Die Build-Werkzeuge

Apache Ant und Maven

Martin Meintel, Jürgen Schmiing

mema2@ba-horb.de, scju@ba-horb.de

BA-Horb,
Studiengang Informationstechnik, 5. Semester

Betreuender Dozent: Prof. Dr. Ing. Olaf Herden

Abstract

Gerade bei größeren Projekten speziell auf Java-Basis ist die Nutzung von Tools zur Unterstützung des Erstellens und des Verteilens unerlässlich. Da zusätzlich zum Build-Vorgang unter Umständen Dateien auf Web-Servern publiziert werden müssen, steigt der Aufwand für jeden einzelnen Programmierer stark an.

Die vorliegende Arbeit beschäftigt sich mit der Nutzung von Apache Ant, einem Build-Tool und make-Alternative für die Java-Welt. Es werden neben den grundlegenden Elementen wie Targets, Properties und Tasks zunächst die Funktionsweise im Wesentlichen erläutert. Es werden Beispiele zur Nutzung von Selektoren, Verwendung von Archiven, Erstellung von eigenen Tasks und ein Beispiel zum Aufbau eines Build-Scripts gegeben. Zudem wird Apache Maven, der Nachfolger von Ant vorgestellt und wesentliche Neuerungen erläutert.

Inhaltsverzeichnis

1 Einleitung

1.1 Allgemeines zu Ant

Um ihnen in dieser Einleitung einen groben Überblick über Ant zu verschaffen, ist es zunächst notwendig den Begriff des Build Werkzeuges näher zu erläutern. Unter einem Build Werkzeug wird ein Werkzeug verstanden, dass den zuvor programmierten Quellcode in eine Zielsprache mit minimalem Aufwand überführt.
Gerade bei komplexerem Sourcecode, welcher sich nicht nur auf einige wenige Dateien und Klassen beschränkt, ist das effiziente Kompilieren eine große Herausforderung. So bestehen zwischen verschiedenen Dateien Abhängigkeiten. Nehmen wir das Beispiel einer Klasse, die verschiedene Methoden beinhaltet. Ändert der Entwickler die Klasse, durch beispielsweise das Umbenennen einer Methode, so müssen alle Klassen neu kompiliert werden, welche davon betroffen sind, oder zumindest auf korrekte Einbindung überprüft werden. Natürlich könnte man auch einfach das komplette Projekt neu übersetzen, aber jeder der schon mal an einem größeren Projekt gearbeitet hat und es kompiliert hat, wird zustimmen, dass dies ein erheblicher Zeitaufwand darstellt. Beispielsweise das Kompilieren eines neuen Kernels. Deshalb ist ein gutes Build Werkzeug in der Lage, diese Abhängigkeiten aufzulösen und zu erkennen, welche Dateien bzw. Klassen neu kompiliert werden müssen. Andernfalls kommt es sehr wahrscheinlich zu Compiler- oder Linkerfehlern.

Um verschiedene Ziele über ein Build Werkzeug verfolgen zu können, werden beispielsweise in den meisten Makefiles, wie auch in Ant, verschiedene Targets angegeben.
Ein Beispiel hierfür wäre der Aufruf ‘make’, ‘make install’ und ‘make modules_install’. Dieses Beispiel bezieht sich auf das make Tool unter Unix, welches in den 70er Jahren von Stuart Feldman entwickelt wurde und als einer der Vorreiter in den Build Werkzeugen zu betrachten ist. Beim ersten Aufruf wird das Default Target aufgerufen, welches zuvor definiert wird. Beim zweiten Aufruf wird das Target ‚install’ und beim dritten Aufruf das Target ‚modules_install’ aufgerufen. Die jeweiligen Targets können verschiedene Arbeitsschritte beinhalten, wie etwa das Kompilieren oder das Kopieren von Dateien.
Bevor make beispielsweise eine Datei neu kompiliert überprüft es, ob Dateien bereits existieren und neu zu übersetzen sind. Existieren die Dateien noch nicht, werden sie kompiliert. Andernfalls nimmt make den Zeitstempel dieser Version und vergleicht ihn mit den Zeitstempeln aller Dateien, welche in Abhängigkeit zu dieser in der Abhängigkeitszeile angegeben sind. Ist dieser aktueller als der der alten Version, werden diese Datei und alle von dieser Datei abhängigen Dateien neu übersetzt (mehr zum Build Werkzeug make in [SCHA00]).

Ebenso stellt ein Build Tool weitere Aktionen zur Verfügung. Beispiele für diese Aktionen auch genannt Tasks könnten etwa das automatische Aufrufen weiterer Werkzeuge zum Beispiel für das Erstellen einer Dokumentation aus dem Quellcode, oder das Zusammenführen von verschiedenen Programmen zu einem Gesamtpaket sein. Ant ist ein Open Source Build Werkzeug, welches in Java programmiert wurde. Es wurde 1998 von James Duncan Davidson entwickelt. Zunächst war Ant lediglich ein Build Werkzeug für den ersten Tomcat Server. Später wurde klar, dass Ant viele Probleme lösen würde, welche mit normalen Makefiles auftraten. So wurde Ant aus dem Tomcat Projekt herausgenommen und quasi selbstständig. Ant ist heute in der Java Welt, nicht zuletzt auf Grund seiner hervorragenden Eigenschaften sehr weit verbreitet. Ant heisst der Übersetzung nach Ameise. Jedoch steht Ant eigentlich für Another Neat Tool (Noch ein hübsches Werkzeug). Nichts desto trotz bestehen Parallelen zu Ameisen. So ist Ant zwar verhältnismäßig klein, dennoch ist es in der Lage uns eine Menge Arbeit abzunehmen. Ant ist plattformunabhängig. Da es in Java geschrieben wurde, ist lediglich eine Java Laufzeitumgebung erforderlich. Das Logo von Ant deutetet auf Grund seiner spitzen Klammern, in welchen “Apache Ant” geschrieben steht bereits darauf hin, dass

