Eigene Ant-Erweiterungen schreiben

Kleiner Helfer, ganz groß

Ant ist das Make für Java-Projekte, kann aber viel mehr: Wer Sonderwünsche hat, kann Ant beliebig erweitern und neue Tasks mit eigenen Java-Klassen verbinden. Wie das geht, zeigt dieser Coffee-Shop. Bernhard Bablok


www.visipix.com / Rolf Ryter

In der Einleitung der Ant-Dokumentation heißt es „Makefiles are inherently evil“ (Makefiles sind von Grund auf böse) [1]. Die unsichtbaren Tabs als bedeutungsvolle Zeichen zu benutzen führe nämlich zu schwer auffindbaren Fehlern. Der Autor dieses Beitrags ist hier anderer Meinung: Jedes strukturierte Dokument hat seine Konventionen, bei Python ist es die Einrückung, bei Makefiles sind es eben Tabs und Doppelpunkte und bei XML-Dateien die spitzen Klammern.

Der große Vorteil von Ant ist aber, dass es unabhängig vom Betriebssystem und recht leicht zu erweitern ist. Bei Makefiles gibt es genau einen Weg, um mehr Funktionalität hinzuzufügen: Man ruft ein Programm oder Shellskript auf. Auch Ant kann externe Programme ausführen, aber damit beginnt wieder die Abhängigkeit vom Betriebssystem. Besser ist es, eine eigene Task (in Ant definierter Arbeitsschritt) zu schreiben oder sich die Funktionalität aus vorhandenen Tasks zusammenzubasteln.

Dieser Coffee-Shop konzentriert sich auf die zweite Lösung. Er setzt grundlegende Kenntnisse von Ant voraus, siehe dazu den einführenden Artikel in einem früheren Heft [2]. Entwickelt wurde unter Ant 1.6.0, ältere Versionen unterstützen nicht alle Möglichkeiten. Der gesamte Code ist auf dem Listing-Server des Linux-Magazins verfügbar [3].

Versionsnummern verwalten

Die Beispiel-Task soll die Programmversion eines Java-Projekts pflegen. Listing 1 zeigt einen Ausschnitt aus dessen Buildfile. In Zeile 14 lädt es eine Eigenschaft (Property) aus der Datei »version.properties«, die sehr einfach aufgebaut ist:

```
app.version = 0.4.2
```

Die Version setzt sich aus Major-Nummer, Minor-Nummer und Patch-Level zusammen und wird in weiteren Tasks verwendet, etwa bei der Paketierung (hier landen die Dateien wie bei vielen Projekten üblich in einem Unterverzeichnis »*AppName-Version*«). Aufgabe der Task »VersionManager« ist es, die einzelnen Komponenten hochzuzählen. Dabei soll sie untergeordnete Komponenten auf »0« setzen, wenn sie eine übergeordnete Komponente erhöht. Listing 2 zeigt Beispiele für den Aufruf des »VersionManager«.

Die Task hat zwei Attribute: das optionale »file« mit dem Defaultwert »version.properties« sowie das Attribut »inc« mit den möglichen Werten »major« (Zeile 77), »minor« (Zeile 73) und »patchlevel« (Zeile 69).

Strukturiert durch Tasks

Jede Task wird durch eine Java-Klasse implementiert. Das Buildfile legt die jeweilige Klasse für eigene Tasks fest, siehe Zeilen 63 bis 65 in Listing 2. Die Klasse erweitert typischerweise »org.apache.tools.ant.Task«. Eventuell ist eine andere Basisklasse sinnvoller, zum Beispiel »org.apache.tools.ant.taskdefs.MatchingTask«, falls die Task Dateien über »exclude«- oder »include«-Patterns verarbeitet.

Für jedes Attribut ist eine Setter-Methode notwendig, im Beispiel also »setFile« und »setInc«, in Listing 3 die Zeilen 75 und 85. Der Typ dieser Methoden ist »public void«. Je nach Argumenttyp sorgt Ant für eine entsprechende Para-

Listing 1: »build.xml«

```
001 <?xml version="1.0" encoding="ISO-8859-1"?>
008
009 <project name="VersionManager" default=
    "compile" basedir=".">
010
011 <!-- ====== Property Definitions ====== -->
012
013 <property file="build.properties"/>
014 <property file="version.properties"/>
015 <property file="${user.home}/build.
    properties"/>
016
017 <property name="app.name" value="version-
    manager"/>
```

meterumwandlung, etwa nach Boolean oder im Beispiel zum Typ »java.io.File«. Die Methode »void execute()« ab Zeile 109 implementiert dann die eigentliche Arbeit der Task.

