Plus CD!

Stimmen zu EclipseCon, JAX und Eclipse Forum Europe >> 10

4.09

Deutschland € 9,80
Österreich € 10,80, Schweiz CHF 19,20 S&S

eclipse MAGAZIN

www.eclipse-magazin.de

CD-INHALT

Exklusiver Buchauszug:
„Eclipse Web Tools Platform" von Kai Brüssau, Kapitel 3: Java Servlets entwickler.press

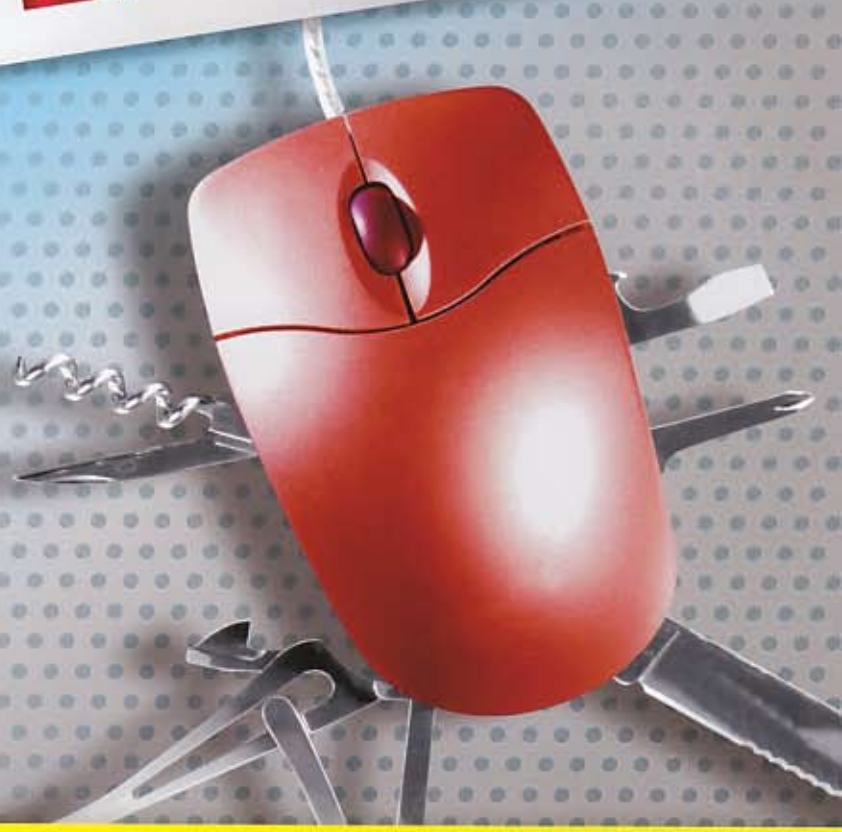
Chord Scale Generator
Eclipse-Tool für Saiteninstrumente

Eclipse-Tools
Java Workflow Tooling (JWT) 0.5.0, Bonita 4.1, FraSCAti 0.5

eDine
Innovatives Restaurant-Management

WTP
Dali JPA Tools, Ajax Toolkit Framework

WEB TOOLS PLATFORM



D 68864

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Unit Testing von RCP-Anwendungen mit dem Eclipse Testing Framework

RCP-Tests mit Fragmenten

Quellcode auf CD!

>> MARTIN DILGER

Der Build von RCP-Anwendungen ist ein nicht unbedingt triviales Thema. Wenn dann erschwerend hinzukommt, dass das Ganze bitte auch automatisiert, beispielsweise über ein Continuous Integration System wie CruiseControl, zudem noch „headless" und mit gleichzeitiger Durchführung der Unit-Tests geschehen soll, wird es allerhöchste Zeit, dieses Thema in einem eigenen Artikel zu beleuchten.

Das Schreiben von Unit-Tests ist ein elementarer Bestandteil des Entwicklungsprozesses von Software. Für RCP-Applikationen müssen jedoch, im Gegensatz zu Standard-Java-Applikationen, aufgrund der modularen Architektur der OSGi-Plattform einige Besonderheiten beachtet werden. Eine typische RCP-Anwendung besteht aus mehreren Plug-ins, die gemeinsam die Gesamtfunktionalität realisieren. Ein Plug-in (oder auch Bundle) entspricht hierbei dem theoretischen Konzept eines Moduls [1] und bietet üblicherweise ein öffentliches API, kapselt aber interne Implementierungsdetails. Hierfür bietet OSGi den Export-Package-Header, der es einem Plug-in erlaubt, in seinem Manifest nur diejenigen Packages zu veröffentlichen, die zum öffentlichen API des Plug-ins gehören. Dies ist ein sehr mächtiges Konzept, stellt aber bereits die erste Hürde für die Entwicklung von Unit-Tests dar. Die Schwierigkeit besteht darin, dass nicht nur die öffentlichen Elemente, sondern eben gerade die internen Implementierungsdetails ein ausführliches Testen erfordern.