XML bei Ant eine Rolle spielt. Ebenso wie bei Makefiles gibt es auch in Ant eine Konfigurationsdatei. Diese wird Build Datei genannt. Die Build Datei ist jedoch im Gegensatz zu den meisten Makefiles in XML dargestellt. (Allgemeines zu Ant in [WIKI05])

1.2 Motivation

Der große Vorteil von Ant im Gegensatz zu normalen Makefiles ist, dass die verwendeten Tasks wie beispielsweise das Kopieren, das Versenden von E-Mails, das Löschen, Verschieben und das Umbenennen von Dateien, um nur wenige der derzeit implementierten Tasks zu nennen, bereits in dem Tool implementiert sind. Dies stellt sicher, dass die Plattformunabhängigkeit gewährleistet bleibt. Es besteht zwar auch in Ant die Möglichkeit Systemcalls aufzurufen, jedoch wird davon auf Grund des möglichen Verlustes der Plattformunabhängigkeit abgeraten. Werden neue Tasks benötigt, so können diese in Form von eigenen Java Programmen implementiert werden. Ebenso werden durch das Beschreiben der Konfigurationsdateien in XML von vorneherein plattformabhängige Faktoren wie beispielsweise das unterschiedliche Darstellen von Zeilenumbrüchen umgangen. Es besteht ebenso die Möglichkeit Ant in eigene Anwendungen zu integrieren, um so verschiedene Aufgaben das Tool übernehmen zu lassen. Beispiel hierfür wäre ein Installationsprogramm.

1.3 Einsatzgebiete

Das eigentliche Einsatzgebiet von Ant ist die Generierung von komplexen Java Applikationen.

1.4 Installation von Ant

Da Ant selbst auf Java basiert, ergibt sich eine breite Palette an möglichen Betriebssystemen, dazu zählen im wesentlichen MS Windows 9x, NT, Linux, Unix, HP-UX, MacOS X, Novell Netware 6 und OS/2 Warp. Der Betrieb von Ant erfordert eine lauffähige JDK ab Version 1.2, die auf dem System installiert sein muss. Die Umgebungsvariable `JAVA_HOME` muss auf das Verzeichnis der Java-Umgebung zeigen.

Zur Installation von Ant genügt es, das heruntergeladene Archiv in ein Verzeichnis zu entpacken, und die Umgebungsvariable `ANT_HOME` auf das entsprechende Verzeichnis zu setzen. Zur einfacheren Handhabung ist es sinnvoll, der Umgebungsvariable `PATH` den Ordner `ANT_HOME\bin` hinzuzufügen.

Eine genaue Anleitung zur Installation ist in [ANT05] zu finden.

2 Aufbau und grundlegende Elemente eines Buildskript

Das Buildskript besteht, wie auch in [FIE04] genannt, aus der Projektdefinition, verschiedenen Properties, Targets und Tasks. Abbildung 1 zeigt diese schematisch.

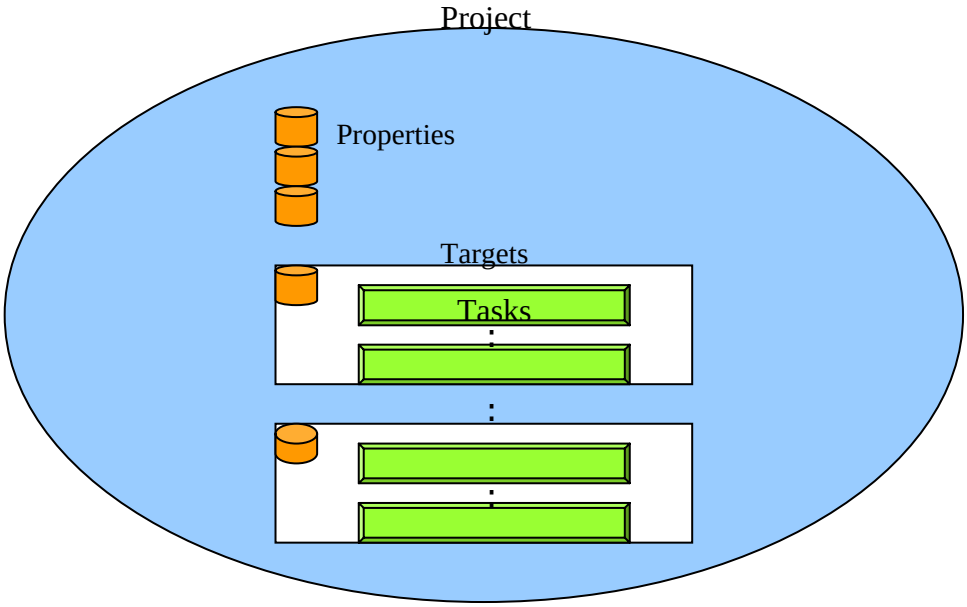

Abbildung 1: Wesentliche Elemente eines Buildskript

2.1 Projektdefinition

Die Projektdefinition beinhaltet den Namen des Projektes, sowie das Standardtarget, welches aufgerufen wird, sofern Ant keine Targets mitgegeben werden. Ebenso beinhaltet es das Basedirectory, welches das Ausgangsverzeichnis darstellt.

Beispiel:

```
<project name="testapplikation" default="compilieren" basedir=".">
```

Da die Builddatei in wohlgeformtem XML dargestellt ist und der Projektname zuerst angegeben wird, ist dieses Tag auch als letztes zu schließen.

