true* gesetzt wurde, denn während der Tests wird die Anwendung gestartet und die Oberfläche bedient (Abb. 2).

Weitere Projektabhängigkeiten

Einer der Vorteile von Maven ist die Leichtigkeit, mit der Projektabhängigkeiten hinzugefügt werden können. Maven löst diese Abhängigkeiten selbstständig mit einem passenden Artefakt aus einem entfernten oder lokalen Repository auf. Mit Tycho funktioniert das leider nicht ganz so einfach, denn alle Projektabhängigkeiten können nur mit einem Bundle aus dem aktuellen Projekt oder aus der Target Plattform aufgelöst werden. Für solche Fälle bietet Tycho den Parameter *pomDependencies=consider*. Dieses fortgeschrittene Tycho-Feature sollte jedoch laut Dokumentation nur als Notlösung verwendet werden. Bei diesem Feature wird die Diskrepanz zwischen der Auflösung der transitiven Abhängigkeiten von Maven und OSGi deutlich, denn alle transitiven Abhängigkeiten müssten explizit angegeben werden. Auch wenn es gelingt, dieses heikle Feature erfolgreich einzusetzen, müssen die benötigten

Bibliotheken in OSGi-Bundles bzw. Plug-ins konvertiert werden. Leider ist es nicht möglich, den „POM-first“- und „Manifest-first“-Ansatz im gleichen Reactor Build zu verwenden, denn Abhängigkeiten werden sehr früh während des Maven-Build-Prozesses aufgelöst, sodass Manifestdateien von OSGi-Bundles nicht für „POM-first“-Projekte generiert werden können.

An dieser Stelle dürfen wir Orbit [10] und das SpringSource Bundle Repository [11] nicht vergessen. Dies sind Repositories, die OSGi-Versionen von OpenSource-Bibliotheken beinhalten, die zur Target-Plattform hinzugefügt werden können.

Fazit

Mit Tycho macht Sonatype einen großen Schritt vorwärts im Bestreben, Eclipse und Maven zu einem schlüssigen Gesamtkonzept zu verbinden. Mit dem „Manifest-first“-Ansatz von Tycho sind die zahlreichen PDE/JDT-Features, wie beispielsweise die editorgestützte Erstellung der Target-Definition-Dateien, trotz Maven nutzbar. Die Eclipse-Plug-in-Entwicklung mithilfe der PDE wird nicht eingeschränkt, denn Tycho unterstützt alle PDE-Projekttypen. Die Eclipse-RCP-Entwicklung wird also nicht schlechter, aber wird sie besser? Ja, denn mit Maven kann Headless gebaut und getestet werden. Und läuft der Maven-Build-Prozess erst einmal, dann fällt der Einsatz eines Continuous-Integration-Servers auch nicht mehr schwer. Der vorgestellte Build-Prozess erfüllt damit die am Anfang des Artikels formulierten Anforderungen. Schwierig ist jedoch die Integration zusätzlicher Abhängigkeiten, denn diese müssen erst in ein OSGi-Bundle verwandelt werden und können auch dann nicht ohne Weiteres zur Target Plattform hinzugefügt werden. Daher bleibt zu hoffen, dass Tycho auch für solche wiederkehrenden Probleme praktikable Lösungen bereitstellen wird.



Kai Spichale arbeitet als Senior Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG und entwirft und entwickelt Unternehmensanwendungen mit allem, was die Java-Plattform zu bieten hat.

Links & Literatur

- [1] <http://tycho.sonatype.org/>
- [2] <http://www.eclipse.org/swtbot/>
- [3] Thoms, Karsten: „Schaffe, schaffe, Eclipse baue!“, in: Eclipse Magazin 4.10
- [4] <http://help.eclipse.org/helios/index.jsp?nav=/4>
- [5] <http://felix.apache.org/site/index.html>
- [6] <http://m2eclipse.sonatype.org/>
- [7] <http://tycho.sonatype.org/packagingtypes.html>
- [8] „Vertrauen ist gut, Kontrolle ist besser“ in: Eclipse Magazin 4.09
- [9] „Automatisierte SWTBot-Tests“ in: Eclipse Magazin 4.10
- [10] <http://www.eclipse.org/orbit/>
- [11] <http://ebr.springsource.com>