

## ANT - Buildmanagement

### Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.  
©2018 Beuth Hochschule für Technik Berlin

## ANT - Buildmanagement



### Buildmanagement

## Lernziele und Überblick

In diesem Modul wird die Funktion des Buildmanagements beschrieben. Ziel ist es dabei, das Buildmanagement praktisch einsetzen und die Bedeutung innerhalb des Softwarelebenszyklus beurteilen zu können.



### Lernziele

- Kennenlernen des Historischen Ursprungs und des Werkzeugs „make“ sowie einiger make-Derivate
- Einordnung in den Entwicklungszyklus
- Einführung und praktischer Umgang mit dem Tool Ant



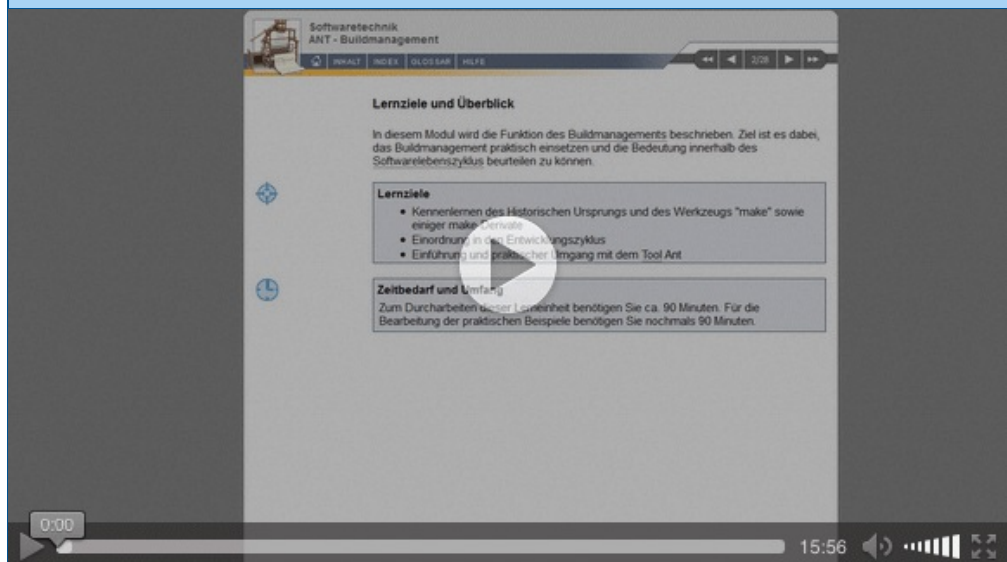
### Zeitbedarf und Umfang

Zum Durcharbeiten dieser Lerneinheit benötigen Sie ca. 90 Minuten. Für die Bearbeitung der praktischen Beispiele benötigen Sie nochmals 90 Minuten.



Film

### Webkonferenz zur Lerneinheit ANT



© Beuth Hochschule Berlin - Dauer: 15:55 Min. - Streaming Media 22.5 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

## 1 Buildmanagement und Continuous Integration

Zu den Managementaufgaben im Softwareentwicklungsprozess gehören neben dem Fehler-, Konfigurations-, Versions- und Releasemanagement auch das Buildmanagement.

In den 80er Jahren entstanden erste Werkzeuge zum Buildmanagement, die oftmals nur wenige zu kompilierende Dateien beinhalteten. Ein Jahrzehnt später wurden die **Softwaresysteme** wesentlich **größer**.

Zusätzlich wuchsen die **Abhängigkeiten** der zu entwickelnden Software. Im Entwicklungsprozess werden die unterschiedlichsten Komponenten von vielen Abteilungen erstellt, welche anschließend zusammengeführt werden müssen. Zusätzlich sind z. B. Applikationsserver nötig, um die Anwendung auszuführen.

In technischer Hinsicht kommen eine Vielzahl von **Protokollen** und **Diensten** hinzu, die entwickelt und aufgesetzt werden müssen. Beispielsweise müssen Datenbankverbindungen hergestellt werden, Web-Services erreichbar sein, etc. Oftmals befindet sich die Anwendung daher in einem kompletten SOA-Verbund und kann niemals isoliert betrachtet werden.

All dies bewirkt, dass die Erstellung einer lauffähigen Anwendung immer **komplexer** wird: herunterladen von Dateien (aus dem Versionsmanagement System wie CVS oder Subversion), übersetzen, testen, zu Archiven wie z. B. jar verpacken, kopieren, anwenden, Verbindungen aufbauen, Dienste starten, usw.

Erst im Anschluss ist die Anwendung fertig und lauffähig. Es wird sehr schnell klar, dass diese Abläufe **nicht** jedes Mal **von Hand** getätigt werden können.

Um Zeit zu sparen und Fehler zu vermeiden, muss dieser Prozess automatisiert werden.