Entwickeln von Unit-Tests für Plug-ins

Es gibt prinzipiell drei Ansätze, Unit-Tests für ein Eclipse Plug-in zu entwickeln. Der auf den ersten Blick einfachste Weg besteht darin, Tests direkt in das betreffende Plug-in zu integrieren (Abb. 1). Hierdurch bekommt man ein „All-in-One"-Plug-in, das es erlaubt, auch die interne Implementierung zu testen, da sowohl die zu testenden Klassen des Plug-ins als auch die Testklassen über denselben Classloader geladen werden (wovon bekanntlich jedes Plug-in einen eigenen hat).

Der Nachteil dieses monolithischen Ansatzes ist, dass etwaige Abhängigkeiten zu den verwendeten Testframeworks wie JUnit direkt in die implementierenden Plug-ins aufgenommen werden müssen. Hierdurch werden Zuständigkeiten vermischt, die eigentlich getrennt gehalten werden müssten. Ein weiterer möglicher Ansatz besteht darin, die Tests in ein jeweils eigenes Plug-in auszugliedern (Abb. 2).

Der Vorteil hiervon ist eine klare Trennung zwischen der Implementierung des Plug-ins und den zugehörigen Tests. Die Nachteile jedoch sind gravierend, da die Tests keinen Zugriff mehr auf die interne Implementierung des zu testenden Plug-ins haben. Einen Workaround bietet hierfür die x-friends-Direktive für den Export-Package-Header des Bundle-Manifests, der es erlaubt, Packages nur für bestimmte Plug-ins, also beispielsweise das Test-Plug-in, zu exportieren. Der folgende Export-Package-Header würde also das interne Package *de.pentasys.rcptesting.model.internal* nur für das Plug-in mit der ID *de.pentasys.rcptesting.model.test* exportieren:

```
Export-Package: de.pentasys.rcptesting.model.internal;
    x-friends:="de.pentasys.rcptesting.model.test"
```

Das ist ein recht statischer Ansatz und hilft außerdem nicht unbedingt, das Manifest des Plug-ins übersichtlicher zu gestalten. Es geht jedoch zum Glück besser: Die dritte Möglichkeit besteht nämlich darin, die Tests für ein Plug-in in Fragmenten zu entwickeln (Abb. 3).

Dies ist im Prinzip der elegante Mittelweg zwischen den beiden eben vorgestellten Lösungen. Ein Fragment ist immer einem bestimmten Host Plug-in zugeordnet und erweitert dessen Class-

Abb. 1: Testen mit einem einzigen Plug-in

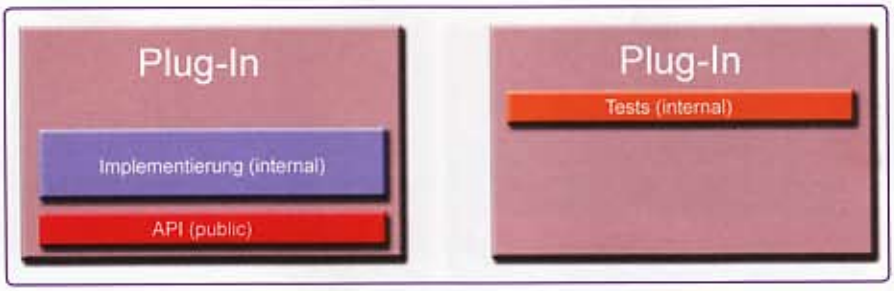


Abb. 2: Testen mit einem separaten Plug-in

path zur Laufzeit. Von der OSGi-Runtime wird sichergestellt, dass sowohl die Klassen des Plug-ins als auch die des Fragments vom gleichen Classloader geladen werden. Hierdurch erhält das Fragment Zugriff auf alle Klassen seines Host Plug-ins, auch ohne dass diese explizit exportiert wurden. Im Folgenden wird sich der Artikel mit diesem dritten Ansatz beschäftigen und anhand eines sehr einfachen Beispiels die Möglichkeiten zum Testen von Plug-ins mithilfe von Fragmenten erläutern. Weiterhin soll die Beispielapplikation über den PDE-Build im Headless-Modus, also ohne eine geöffnete Eclipse-Instanz, gebaut werden. Im Zuge dessen wird sie automatisiert getestet und die Testergebnisse werden anschließend gespeichert.