Klassen implementieren

Listing 3 zeigt den relevanten Code in Ausschnitten. Die Setter-Methoden sind eher trivial, nur »setInc()« verifiziert den eingegebenen Wert mit Hilfe der Methode »validateInc()«. Diese wirft bei Bedarf eine »BuildException« (Zeilen 96 bis 101), die Ant durchgereicht bekommt, das den Build-Vorgang daraufhin abbricht (Abbildung 1).

Das Beispiel demonstriert, wie einfach das Schreiben eigener Tasks ist. Neben der »BuildExeption« verwendet der Code nur eine weitere Ant-Funktionalität: Der selbst geschriebene »VersionManager« erbt von der Ant-Klasse »Task« die Methode »log()« und gibt damit Meldungen an den Benutzer aus (siehe Zeilen 113, 118, 122 und 126). Tasks, die geschachtelte Elemente verarbeiten, sind natürlich etwas komplizierter. Details hierzu finden sich in der Ant-Dokumentation. Dort gibt es auch ein Tutorial im Zip-Format, das sich als Ausgangspunkt für eigene Entwicklungen eignet.

Während Ant die Standardtasks schon kennt, definiert man zusätzliche Tasks mit »taskdef« – denn Ant muss wissen, welche Klasse die Task verarbeitet. Eine Taskdef-Task darf eine Toplevel-Task, also direkt unterhalb von »project« eingetragen sein. Sie kann aber auch in einem eigenen Target enthalten sein.

Das Buildfile des Beispiels verwendet letztere Strategie (siehe Listing 2, Zeilen 62 bis 66 und »depends = ...« in den Zeilen 68, 72 und 76). Sie bietet den Vorteil, dass andere Targets kompiliert werden können, auch wenn die Task-Klasse nicht verfügbar ist – bei einer Toplevel-Task würde Ant den gesamten Build abbrechen. Für die Entwicklung eigener Tasks ist dies sehr nützlich, da sich innerhalb desselben Buildfiles Targets sowohl für die Entwicklung als auch für den Test einbauen lassen.

Empfehlenswert ist dieses Vorgehen auch bei Projekten, die optionale Tasks von dritter Seite verwenden, ohne diese mitzuliefern. So kann ein Nutzer damit arbeiten, ohne vorher alle nötigen Tasks zu installieren.

Unit-Tests

Wer seine Tasks veröffentlichen will, sollte sie zuvor sorgfältig testen. Schließlich ist Ant ein Entwicklungswerkzeug und Entwickler als Zielgruppe sind sehr kritisch (außer ihrem eigenen Code gegenüber). Fürs Testen, speziell für Unit-Tests, bietet Ant von Haus aus gute Unterstützung.

Zuerst ist es erforderlich, den Ant-Quellcode und das JUnit-Paket [4] zu installieren, in das Wurzelverzeichnis des Quellcodes zu wechseln und die folgenden Befehle einzugeben: ►

Listing 2: Aufrufbeispiele »VersionManager«

```
062 <target name="deftask" description="Define
    task VersionManager" depends="jar">
063   <taskdef name="VersionManager"
064            classname="VersionManager"
065            classpath="${app.lib}"/>
066 </target>
067
068 <target name="inc-pl" description="Increment
    patch-level" depends="deftask">
069   <VersionManager file="${version.file}"
    inc="patchlevel"/>
070 </target>
071
072 <target name="inc-minor"
    description="Increment minor-level"
    depends="deftask">
073   <VersionManager inc="minor"/>
074 </target>
075
076 <target name="inc-major" description=
    "Increment major-level" depends="deftask">
077   <VersionManager inc="major"/>
078 </target>
079
080 <target name="inc-fail" description="Test
    invalid inc-type" depends="deftask">
081   <VersionManager inc="foo"/>
082 </target>
```