Abhängigkeiten  
machen ein Build  
von Hand fast  
unmöglich



Hinweis

### Konfigurationsmanagement

Bei diesem Prozess spielt das Konfigurationsmanagement eine Rolle, das nach ISO 10007 aus folgenden Teilen besteht:

- Identifizierung
- Überwachung
- Buchführung
- Auditierung

Das einfachste Beispiel ist die Erstellung verschiedener Konfigurationen für Hardwareplattformen. Es wird z. B. ein Programm für LINUX und eines für Windows benötigt.

Erzeugung eines Prototyps

Ein weiterer Punkt, der sich aus dem agilen Development und den XP-Prinzipien ergibt, ist die Erzeugung eines **Prototyps**, der jederzeit - quasi mit einem Mausklick - erzeugt werden kann. Das ist eine wichtige Komponente, die immer häufiger von Auftraggebern gefordert wird, um kontinuierlich Feedback über den aktuellen Stand zu bekommen.

Moderne IDEs wie Visual Studio, Eclipse, IntelliJ, JBuilder haben aus diesem Grund Buildmanagementsysteme integriert, ein Build kann jedoch nicht immer aus der IDE heraus durchgeführt werden. In großen Industrieunternehmen oder Banken gibt es durchaus Builds, die als Batch-Prozess nachts gefahren werden müssen, da sie viele Stunden andauern.

## 1.1 Buildmanagement

Wie definiert sich nun der Begriff Buildmanagement?



Definition

### Buildmanagement

Unter Buildmanagement versteht man einen Prozess, bei dem ein (Software-)Produkt transparent und wiederholbar erstellt wird. Dies erfordert eine Automatisierung des Prozesses. Daher muss:

- eine stabile Umgebung vorliegen (z. B. Namensgebung), unter der der Prozess ablaufen kann.
- dieser Vorgang mit Hilfe einer Build-Sprache (in der Regel einer Skriptsprache) spezifiziert / programmiert werden.
- dieser Build mit einem Werkzeug (make, ant, etc.) ausgeführt werden können.
- der Build klar definierte Ergebnisse und logs liefern.

Oftmals sind die Build-logs leer oder nicht vorhanden, was, analog der UNIX Konventionen, auf einen positiven Buildabschluss hindeutet. Möglich sind sogar farbige Ausgaben und Reports, wie sie von JUnit bzw. entsprechenden Testrunnern bekannt sind.

Ziele des Buildmanagements

Ein Ziel des Buildmanagements ist es, den Build der Software **unabhängig vom** Software-Release (der **Version**) zu gestalten. Eine neue Version soll also nach Möglichkeit keine Änderungen im Buildskript nach sich ziehen.

Wichtig ist dabei, dass die Zielkonfigurationen und die verschiedenen Builds für unterschiedliche Ausgaben (Linux, Windows, etc.) oder Formate (jar, ear Enterprise Archiv, Webstart, etc.) genauso (auch im Buildfile) **dokumentiert** werden, wie in den verwendeten Werkzeugen (z. B. jdk 6 „Mustang“).

Inzwischen ist das Buildmanagement so wichtig und aufwendig geworden, dass daraus ein eigener Berufszweig entstanden ist. Buildmanager sind daher „täglich“ damit beschäftigt, diesen Prozess zu (re)definieren und durchzuführen.

### Ein Leben ohne Buildmanagement?

Was würde passieren, wenn in einem größeren Unternehmen kein Buildmanagement eingesetzt werden würde?

- Das Endprodukt würde nicht laufen und niemand würde die Ursache dafür kennen.
- Details würden einfach vergessen werden (z. B. das manuelle Ausführen des RMI Compilers, das ein Fehlen der Stubs zur Folge hat).
- Das Wissen über den Erstellungsprozess der Software wäre nicht zentral vorhanden und dokumentiert, sondern befände sich beispielsweise in den Köpfen der Mitarbeiter (die garantiert krank werden), auf Notizzetteln oder in diversen Dateien.
- Die Vergabe der Versionsnummer (z. B. 1.2.24) würde von Hand vorgenommen werden.
- Die Prozesse würden zu viel Zeit in Anspruch nehmen. (Tests oder Generierung der Dokumentation müssten manuell angestoßen werden, etc.)

Größere Unternehmen  
ohne Buildmanagement?

## 1.2 Continuous Integration

MARTIN FOWLER und KENT BECK haben die Begriffe des Continuous Integration geprägt. Sie werden fast immer im Zusammenhang mit **Extreme Programming** (siehe Lerneinheit VOR - Vorgehensmodelle / agile Modelle) genannt.

Das wohl bekannteste Dokument zur Continuous Integration stammt von Martin Fowler und ist am 1. Mai 2006 auf den neuesten Stand gebracht worden:

<http://www.martinfowler.com>.