Listing 1

```
public interface IPersonService {

  public Person getPerson(String name);

  public String greetPerson(Person person,
    String flowerOfSpeech);

  public List<Person> getPersons();
}
```

Listing 2

```
public class PersonServiceTest extends TestCase {

  public void testGetPerson() {
   // never null
   assertNotNull(service.getPerson("Heinrich"));
  }

  public void testGreet() {
   assertEquals("Hello again Heinrich !!",
                        service.greetPerson(service
     .getPerson("Heinrich"), "Hello again"));
  }
  public void testFailure(){
   assertNull(service.getPerson("Heinrich"));
  }
}
```

Ein Beispiel

Um den Rahmen des Artikels nicht zu sprengen, wird ein sehr einfaches Beispiel verwendet. Im Prinzip handelt es sich um eine personalisierte Version der uns allen bekannten „Hello World"-Anwendung. Hierbei gibt es über eine Benutzeroberfläche die Möglichkeit, einer bestimmten Person einen personalisierten Gruß über die Konsole zukommen zu lassen. Der Aufbau der Anwendung ist in Abbildung 5 dargestellt.

Die Anwendung besteht aus drei Plug-ins (*de.pentasys.rcptesting.model*, *de.pentasys.rcptesting.ui* und *de.pentasys.rcptesting.app*) sowie einem Test-Plug-in (*de.pentasys.rcptesting.tests*), auf das später noch näher eingegangen wird (die Plug-ins sowie alle anderen Quellcodes zum Artikel befinden sich auch auf der beiliegenden Heft-CD). Jedem Plug-in der Anwendung ist ein eigenes Testfragment zugeordnet, das alle Tests zu dem betreffenden Plug-in enthält. Das ist ein guter Ausgangspunkt für eine RCP-Anwendung, die später noch beliebig durch weitere Plug-ins erweitert werden kann. Sowohl die Plug-ins der Anwendung als auch die Testfragmente werden in jeweils einem eigenen Feature (*de.pentasys.rcptesting.feature*, *de.pentasys.rcptesting.tests.feature*) zusammengefasst. Zusätzlich definiert das App-Plug-in eine featurebasierte *product*-Konfiguration, die später für den Build-Prozess benötigt wird. Das der Anwendung zugrunde liegende Domänenmodell besteht aus lediglich einer Klasse *Person* mit dem Attribut *name*. Für den Zugriff auf bestimmte Personen sowie das Senden einer Grußbotschaft bietet das Modell-Plug-in den Service *PersonService*, der über den Extension Point *org.eclipse.ui.services* mit dessen Schnittstelle *IPersonService* (Listing 1) registriert wird. Das Modell-Plug-in exportiert nur die Schnittstelle des Services.