Listing 3: »VersionManager.java«

```
022 import java.io.*;
023 import java.util.*;
024 import org.apache.tools.ant.*;
025
040 public class VersionManager extends Task {
041
046   private File iVersionFile = new
    File("version.properties");
052   private String iInc;
059   private final String TYPE_MAJOR = "major",
060                        TYPE_MINOR = "minor",
061                        TYPE_PL    =
    "patchlevel";
067   private int iMajorVersion, iMinorVersion,
    iPatchLevel;
068
075   public void setFile(File versionFile) {
076     iVersionFile = versionFile;
077   }
078
085   public void setInc(String inc) {
086     iInc = inc;
087     validateInc();
088   }
089
096   private void validateInc() {
097     if (! iInc.equals(TYPE_MAJOR) &amp;&amp;
098         ! iInc.equals(TYPE_MINOR) &amp;&amp;
099         ! iInc.equals(TYPE_PL))
100       throw new BuildException("Invalid value
    of inc: " + iInc);
101   }
102
109   public void execute() {
110     try {
111       readFile();
112       if (iInc.equals(TYPE_MAJOR)) {
113         log("incrementing major-version");
114         iMajorVersion += 1;
115         iMinorVersion  = 0;
116         iPatchLevel    = 0;
117       } else if (iInc.equals(TYPE_MINOR)) {
118         log("incrementing minor-version");
119         iMinorVersion += 1;
120         iPatchLevel    = 0;
121       } else {
122         log("incrementing patch-level");
123         iPatchLevel += 1;
124       }
125       writeFile();
126       log("new version is: " +
127           iMajorVersion + "." + iMinorVersion
    + "." +
128           iPatchLevel);
129     } catch (IOException e) {
130       throw new BuildException("Failed with
    IOException: " + e.toString());
131     }
132   }
182 }
```

```
export CLASSPATH=Pfad-zu-junit.jar
ant test-jar
cp build/lib/ant-testutil.jar $ANT_HOME/lib
cp Pfad-zu-junit.jar $ANT_HOME/lib
```

Alternativ zur Installation unter »$ANT_HOME/lib« kann man die Dateien zusammen mit den außerdem nötigen »ant.jar« und »junit.jar« in den Classpath aufnehmen. Nach diesen vorbereitenden Schritten ist es leicht, den Test durchzuführen. Listing 4 zeigt dessen Quellcode, Listing 5 die zugehörigen Targets im Buildfile und Abbildung 2 den Report eines Testlaufs.

Tipps und Tricks

Wer nur vorhandene Buildfiles kopiert und an eigene Bedürfnisse anpasst, übersieht viele Möglichkeiten von Ant. Der erste Tipp ist, einen so genannten Logger zu verwenden. Abbildung 1 zeigt den »AnsiColorLogger«, der die Ausgabe durch Farben strukturiert. Es gibt auch Logger, die sich für automatisierte Builds eignen, etwa den »MailLogger«, der das Ergebnis per E-Mail verschickt.

Wie oben beschrieben gibt es mehrere Core-Tasks, mit denen sich gewünschte Funktionalität zusammenbauen lässt. Auch die Task »VersionManager« komplett selbst zu schreiben ist eigentlich überflüssig, wenn auch für Demozwecke durchaus nützlich. Listing 6 zeigt, wie eine »propertyfile«-Task dieselbe Funktionalität realisiert. Diese Task kann noch mehr, beispielsweise mit einem Kalenderdatum rechnen. Zusätzlich sei auf die »BuildNumber«-Task verwiesen: Sie ist der »VersionManager«-Task ähnlich, verwaltet jedoch nur eine einzelne Build-Nummer.

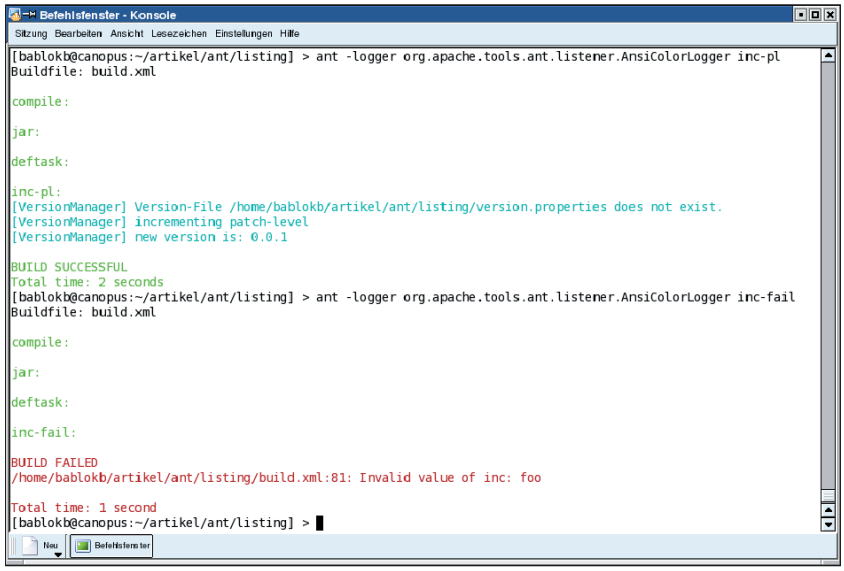

Abbildung 1: Die »VersionManager«-Task in Aktion. Für farbige Ausgabe sorgt das Modul »AnsiColorLogger«.