2.2 Properties

Properties können als Variablen betrachtet werden (vgl. [LEM02]). Sie sind Case Sensitiv. Properties können nur einmal gesetzt werden. Auf einige Variablen kann direkt zugegriffen werden. So ist zum Beispiel die Variable ‚user.home' durch die Java Virtual Machine bereits definiert. Es gibt insgesamt sechs Möglichkeiten, Properties zu setzen:

- Durch das Angeben von Property Name und dem ‚value' Attribut
  `<property name="sourcedir" value="./source/"/>`
- Durch das Angeben von Property Name und dem ‚refid' Attribut.
- Durch das Setzen des ‚file' Attributes mit dem Dateiname der Property Datei.
  `<property file="file.suffix"/>`

- Durch das Setzen des ‚url' Attributes, von wo die Properties geladen werden sollen. (Die URL muss an eine Datei gerichtet sein, welche im richtigen Format dargestellt ist)
  ```
  <property url="http://www.domain.org/property.file"/>
  ```

- Durch das Setzen des ‚resource' Attributes mit dem Dateinamen der zu ladenden Datei. Eine Ressource ist eine Property Datei im aktuellen Classpath oder dem angegebenen Classpath.
  ```
  <property resource="file.prop"/>
  ```

- Durch das Setzen des ‚environment' Attributes, welches mit einem Präfix zu benutzen ist. Properties werden dann für jede Umgebungsvariable definiert durch das Angeben der entsprechenden Namen der Umgebungsvariablen. Achtung! Auch die Umgebungsvariablen in Windowssystemen sind hier Case Sensitiv.
  ```
  <property environment="env"/>
  <echo message="ANT Home Directory: ${env.ANT_HOME}"/>
  ```

Beispiel:

```
servername=www.ba-horb.de
serverport=8080
url=http://${servername}:${serverport}/
```

2.3 Nutzung von Property-Dateien

Wie bereits zuvor erwähnt, können Properties als dynamische Elemente verwendet werden, beispielsweise für Pfadangaben oder für bedingte Verarbeitung. Gerade für Benutzer, die nicht mit dem Umgang von Ant vertraut sind, oder das Build-Script recht komplex ist, bietet es sich an, Properties in so genannte Property-Files auszulagern. Dabei wird statt der eigentlichen Definition der Properties im Build-Script nur noch ein Verweis auf die Datei gegeben, zum Beispiel:

```
<property file="c:\myproject\project.properties"/>
```

In der Property-Datei tauchen nur noch die Definitionen der einzelnen Variablen auf:

```
Build.dir="c:\myproject\build"
Src.dir="c:\myproject\src"
Doc.dir="c:\myproject\doc"
```

Die Nutzung von Property-Files erlaubt damit zum Beispiel eine flexible Anpassung von Pfadangaben, die sich von Mitarbeiter zu Mitarbeiter unterscheiden können. Damit ist auch sichergestellt, dass Erweiterungen am Buildskript nicht die Folge haben, dass sämtliche Pfadangaben nach dem Update neu gesetzt werden müssen.

Anmerkungen zu Property-Dateien:

- Falls die angegebene Datei nicht existiert, wird keine Fehlermeldung ausgegeben, außer die –verbose Option wurde angegeben.
- Aneinandergereihte Leerzeichen werden nicht zusammengefasst.
- Um Sonderzeichen anzugeben wird ein Backslash verwendet.
- Properties können verschachtelt werden

Werte können ihren Properties direkt zugewiesen werden.

2.4 Tasks

In den Tasks werden die einzelnen Aufgaben ausgeführt. Sie können mit Methoden verglichen werden. Die Tagnamen entsprechen den einzelnen Tasks. Die Attribute sind hier je nach Task verschieden. Je nach Task ist es auch möglich, dass weitere Tags benötigt beziehungsweise verwendet werden können. Derzeit sind über 150 Tasks in Ant eingebaut. Jedoch können, wie bereits im Punkt 'Motivation' erwähnt, auch eigene implementiert werden, da Ant eine solche Schnittstelle bietet. Beispiele für Tasks sind zum Beispiel das Kompilieren von Quellcode, CVS zum Durchführen von CVS Operationen oder Replace zum Ersetzen von Text in Dateien. Dabei wird zwischen zwei verschiedenen Arten von Tasks unterschieden: Zum einen gibt es Built-In Tasks und zum anderen gibt es Optional Tasks. Für die Optional Tasks werden zusätzliche externe Libraries benötigt.

Hier einige Tasks mit Beschreibung:

Task	Beschreibung
javac	ruft den Java Compiler auf
java	ruft eine Java Klasse auf
copy	kopiert eine oder mehrere Dateien
delete	löscht eine oder mehrere Dateien / Verzeichnisse
move	verschiebt eine Datei in ein Verzeichnis oder benennt sie um
mkdir	erstellt ein Verzeichnis
synch	synchronisiert zwei Verzeichnisse
ant	führt Ant auf ein entsprechendes Buildfile aus.
antcall	Aufruf eines Targets
mail	versendet eine eMail

Beispiel:

```
<copy todir="${newdir}" >
    <fileset dir="fromdir"
        <includes="**/*.java"/>
        <excludes="**/*old"/>
</copy>
```

Anmerkung: **/* bedeutet, dass rekursiv kopiert wird.

2.5 Targets

In einem Projekt kann es mehrere Targets geben. Das Standard Target wird im Projekt Tag durch das Attribut ‚default' definiert. Dieses Target wird aufgerufen, falls Ant kein Target mitgegeben wird. In Targets können Tasks und Properties definiert werden. Targets sollen zur Strukturierung dienen und können voneinander abhängig und ineinander verschachtelt sein. Die Strukturierung wird unter anderem dadurch erreicht, dass mehrere Tasks in einem Target zu einer Einheit zusammengefasst werden können. Sollen verschiedene Targets voneinander abhängig sein, so kann ist dies mit Hilfe des Attributes `depends` ausgedrückt werden. Mit Hilfe von Attributen wie zum Beispiel `if` und `unless` können die Verfügbarkeiten von Properties überprüft werden, um sicherzustellen, dass diese auch existieren beziehungsweise nicht existieren, bevor ein Target ausgeführt wird. Das Attribut `if` setzt für den weiteren Verlauf voraus, dass der Property vorhanden ist. Bei `unless` ist das Gegenteil der Fall.