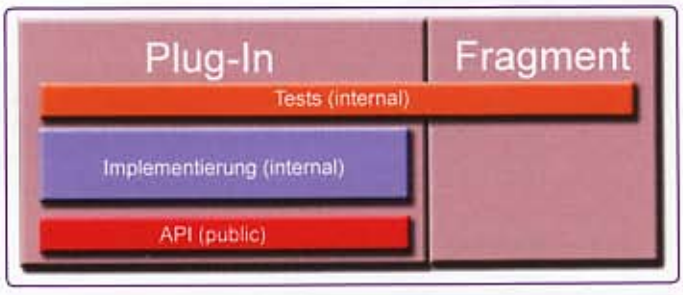


Abb. 3: Testen mit Fragmenten

Abb. 4: Der GEZ Viewer

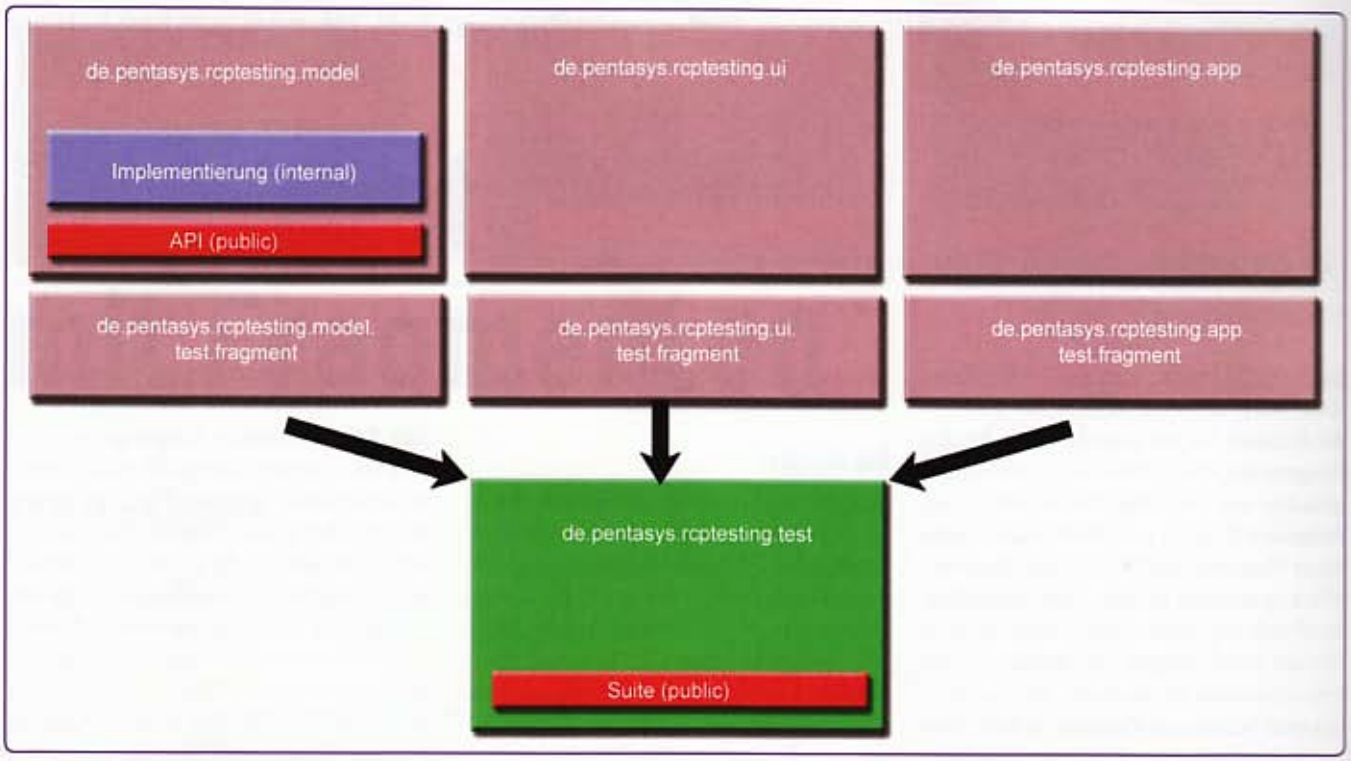


Abb. 5: Die Architektur

Die Implementierung selbst bleibt verborgen: *Export-Package: de.pentasys.rcptesting.model*. Würde also die Entwicklung von Tests nicht mithilfe von Fragmenten, sondern in einem eigenen Plug-in erfolgen, müsste entweder, wie bereits besprochen, das interne Package *de.pentasys.rcptesting.model.internal* über die Direktive:

```
de.pentasys.rcptesting.model.internal;
        x-friends:=de.pentasys.rcptesting.model.test
```

exportiert oder die zu testenden Klassen über den Java-Reflection-Mechanismus mit dem Bundle Classloader geladen werden. Ein einfacher Testfall für den Service ist in Listing 2 dargestellt. Hierbei wird zum einen getestet, ob der Service niemals *null* für eine Person liefert, und zum anderen, ob die zuvor erwähnte Grußfunktion korrekt arbeitet. Zusätzlich ist ein dritter Testfall vorhanden, der garantiert immer fehlschlägt.