Ant verwendet wie Make implizit Bedingungen. Explizit lassen sie sich über Properties steuern. Dazu dienen die beiden Target-Parameter »if = property« und »unless = property«:

```
<target name="compile" ...>
<javac ... if="with.test">
...
</javac>
...
</target>
```

Listing 4: »VersionManagerTest.java«

```
22 import org.apache.tools.ant.*;
23
31 public class VersionManagerTest extends
   BuildFileTest {
32
37   public VersionManagerTest(String s) {
38     super(s);
39   }
40
47   public void setUp() {
48     configureProject("build.xml");
49   }
50
57   public void testPatchLevel() {
58     executeTarget("inc-pl");
59     assertLogContaining("does not exist.");
60     assertLogContaining("new version is: 0.0.1");
61   }
62
69   public void testMinor() {
70     executeTarget("inc-minor");
71     assertLogContaining("incrementing minor-
   version");
72     assertLogContaining("new version is: 0.1.0");
73   }
74
81   public void testMajor() {
82     executeTarget("inc-major");
83     assertLogContaining("incrementing major-
   version");
84     assertLogContaining("new version is: 1.0.0");
85   }
86
93   public void testFail() {
94     expectBuildException("inc-fail","");
95   }
96 }
```

Listing 5: JUnit-Tests

```
092 <target name="junit" description="Runs the unit
    tests" depends="jar">
093   <delete file="${version.file}" />
094   <delete dir="${junit.home}/xml" />
095   <mkdir  dir="${junit.home}/xml" />
096   <junit printsummary="yes" haltonfailure="no">
097     <classpath refid="app.classpath"/>
098     <formatter type="xml"/>
099     <batchtest fork="yes" todir="${junit.
    home}/xml">
100       <fileset dir="${src.home}"
    includes="**/*Test.java"/>
101     </batchtest>
102   </junit>
103 </target>
104
105 <target name="junitreport" description="Create a
    report for the test result">
106   <mkdir dir="${junit.home}/html" />
107   <junitreport todir="${junit.home}/html">
108     <fileset dir="${junit.home}/xml">
109       <include name="*.xml"/>
110     </fileset>
111     <report format="frames"
    todir="${junit.home}/html"/>
112   </junitreport>
113 </target>
114
115 <target name="test"
116       depends="junit,junitreport"
117       description="Runs unit tests and creates
    a report"/>
```

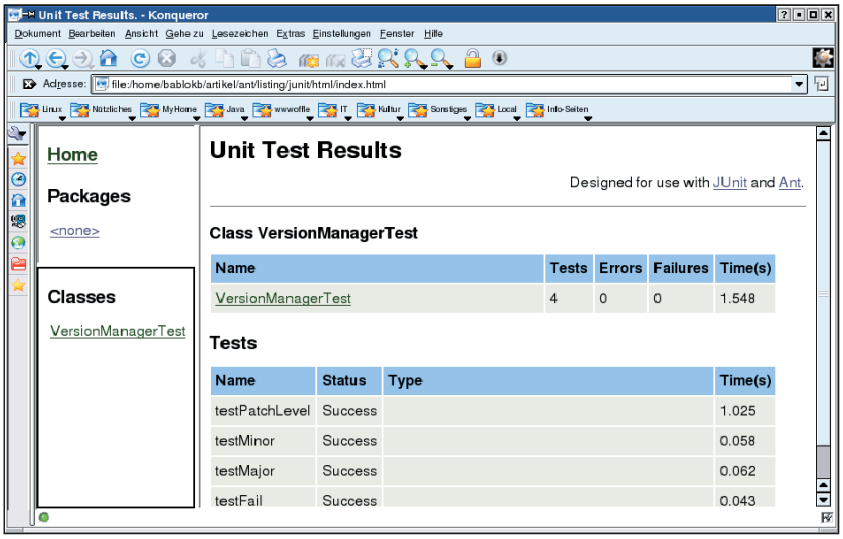

Abbildung 2: Die JUnit-Testergebnisse übersichtlich aufbereitet: Alle vier Testvarianten wurden erfolgreich durchgeführt, Fehler sind nicht aufgetreten.