Beispiel:

```
<target name="compile" if="source">
    <echo message="Compiling started…"/>
    <javac srcdir="${source}" destdir="${build}"/>
    <echo message="Mission accomplished"/>
</target>
```

Innerhalb eines Targets besteht die Möglichkeit ein weiteres Target aufzurufen, auch ohne, dass Abhängigkeiten definiert werden müssen. Dies kann über zwei Wege geschehen. Zum einen ist es mittels des Tasks ‘antcall’ möglich ein in der selben Builddatei liegendes Target aufzurufen. Zum anderen kann mittels des Tasks ‘ant’ ein Target einer anderen Builddatei aufgerufen werden. Dabei besteht die Möglichkeit, die Properties an das Zieltarget weiterzugeben. Standardmäßig ist diese Funktion aktiviert. Soll diese deaktiviert werden, so muss man das Attribut ‘inheritall’ auf false setzen. Dadurch besteht die Möglichkeit, dass Properties in den darauf folgenden Targets neu definiert werden können, was beim Aufruf durch ‚depends’ nicht möglich ist. Ebenso kann der Task nach einem beliebigen Task ausgeführt werden, während der Aufruf durch ‘depends’ stets zu Beginn ausgeführt wird, um der Abhängigkeitsbedingung nachzukommen.
Mittels des Task ‘parallel’ besteht die Möglichkeit, Targets scheinbar (OS-Scheduler) parallel ablaufen zu lassen. Um auch hier auf eventuelle Abhängigkeiten reagieren zu können, gibt es auch hier die Möglichkeit auf bestimmte Events zu warten. Beispielsweise bis die Existenz einer Datei bestätigt wird oder mittels einer Zeitkonstanten.

Beispiel:

```
<target name=”bla_parallel”>

    <parallel>
        <antcall target=“a“/>
        <antcall target=“b“/>
        <antcall target=“c“/>
     </parallel>

     <waitfor maxwait="20" maxwaitunit="second">
        <available file="errors.log"/>
     </waitfor>

</target>
```

Dieses Beispiel ruft die Targets ‘a’, ‘b’ und ‘c’ parallel auf. Anschließend wird maximal 20 Sekunden lang gewartet, bis die Datei errors.log existiert. Sind die 20 Sekunden abgelaufen und es existiert die Datei nicht, wird eine Fehlermeldung ausgegeben.

Im Wesentlichen sind für Ant drei Quantoren von Bedeutung: Der Stern (*), der als Platzhalter für kein, ein oder mehrere Zeichen dient. Als Erweiterung dazu existiert der Doppelstern (**), der für kein, ein oder mehrere Unterverzeichnisse verwendet werden kann. Als dritter Quantor existiert das Fragezeichen (?), der anstelle von genau einem Zeichen stehen kann.
Wie in anderen Scriptsprachen auch üblich, lassen sich die Quantoren in beliebiger Kombination verwenden.

3 Ein einfaches Buildskript

```
<project name=“something“ default=“make“ basedir=“.“>
    <property name=”source_dir” value=”source”/>
    <property name=”last_src_dir” value=”last”/>
    <property name=”dest_dir” value=”build”/>
    <property file=”propertyfile.name”/>
    <property environment=”env”/>

    <target name=”clean”>

        <delete verbose=”true”>
            <fileset dir="${dest_dir}" includes="**/*.class"/>
        </delete>

    </target>
```

```
    <target name=”make”>

        <copy todir=”${dest_dir}”>
             <fileset dir=”${last_src_dir}”>
                 <exclude name=”**/*~”/>
             </fileset>
        </copy>

        <javac  srcdir=”${source_dir}” destdir=”${dest_dir}”>
        </javac>

    </target>

    <target name=”manual”  depends”make”>

        <copy todir=”${env.TEMP}” >
             <fileset dir=”${dest_dir}”>
                 <include name=”**/*.html”/>
             </fileset>
         </copy>

         <mail mailhost=”smtp” mailport="25" user=”nobody”
          password=”seCr3t” subject=”New XY Version available”
          from=”me@me.com” tolist=”all@sameinterest.com”
          files=”${dest_dir}/info.log”>

             <message>
                    Hi, es ist wieder eine neue Version verfügbar.
             </message>

       </mail>

    </target>

</project>
```

3.1 Beschreibung

Zu Beginn des einfachen Buildskript, welches die build.xml darstellt, steht zunächst das Projekttag, welches den Namen des Projektes, sowie das Standardtarget und den absoluten Pfad zum Projekt enthält. Darunter werden zunächst drei Properties mittels der Attribute ‚name’ und ‚value’ definiert. Auf diese Definitionen folgt der Import von weiteren Properties über ein Propertyfile. Zuletzt werden die im System vorhandenen Umgebungsvariablen als Properties eingebunden. Auf die Definition der Properties folgt die Definition des Targets ‚clean’, in welchem der Task ‚delete’ aufgerufen wird. In diesem Task werden alle Dateien im Verzeichnis der Variable ‚dest_dir’ und Unterverzeichnissen gelöscht, welche die Endung ‘.class’ besitzen. Beim Löschen der Dateien werden die jeweiligen Namen ausgegeben. Beim Ausführen des Targets ‚make’, welches zugleich das Standardtarget in diesem Projekt darstellt, werden zunächst alle Dateien und Unterverzeichnisse aus dem Verzeichnis der Variablen ‚src_dir’, welche keine Tilde enthalten in das Verzeichnis des Properties ‘`last_src_dir`’ kopiert. Daraufhin werden die Dateien, welche im Verzeichnis des Properties ‚src_dir’ liegen kompiliert und die Binaries im Verzeichnis von ‚dest_dir’ angelegt. Das letzte Target, welches im Beispielskript vorkommt, ist das ‚manual’ Target, welches vom ‚make’ Target abhängig ist, was bedeutet, dass das ‘make’ Target zuerst aufgerufen wird, falls sich Dateien geändert haben. Ist die Abhängigkeitsfrage geklärt, so werden mittels des ‘copy’ Task alle Dateien mit der Erweiterung html aus ‘dest_dir’ in das Verzeichnis kopiert, welches durch die Umgebungsvariable ‚MANUALS’ repräsentiert wird. Der darauf folgende Task wird dazu verwendet, eine eMail zu verschicken. Im Mailtag wird im Beispiel der Mailhost, der Mailport, der User des Mailaccounts, dessen Passwort, der

Betreff der Mail, der Absender, sowie die Empfängeradresse/n angegeben. Ebenso eine Datei, welche mit der Mail verschickt werden soll.