Die Tests in einem Fragment können ebenso wie die Tests in einem Plug-in über den JUnit-Testrunner ausgeführt werden (Rechte Maustaste auf das Fragment *Run as – JUnit-Plug-in-Test*). Alle Tests in einem Fragment sollten in einer Fragment-TestSuite zusammengefasst werden:

```
public class AllModelTests {
 public static Test suite(){
  TestSuite testSuite = new TestSuite();
  testSuite.addTestSuite(PersonServiceTest.class);
  return testSuite;
 }
}
```

Diese einzelnen Fragment-Suites werden anschließend im globalen Test-Plug-in zu einer einzigen TestSuite zusammengefasst, die dann bei der Ausführung alle vorhandenen Tests der Applikation abarbeitet. Diese globale TestSuite wird später für den Headless-Build verwendet. Hierzu exportiert jedes Fragment seine TestSuite:

```
Export-Package: de.pentasys.rcptesting.model.tests
```

Listing 3

```
<property file="build.properties"/>
<target name="pdeBuild" depends="initBase,initBuild">
 <java classname="org.eclipse.equinox.launcher.Main"
                    fork="true" failonerror="true">
  <arg value="-application" />
  <arg value="org.eclipse.ant.core.antRunner" />
  <arg value="-buildfile" />
  <arg value="${baseLocation}/plugins/
        org.eclipse.pde.build_${pdeVersion}/scripts/
                productBuild/productBuild.xml" />
  <classpath>
   <pathelement location="${baseLocation}/plugins/
org.eclipse.equinox.launcher_${launcherVersion}.jar" />
  </classpath>
 </java>
</target>

<target name="initBase">
 <mkdir dir="${base}/../" />
 <get verbose="true" dest="${base}/../base.zip"
                    src="${eclipseBaseURL}" />
 <unzip dest="${base}" src="${base}/../base.zip" />
 <get dest="${base}/../delta.zip" src="${eclipseURL}
        /eclipse-${eclipseBuildId}-delta-pack.zip" />
 <unzip dest="${base}" src="${base}/../delta.zip" />
 <get dest="${base}/../etf.zip" src="${eclipseURL}
    /eclipse-test-framework-${eclipseBuildId}.zip" />
 <unzip dest="${base}" src="${base}/../etf.zip" />
</target>

<target name="initBuild">
 <copy todir="${buildDirectory}/plugins">
  <fileset dir="../">
   <include name="de.pentasys.*/**" />
   <exclude name="de.pentasys.*feature/**" />
   <exclude name="de.pentasys.*build/**" />
  </fileset>
 </copy>
 <copy todir="${buildDirectory}/features">
  <fileset dir="../">
   <include name="de.pentasys.*feature/**" />
  </fileset>
 </copy>
</target>
```

Diese Implementierung der globalen TestSuite entspricht prinzipiell genau der eben vorgestellten. Es werden lediglich die Fragment-TestSuites hinzugefügt: *suite.addTest(AllModelTests.suite());*. Hier trifft man auf das Problem, dass die in Fragmenten definierten Suite-Klassen zur Entwicklungszeit durch das PDE nicht aufgelöst werden können, da Fragmente per Default erst zur Laufzeit geladen werden. Um dieses Problem zu umgehen, wird in jedes Manifest eines Plug-ins, das über ein Testfragment verfügt, folgende Zeile eingefügt: *Eclipse-ExtensibleAPI: true*. Wird dieser Wert auf *true* gesetzt (der Default ist *false*), werden Fragmente bereits zur Entwicklungszeit aufgelöst, wodurch ein Laden der TestSuites aus Fragmenten möglich wird. Man beachte, dass sich dies nur zur Entwicklungszeit auswirkt. Das Verhalten der Anwendung zur Laufzeit ändert sich hierdurch nicht. Führt man jetzt das Test-Plug-in als JUnit-Plug-in-Test aus, sollten alle drei vorhandenen Testfälle abgearbeitet werden.

Headless-Build der Anwendung

Um die Anwendung später bauen zu können, ohne dies manuell über den Export-Wizard tun zu müssen, wird als Nächstes der Headless-Build-Prozess konfiguriert. Dies kann aufgrund des beschränkten Umfangs des Artikels leider nur in Ansätzen erläutert werden. Zunächst müssen hierfür folgende Dateien heruntergeladen werden:

- Eclipse 3.4 SDK [4]
- Eclipse Delta Pack [5]
- Eclipse Testing Framework (ETF) [6]

Das Eclipse-SDK sowie das Delta Pack werden benötigt, um die Anwendung *headless* (inklusive Launcher etc.) bauen zu können. Das Eclipse Testing Framework (ETF) wird benötigt, um die Unit-Tests in der Anwendung automatisiert auch im Headless-Modus ausführen zu können. Hierfür bietet das ETF einige Ant-Tasks, die das Testen einer Anwendung erleichtern (dazu jedoch später mehr). Für den Headless-Build-Prozess müssen alle Plug-ins der Anwendung in ein bestimmtes Verzeichnis (Build-Verzeichnis) kopiert werden. Dieses muss zwingend die in Abbildung 6 dargestellte Verzeichnisstruktur aufweisen.