In diesem Beispiel ist die erste Javac-Task nur ausführbar, wenn die Property mit dem Namen »with.test« gesetzt ist, unabhängig von deren konkretem Wert, zum Beispiel auf der Command Line:

```
ant -Dwith.test=foo meintarget
```

Spezielle Tasks wie »Available«, »Uptodate« oder »Condition« setzen bei definierten Bedingungen solche Variablen. Nutzbar ist dies bei Builds, um Funktionalität ein- oder auszuschalten. Das funktioniert analog zu den »--with-*xxx*«-Konstruktionen bei der Verwendung von Autoconf.

Die TStamp-Task setzt die Properties »DSTAMP«, »TSTAMP« und »TODAY«. Damit lassen sich nicht nur Bedingungen formulieren, sondern auch weitere Tasks mit diesen Properties versorgen, etwa um das Build-Datum in die Dokumentation aufzunehmen.

Kleine Schwächen

Für Unix-User zeigen sich die Grenzen von Ant in den Dateiberechtigungen. Da Java wegen seiner Plattformunabhängigkeit nicht mit Berechtigungen umgehen kann, erlebt man mit manchen Tasks böse Überraschungen. Die Copy-Task etwa erzeugt Dateien immer mit der aktuellen Umask. Und auch die Tar-Task erhält nicht automatisch die Attribute der einzelnen Dateien, ist aber durch etwas Handarbeit zu korrigieren.

Eine ganze Reihe weiterer Tasks wäre erwähnenswert. Es lohnt sich, jede Task im Handbuch zumindest zu überfliegen. Neben den Core-Tasks gibt es optionale Tasks zum Herunterladen. Zurzeit will das Ant-Team keine weiteren Tasks mehr in die Distribution aufnehmen. Links zu unabhängigen Entwicklern gibt es aber auf den Ant-Projektseiten.

finally{}

Mit Artikeln über Ant könnte man ein ganzes Linux-Magazin füllen. Die gezeigten Beispiele geben hoffentlich einen Anstoß dazu, sich intensiver mit dieser inzwischen etablierten Basistechnologie zu beschäftigen und Ant-Funktionalität in die eigenen Projekte zu integrieren. Auch der nächste Coffee-Shop hat ein wichtiges Entwicklungstool zum Thema: Javadoc kann mehr als nur Dokumentation erzeugen. Bis dahin viel Spaß mit Ant und Buildfiles. *(ofr)* ■

Infos

[1] Ant Homepage: [http://ant.apache.org/]
[2] S. Eschweiler, „ Ameisen am Werk": Linux-Magazin 05/01, S. 126 [http://www.linux-magazin.de/Artikel/ausgabe/2001/05/ants/ant.html]
[3] Listings dieses Coffee-Shops: [http://www.linux-magazin.de/Service/Listings/2004/08/Coffeeshop]
[4] JUnit: [http://www.junit.org]

Der Autor

Bernhard Bablok arbeitet bei der AGIS mbH als Anwendungsentwickler. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um die Objektorientierung. Er ist unter der Adresse [coffee-shop@bablokb.de] zu erreichen.

Listing 6: Verwendung der »propertyfile«-Task

```
121 <target name="inc-major-pf"
122         description="Increment major-level with
propertyfile-task">
123   <propertyfile
124      file="version2.properties"
125      comment="Standard mechanism">
126    <entry  key="version.pl" value="0"/>
127    <entry  key="version.minor" value="0"/>
128    <entry  key="version.major" type="int"
operation="+" default="0"/>
129    <entry  key="version2"
130            value="${version.major}.${version.
minor}.${version.pl}"/>
131  </propertyfile>
132 </target>
133
```

```
134 <target name="inc-minor-pf"
135         description="Increment minor-level with
propertyfile-task">
136   <propertyfile
137      file="version2.properties"
138      comment="Standard mechanism">
139    <entry  key="version.pl" value="0"/>
140    <entry  key="version.minor" type="int"
operation="+" default="0"/>
141    <entry  key="version.major" default="0"/>
142    <entry  key="version2"
143            value="${version.major}.
${version.minor}.${version.pl}"/>
144  </propertyfile>
145 </target>
146
```

```
147 <target name="inc-pl-pf"
148         description="Increment patch-level with
propertyfile-task">
149   <propertyfile
150      file="version2.properties"
151      comment="Standard mechanism">
152    <entry  key="version.pl" type="int"
operation="+" default="0"/>
153    <entry  key="version.minor" default="0"/>
154    <entry  key="version.major" default="0"/>
155    <entry  key="version2"
156            value="${version.major}.
${version.minor}.${version.pl}"/>
157  </propertyfile>
158 </target>
```