Beispielaufruf von Ant:

```
ant –f build_new.xml –logfile=log.txt
```

Beim Beispielaufruf wird das Buildfile build_new.xml aufgerufen. Während des Durchlaufs wird in die Logdatei log.txt mitgeschrieben. Standardmäßig wird versucht, die Datei build.xml aufzurufen.

4 Nutzung von Selektoren

Für sämtliche Tasks, die zur Behandlung von Dateien und Verzeichnissen dienen, ist es möglich neben einzelnen Dateien auch eine Auswahl von Dateien und Verzeichnissen anzugeben. Dies erleichtert die Auswahl selbst und die Übersichtlichkeit des Build-Script. Die einfachsten Selektoren sind dabei sicherlich die schon erwähnten Wildcards *, ** und das Fragezeichen.

4.1 Die Gruppierungen FileSet und DirSet

Die Gruppierung FileSet dient zur Selektion von Dateien in einem angegebenen Ordner. Dabei können Dateien über die bekannten Quantoren zur Liste hinzugefügt oder explizit herausgenommen werden. Das Einbeziehen von Dateien in Unterverzeichnissen ist ebenfalls möglich.
Das nachfolgende Beispiel fügt alle Java-Dateien (auch in Unterverzeichnissen) der Liste hinzu, die die Erweiterung `java` hat. Dabei werden Dateien in Unterverzeichnissen mit der Zeichenfolge „Test“ nicht hinzugefügt.

```
<fileset dir="${project.src}" casesensitive="yes">
  <include name="**/*.java"/>
  <exclude name="**/*Test*"/>
</fileset>
```

Die Gruppierung DirSet ist grundsätzlich gleich aufgebaut, erlaubt dabei allerdings nur die Auswahl von Verzeichnissen. Denkbar wäre hier ein ähnliches Beispiel, das alle Verzeichnisse im Quellverzeichnis exklusive der Ordner mit dem Text „Test“ im Namen auswählt.

```
<dirset dir="${project.src}">
  <include name="apps/**/classes"/>
  <exclude name="apps/**/*Test*"/>
</dirset>
```

4.2 Die Gruppierung FileList

Sofern es nötig ist, lässt sich bei Ant auch eine genaue Liste von Dateien spezifizieren. Dazu kann der Eintrag FileList verwendet werden. Wie auch bei FileSet und DirSet wird ein Basisverzeichnis angegeben, in dem sich die auszuwählenden Dateien befinden. Die Dateien selbst werden durch Leerzeichen oder Kommata getrennt.
Im folgenden Beispiel werden die drei Dateien Readme.doc, Install.doc und Features.doc aus dem Dokumentationsverzeichnis des Projekts der FileList mit der ID Infofiles hinzugefügt.

```
<fileset id=“infofiles“
         Dir=“${project.doc}“
         Files=“Readme.doc, Install.doc, Features.doc“/>
```

4.3 Die Gruppierung PatternSet

Über die Gruppierung PatternSet lässt sich eine Liste von Dateien auswählen, ähnlich wie bei DirSet und FileSet. Im Gegensatz dazu ist es beim PatternSet allerdings möglich, das Hinzufügen bzw. das Entfernen von Dateien abhängig von Properties zu machen. Dazu werden die Schlüsselwörter `if` (falls gesetzt) und `unless` (falls nicht gesetzt) benutzt. Dadurch lassen sich zum Beispiel unterschiedliche Versionen von Anwendungen, abhängig von dem gewählten Funktionsgrad bauen.
Im folgenden Beispiel werden alle Dateien aus dem Quellverzeichnis dem PatternSet hinzugefügt, und nur bei gesetztem Attribut Test auch die Quellen in den Testverzeichnissen.

```
<patternset id="sources">
  <include name="${project.src}/**/*.java"/>
  <exclude name="${project.src}**/*Test*" unless=“Test“/>
</patternset>
```

5 Integration von Versionsmanagementsystemen

Gerade bei größeren Projekten, bei denen die Vorzüge von Ant auch deutlicher werden, werden üblicherweise Versionsmanagementsysteme eingesetzt. Bekannte Vertreter sind dabei ClearCase, Visual Source Safe, Perforce, StarTeam oder auch CVS. Bei der Teamarbeit sind solche Tools unverzichtbar, um Inkonsistenzen zwischen verschiedenen Codeversionen zu vermeiden.
Im Folgenden sollen ein paar kurze Bespiele zur Verwendung des Concurrent Version System (CVS) zusammen mit Ant gegeben werden; eine Unterstützung der zuvor genannten Systeme ist ebenfalls integriert.

5.1 Anmeldung bei CVS

Um Zugriff auf die Sourcen zu erhalten, ist es zunächst notwendig, sich beim System anzumelden. Dazu sind ein gültiger Benutzername und ein Passwort erforderlich. Das folgende Beispiel zeigt das Schema eines Logins:

```
<target name="login_cvs" >
       <cvspass
       cvsroot=":pserver:Benutzername@Server:/home/benutzer/
              repository"  password="123456" />

       [...]
</target>
```

5.2 Herunterladen (Auschecken) von Sourcen

Eine der sicherlich wichtigsten Aufgaben, der bei der Zusammenarbeit zwischen Ant und einem Versionsmanagementsystem auftaucht, ist das Herunterladen oder Auschecken von Sourcen. Damit wird der aktuelle Stand der Daten aus dem Repository in das angegebene Verzeichnis heruntergeladen, wie auch das folgende Beispiel zeigt:

```
<target name="checkout" >
       <cvspass
       cvsroot=":pserver:Benutzername@Server:/home/benutzer/
              repository"  password="123456" />

       <cvs package="BeispielProjekt" dest="${project.src}" />
</target>
```

5.3 Sichern (Einchecken) von Sourcen

Das Einchecken von Sourcen scheint auf den ersten Blick zwar nicht unbedingt eine sinnvolle Verwendung zu finden, da speziell beim Auftreten von Konflikten zwischen bereits existentem Code und dem geänderten Code eine Behandlung der Probleme unmöglich ist. Dennoch wäre zum Beispiel die Situation denkbar, nach einem automatischen Testlauf die Ergebnisse ins Repository zu sichern. Das folgende Beispiel zeigt das Einchecken der Testergebnisse:

```
<target name="checkin" >
       <cvspass
       cvsroot=":pserver:Benutzername@Server:/home/benutzer/
              repository"  password="123456" />

       <cvs dest="${project.test}/Results" command="commit -m
              'Testergebnisse'"/>

</target>
```

Neben den hier angegebenen Kommandos ist natürlich der weitaus größere Funktionsumfang des CVS und der anderen Versionsmanagementsysteme in Ant verfügbar. Im Regelfall unterstützen jedoch Entwicklungsumgebungen wie Eclipse bereits die Kommunikation mit Versionsmanagementsystemen, sodass diese Routinen in Ant nicht benötigt werden. Eine größere Liste inklusive Beispiel ist in [HOL05] zu finden.