Normalerweise würde sich das PDE bei entsprechender Konfiguration die benötigten Projekte selbst aus der Versionsverwaltung auschecken. Dies wurde bereits ausführlich in [7] beschrieben. Hier wollen wir stattdessen die Bereitstellung der benötigten Plug-ins in den Verzeichnissen durch ein Ant-Skript sicherstellen.

Die modulare Architektur der OSGi-Plattform bietet einige Besonderheiten

Der PDE Build ist in Form von Skripten im Plug-in *org.eclipse.pde.build* definiert. Dieses befindet sich im Plug-in-Verzeichnis einer lokalen Eclipse-Distribution. Die Templates, die für die Konfiguration des Build-Prozesses benötigt werden, befinden sich hier im Unterverzeichnis *templates/headless-build*.

Konfiguration des PDE Build

Am einfachsten definiert man im Workspace ein weiteres Projekt (*de.pentasys.rcptesting.app.build*), das die gesamte Build-Konfiguration beinhaltet. In dieses kopiert man zunächst die beiden Dateien *build.properties* und *customTargets.xml* aus dem PDE-Build Plug-in. Zusätzlich wird eine Datei *build.xml* erzeugt, die uns als Startpunkt des Build-Prozesses dient. Die *build.properties* dient der Konfiguration. Hier wird beispielsweise festgelegt, wo sich die Plug-ins der zu bauenden Anwendung oder die Target-Plattform befinden. Die *customTargets.xml* bietet nützliche Ant-Tasks als Callbacks an, die während des Build-Vorgangs vom PDE Build automatisch aufgerufen werden und ideal für Aufgaben wie das Durchführen von Tests oder Aufräumarbeiten nach einem erfolgreichen Build geeignet sind. Diese beiden Dateien müssen für den Build-Vorgang konfiguriert werden. Die Datei *build.properties* beinhaltet sehr viele Properties, von denen jedoch nur wenige angepasst werden müssen. Da ein *product-build* konfiguriert werden soll, wird zunächst die Property *product* mit

Abb. 6: Die Struktur des Build-Verzeichnisses

dem Pfad des Plug-ins sowie der zuvor definierten *product-id* definiert:

```
product=/de.pentasys.rcptesting.app/rcptesting.product
```

Als Nächstes muss der Pfad zum Build Directory über die Property *buildDirectory* konfiguriert werden. In diesem Directory müssen sich später die zu bauenden Plug-ins befinden: *buildDirectory=g:/workspaces/rcptesting/build*. Zusätzlich muss der Pfad zu einem lokalen Eclipse-SDK konfiguriert werden, das für den Build verwendet wird. Man sollte hier nicht seine Entwicklungsumgebung verwenden. Am besten wird die gesamte Build-/Testumgebung on-the-fly während des Build-Prozesses generiert. Die beiden Properties *base* und *baseLocation* geben hierbei an, wo sich die lokale Eclipse-Installation (Target-Plattform) für den Build befindet (in dieses Verzeichnis wird später einfach eine neue Eclipse-Distribution entpackt):

```
base=d:/workspaces/rcptesting
baseLocation=${base}/eclipse
```

Wo sich die zuvor heruntergeladenen Zip-Dateien befinden, aus denen eine lokale Eclipse-Distribution entpackt werden kann, wird mit den Properties *eclipseURL*, *eclipseBuildId* und *eclipseBaseURL* konfiguriert. Für das hier verwendete Beispiel müssen sich im Verzeichnis *g:/development/sdks* neben einer kompletten Eclipse-SDK das Eclipse Delta Pack und das ETF in gezippter Form befinden:

```
eclipseURL=file:g/development/sdks
eclipseBuildId=3.4
eclipseBaseURL=${eclipseURL}
                /eclipse-SDK-${eclipseBuildId}-win32.zip
```

Zuletzt muss noch konfiguriert werden, für welche Plattform das Produkt gebaut werden soll, in diesem Fall für die Windows-Plattform (alternative Konfigurationen, beispielsweise für Linux, findet man in den Kommentaren): *configs = win32,win32,x86*. Wird mit einer