6 Verwendung von Archiven

Gerade bei der Entwicklung von Webanwendungen ist es wichtig, mit Archiven arbeiten zu können, die später auf dem Webserver publiziert werden sollen. Zur Behandlung dieser Archive bietet Apache Ant ebenfalls Routinen an.

Im Wesentlichen ist bei der Behandlung von Archiven die Angabe des entsprechenden Archivs und des Verzeichnisses, in dem die Quelldateien sind, erforderlich. Das folgende Beispiel zeigt das Packen aller Dokumente aus dem Verzeichnis htdocs in die Datei manual.zip:

```
<zip destfile="${dist}/manual.zip"
     basedir="htdocs/manual"
/>
```

Bei Angabe der Dateien ist es ebenfalls möglich, eine vordefinierte Liste oder ein PatternSet anzugeben.

Umgekehrt ist es natürlich auch möglich, ein vorhandenes Archiv in einen Ordner auf der Festplatte zu entpacken. Das folgende Beispiel entpackt alle Dateien aus dem Archiv project.zip in den Sourcen-Ordner des Projekts.

```
<unzip src="project.zip"
       dest="${project.src}">
</unzip>
```

Ähnlich zur Syntax für ZIP-Dateien ist die Verarbeitung von JAR, TAR, EAR oder WAR-Archiven. Gerade bei JAR Dateien können zusätzlich Manifest-Informationen angegeben werden, also Informationen über die auszuführende Klasse, den Entwickler oder über Signierung. Als Manifest-Informationen können zum einen die Einstellungen direkt angegeben werden oder eine externe Manifest-Datei eingebunden werden.

```
<jar destfile="test.jar" basedir=".">
   <include name="build"/>
```

```
    <manifest>
      <attribute name="Built-By" value="${user.name}"/>
      <section name="common/class1.class">
        <attribute name="Sealed" value="false"/>
      </section>
    </manifest>
 </jar>
```

Für die Archive WAR und EAR können zusätzlich Dateien angegeben werden, die als Deployment-Deskriptoren benutzt werden sollen.

7 Empfehlung zur Gliederung eines Build-Script

Die Abbildung 2 soll als Bespiel dienen, wie ein gut strukturiertes Build-Script aussehen könnte:

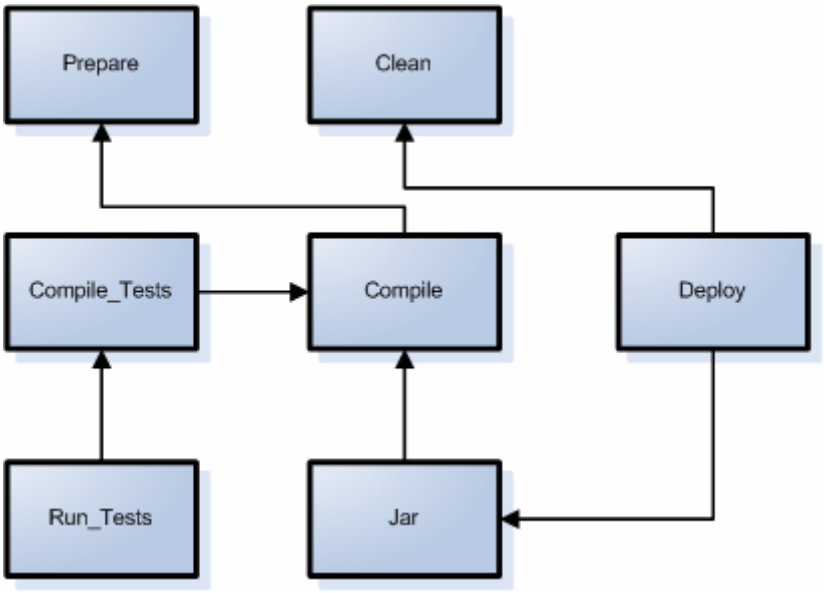

Abbildung 2: Exemplarische Gliederung eines Build-Skript

Die Pfeile im der Abbildung dienen dabei der Veranschaulichung von Abhängigkeiten, die in den einzelnen Tasks mit dem Schlüsselwort `depends` angegeben werden. Nachfolgend sollen die einzelnen Phasen kurz näher erläutert werden:

Prepare
In der Prepare-Phase lassen sich vorbereitende Aktivitäten erledigen, zum Beispiel das Überprüfen von bestimmten Bedingungen oder Vorbereitungen für das Ausgabeverzeichnis.

Clean
Die Clean-Phase dient im Wesentlichen dazu, bereits erstellte Dateien und Klassen zu löschen. Damit soll sichergestellt werden, dass alle Dateien neu erzeugt werden.

Compile
Sämtliche Aktivitäten, die mit dem Kompilieren von Dateien zusammenhängen, können in der Compile-Phase untergebracht werden.

Compile_Tests
Die Compile_Tests-Phase kann dazu genutzt werden, evtl. vorhandene JUnit-Tests zu kompilieren. Durch die Auslagerung der Tests lässt sich eine einfache bedingte Kompilierung für dieses Target ermöglichen.

Run_Tests
Das Target Run_Tests kann zum Ausführen von automatisierten Tests genutzt werden, zum Beispiel für JUnit-Tests. Die Ergebnisse des Testlaufs können dabei neben einer Logdatei auch in ein Versionsmanagementsystem eingecheckt werden oder per E-Mail verschickt werden.

Jar
Über das Ziel Jar lassen sich üblicherweise Dateien in Archive zusammenfassen, beispielsweise zu jar-, war- oder ear-Archiven. Diese werden vor allem bei Web-Anwendungen benötigt.

Deploy
Das Ziel deploy dient zum Kopieren der Dateien ans Ziel. Durch die Einbeziehung des Targets Clean wird sichergestellt, dass für die Verteilung alle Dateien neu erstellt werden.

8 Erweiterungen

Neben der großen Anzahl von Möglichkeiten, die Ant bereits von Haus aus bietet, sind eine Vielzahl von Erweiterungen verfügbar. Ein kleiner Auszug von verfügbaren Erweiterungen und den jeweiligen Internetadressen ist im Folgenden abgedruckt:

Verarbeitung von XSL Dateien (Formatierung von XML Daten)
http://xml.apache.org/xalan-j/index.html

JUnit Tasks – Integration von JUnit-Tests
http://www.junit.org/

Bibliothek zur formatierten Ausgabe von Logging-Meldungen
http://jakarta.apache.org/commons/logging/index.html

Integration verschiedener Tex-Varianten
http://www.dokutransdata.de/ant_latex/

Zugriff auf Netzwerkdienste (FTP, Telnet, …)
http://jakarta.apache.org/commons/net/index.html

Versenden von Mails
http://java.sun.com/products/javamail/

8.1 Entwicklung eigener Erweiterungen

Natürlich ist es auch möglich, eigene Erweiterungen für Ant zu schreiben. Dabei muss die bestehende Klasse Task.java erweitert werden. Sofern es sich um einen Task handelt, der keine weiteren Eingaben erwartet, genügt der folgende Code:

```
package de.meinefirma;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

public class EigenerTask extends Task {
    private String msg;

    // Die Methode wird beim Aufruf des Tasks ausgeführt
    public void execute() throws BuildException {
        System.out.println(msg);
    }
```

```
    // Setter für das Attribut
    public void setMessage(String msg) {
        this.msg = msg;
    }
}
```

Um den selbstentwickelten Task einzubinden, der als Attribut noch eine Nachricht erwartet, ist das folgende Skript erforderlich:

```
<taskdef name="EigenerTask"
         classname="de.meinefirma.EigenerTask"/>

<target name="main">
    <EigenerTask message="Aufruf erfolgreich!"/>
</target>
```

9 Apache Maven – der Nachfolger von Ant

Maven ist ebenso wie Ant ein Build Werkzeug. In Maven wurden diverse Verbesserungen vorgenommen, die als Schwächen von Ant angesehen wurden. Dabei ist Maven eine echte Obermenge zu Ant. Das bedeutet, dass alle Ant Tasks ebenso in Maven verwendet werden können. Die bisherige Vorgehensweise bei Ant war diejenige, dass für jedes Projekt ein eigenes Build Skript geschrieben wurde, in welchem die verschiedenen Targets mit ihren einzelnen Tasks aufgelistet wurden. Jedoch bestand keine Möglichkeit, die jeweiligen Targets für weitere Projekte zu verwenden. Sie mussten stets angepasst werden. Dies hatte zur Folge, dass man für jedes Projekt ein eigenes, von anderen verschiedenes Build File hatte. Bei Maven ist der Hauptgedanke dieser, dass der Ablauf beim Erstellen von Projekten häufig der Gleiche ist. Daher werden die Build Skripte durch Plug-Ins und dem Projekt Object Model (POM) ersetzt. Die Verzeichnisstruktur innerhalb der verschiedenen Projekten wird von Maven vorgegeben. So müssen innerhalb des Projektverzeichnisses die Dateien ‘project.xml’, ‚maven.xml’ und ‘project.properties’ vorhanden sein. Ebenso müssen sich im Projektverzeichnis die beiden Verzeichnisse ‘src’, in welchem sich die Dateien befinden, welche in einem Versionskontrollsystem enthalten sind, und ’target’ befinden, in welchem die generierten Dateien gespeichert werden. Die Plug-Ins entsprechen den Tasks aus Ant. Die Projektkonfigurationsdatei, welche durch das POM in Form von XML dargestellt ist, enthält die Informationen, welche die einzelnen Plug-Ins zur Laufzeit benötigen. Ebenso generiert Maven Dokumentationen und Metriken zur Bewertung der Software Qualität. Die Datei ‚maven.xml’ ist eine Build Datei, vergleichbar mit der ‚build.xml’ in Ant. In ihr können eigene Goals definiert werden. Hier kommt Jelly zur Anwendung, eine Scriptsprache im XML Format. Mit ihr können etwa Schleifen realisiert werden, was bei Ant beispielsweise nicht möglich ist. In dieser Konfigurationsdatei hat der Entwickler vollen Zugriff auf alle Maven Plugins und Java Objekte, welche man instanziieren kann. In einer ‚maven.xml’-Datei können ebenso wie in der Ant Builddatei mehrere Goals definiert und, wenn notwendig, Ant Tasks aufgerufen werden. Um Plugins wie etwa das ‚java:compile’ dennoch beeinflussen zu können, hat man die Möglichkeit sogenannte Pregoals und Postgoals zu definieren. Diese Goals lassen sich verschiedenen Plugins zuweisen und mit ihnen beispielsweise Kopiervorgänge, Textersetzungsvorgänge oder andere Tasks durchführen, bevor beziehungsweise nachdem ein Plugin ausgeführt wurde. Jedoch gilt, dass die Datei ‚maven.xml’ so kurz wie möglich gehalten und nicht als eine Ant ‘build.xml’ angesehen werden sollte.

9.1 Aufbau eines POM

In der ‚project.xml’ wird das Projekt beschrieben. Hier sollten alle Informationen enthalten sein, welche für den gesamten Lebenszyklus des Projektes von Interesse sind.

In ihr werden Abhängigkeiten, Informationen für Plugins, Projektinformationen wie aktuelle Version, Firma und viele mehr gespeichert.

9.2 Beispieldatei

```
<?xml version=“1.0“ encoding=“ISO-8859-1“?>
<project>

    <pomVersion>3</pomVersion>
    <groupId>Irgendein-Projekt</groupId>
    <artefactId>Artefakt-1</artefactId>
    <name>Beispiel Projekt</name>
    <currentVersion>1.0</currentVersion>

    <organization>
        <name>Ein Firmenname</name>
        <url>http://www.myurl.com</url>
        <logo>/images/logo.gif</logo>
    </organization>

    <description>  Maven  ist  mehr  als  ein  Build  Werkzeug!
</description>

    <developers>
        <developer>
            <name>Ein Top Mann</name>
            <email>top@mann.de</ email >
        </developer>
    </developers>

    <mailingLists>
        <mailingList>
            <name>Interessenten</name>
            <subscribe>will@ich.de</subscribe>
            <unsubscribe>will@nichmehr.de</subscribe>
        </mailingList>
    <mailingLists>

    <repository>
        <connection>cvs:server:ja@wohl.de/repository</connection>
    </repository>

    <dependencies>
        <dependency>
            <id>my-lib</id>
            <jar>my_lib.jar</jar>
        </dependency>
    </dependencies>

    <reports>
        <report>maven-checkstyle-plugin</report>
    </reports>

    <build>
        <sourceDirectory></sourceDirectory>

<unitTestSourceDirectory>${basedir}/testUnits</unitTestSourceDirec
tory>
        <unitTest>
            <includes>
                <include>**/*junit.java<include>
            </includes>
        </unitTest>
    </build>

</project>
```

In dieser Beispieldatei befindet sich zunächst die XML Deklaration. Auf sie folgt ebenso wie der Ant 'build.xml' der Projekttag, welcher alle Informationen beinhaltet. Zunächst wird die Version des POM angegeben. Darauf folgt die eindeutige Gruppen- und Artefact ID, der Projektname sowie der aktuelle Versionsstand. Ebenso werden in der Projektdatei Firma sowie Entwickler angegeben. Plugins wie beispielsweise das Java Plugin, welches verwendet wird, um Java Quellcode zu übersetzen, erhalten ihre Informationen aus den dementsprechenden Tags. Hierdurch erreichen wir im Gegensatz zu den Builddateien unter Ant eine klare Kapselung.

9.3 Beispiel einer Maven.xml

```
<project   default="any-build" xmlns:j="jelly:core"
 xmlns:u="jelly:util">

    <goal name=”doSth”>
       <task att=”value”/>
    </goal>

    <preGoal name=”java:copmile”>
        <copy file=”${maven.final.name}”>
         todir=”${home.prop}”/>
    </preGoal>

    <postGoal name=”jar:jar”>
        <echo>Jar File Successfully built!</echo>
    </postGoal>
</project>
```

Diese Datei enthält zunächst wiederum einen Projekttag. Auf ihn folgen hier die jeweiligen Goals. Das aufgeführte PreGoal wird ausgeführt, bevor das Goal java:compile aufgerufen wird. Das PostGoal im Beispiel würde ausgeführt werden, nachdem das Goal jar :jar aufgerufen wird.
Um Maven Plugins auszuführen, wird Maven der dementsprechende Plugin Name mitgegeben. Das entsprechende Plugin sucht dann in der POM Datei nach den entsprechenden Einstellungen. Existiert kein solches Plugin, wird in der 'maven.xml' nach einem entsprechenden Goal gesucht und entsprechend ausgeführt. Existiert ein solches Plugin, wird die Datei dennoch geöffnet und überprüft ob PreGoals oder PostGoals definiert sind. Falls sie sich nähers für Maven interessieren, finden sie ausführliche Beschreibungen unter [MAV05].

10 Fazit

Wie schon die gezeigten Beispiele deutlich demonstrieren, ist das Build-Tool Apache Ant deutlich mächtiger als beispielsweise Make. Auch wenn Ant in erster Linie auf die Erstellung von Java-Anwendungen spezialisiert ist, ist auch eine Nutzung mit anderen Compilern durch Definition eigener Tasks möglich. Durch die Strukturierung mithilfe von XML bleibt das Skript auch für noch unerfahrene Programmierer einfach verständlich und erweiterbar. Nicht nur aufgrund der Plattform-Unabhängigkeit bietet sich das Tool auch für die Verteilung an Webservern an. Auch die Erweiterbarkeit, entweder durch bereits vorgefertigte Lösungen, oder eigene Entwicklungen macht Apache Ant zu einem universellen Werkzeug für den Erstellungs-Vorgang.
Der Nachfolger Apache Maven versucht den Build-Prozess etwas abstrakter zu beschreiben. Dabei wird zunächst ein globaler Ablauf für alle Projekte festzulegt, um dann für jedes einzelne Projekt nur noch wichtige Parameter und Pfadangaben zu machen. Dies ist sicherlich für ähnlich strukturierte Projekte von Vorteil, bei unterschiedlichen und sehr spezifischen Erstellungs-Prozessen ist der Einsatz von Maven gegenüber Ant sicherlich abzuwägen.

11 Literaturverzeichnis

[HOL05] Steve Holzner. Ant: The definitive Guide. Second Edition April 2005. O'Reilly.

[SCHA00] Michael Schaarschmidt. Das make-Tool. Linux Magazin, Ausgabe 03/2000

[ANT05] Apache Ant. Die offizielle Dokumentation. http://ant.apache.org/manual/index.html Abruf am 24. Okt. 2005

[FIE04] Carsten Fiedler. Ausarbeitung Build-Tools. http://www-wi.uni-muenster.de/pi/lehre/ws0405/seminar/07BuildTools.pdf Abruf am 24. Okt. 2005

[LEM02] Simon Lembke. Ant - Das Apache Build-Werkzeug für Java. http://www.fh-wedel.de/~si/seminare/ws02/Ausarbeitung/8.ant/ant0.htm Abruf am 24. Okt. 2005

[MAV05] Apache Maven. Die offzielle Dokumentation. http://maven.apache.org/guides/index.html Abruf am 24. Okt. 2005

[WIKI05] Freie Enzyklopädie Wikipedia: Ant http://de.wikipedia.org/wiki/Ant Abruf am 24. Okt. 2005