Obiges Dokument enthält folgendes Statement:



Definition

### Continuous Integration

*„Continuous Integration ist eine Vorgehensweise, Software zu entwickeln. Die (tägliche) Arbeit eines jeden Entwicklers steuert dazu bei, dass mehrmals täglich integriert werden kann. Diese Integrationsbuilds werden von dem automatisierten Build überprüft (z. B. durch Tests), um Fehler so früh wie möglich festzustellen. Viele Teams haben berichtet, dass dieses Vorgehen die Integrationsprobleme bedeutend reduziert. Dadurch kann zusammenhängende Software schneller erstellt werden.“*

(Aus dem Englischen übersetzt vom Modulautor)

Es gibt viele Darstellungsweisen für dieses Vorgehen. Die wichtigsten Elemente sind daher exemplarisch wie folgt dargestellt:



Abb.: Continuous Integration

### 1.3 Continuous Integration im Detail

Im Folgenden werden die wesentlichen Elemente des Continuous Integration-Vorgehens (analog zu dem referenzierten Dokument [www http://www.martinfowler.com](http://www.martinfowler.com)) dargestellt.

Ein wesentliches Element dieses Prozesses ist das Buildmanagement. Ohne dies sind viele der folgenden Punkte nicht möglich.

Repository

#### ➡ **Verwendung von Repository / Versionskontrollsystem für die Quellen**

Wie in der Lerneinheit SVN für Versionskontrollsysteme erläutert, liefern Systeme wie CVS oder Subversion einen „sicheren Boden“ für alle Veränderungen im Projekt. Diese gewährleisten unter anderem ein Zurückspringen, die Verwaltung mehrerer Versionen, oder sogar das parallele Bearbeiten gleicher Dateien etc.

Automatisierung

#### ➡ **Automatisierung des Buildprozesses**

Aus den vorher genannten Gründen (Performance, Nachvollziehbarkeit, zentrale Dokumentationsstelle, Eliminierung von Fehlerquellen, etc.) sollte der Buildprozess mit Werkzeugen wie Ant, Nant oder MSBuild automatisiert werden.

Testen

#### ➡ **Testen des Buildprozesses**

Das Ergebnis und die Teile des Softwareproduktes sollten als Teil des Buildprozesses automatisiert getestet werden. D. h. es sollten die Tests der \*Unit\* Familie verwendet werden, die dazu dienen „self-testing“-Code zu schreiben. Fowler verweist auf Tools, die in Richtung Akzeptanztests gehen wie beispielsweise:

- FIT <http://fit.c2.com/>
- Selenium <http://www.seleniumhq.org/>
- Sahi <http://sahi.sourceforge.net/>
- Watir <http://wtr.rubyforge.org/>
- FITnesse <http://fitnesse.org/>

Täglich einpflegen

#### ➡ **Jeder trägt täglich etwas zum Projekt bei**

Entwickler sollten neuen Code täglich einchecken, sofern er getestet ist. Dieses schnelle Einstellen von korrektem Code zeigt sofort, ob es Integrationsprobleme gibt (die sich nach dem Build zeigen). Durch dieses Vorgehen wird die sofortige Kommunikation der Entwickler untereinander ermöglicht, wodurch Probleme in späteren Phasen vermieden werden können.

Um weitere Ideen und Informationen zu erhalten, lesen Sie bitte folgende Seite:

[www http://www.martinfowler.com](http://www.martinfowler.com)

## 1.4 Build Varianten

Die Grundidee eines schnellen automatisierten Builds ist es, Integrationsprobleme schnellstmöglich zu erkennen und zu beheben. Eine XP-Richtlinie besagt, dass ein Build ganz einfach und von jedem Entwickler angestoßen werden kann; am besten **nach dem Check-In von neuem Code**. Dieses manuell angestoßene Build dient auch als Test für diesen neuen Code, da diese in Build Tests ausgeführt werden. Diese Builds sollten **nicht länger als 10 Minuten** dauern, was in der Regel (mit kompilieren, testen, packen, kopieren, etc.) auch realisierbar ist.

### Stufenweises Builden

Üblicherweise ist der Buildprozess in **Stufen** unterteilt, so genannte Stages. Ein **stage build** ist daher ein Prozess (eine build pipeline), bei dem die Stufen des Buildprozesses nacheinander durchlaufen werden. Man spricht hierbei auch von einem **Stage-Server**, auf dem „Committed“ und „Gebildet“ wird.

#### 1. commit-build

Wenn neuer Code integriert wird, wird der erste Build als **commit-build** bezeichnet und ausgeführt. Hierbei wird sichergestellt, dass ggf. Tests für diesen Code laufen und der Build für das Hauptprogramm (die sog. **mainline**) erstellt werden kann.

#### 2. secondary build

Nachfolgend kommen weitere Builds wie **secondary builds** hinzu. Diese sind in der Lage, intensivere Tests durchzuführen und das Produkt gezielter zu vollenden. D. h. beispielsweise ein großes war-Archiv erstellen („web-archiv“ aus J2EE), welches dann auf einem Server wie JBoss oder Weblogic ausgeführt werden kann. Kann auch dieser Vorgang - der im Prozess weiter hinten ansteht - problemlos durchgeführt werden, so können diese Builds als erfolgreich abgeschlossen betrachtet werden.

Dieser Prozess ist in folgender Grafik dargestellt:

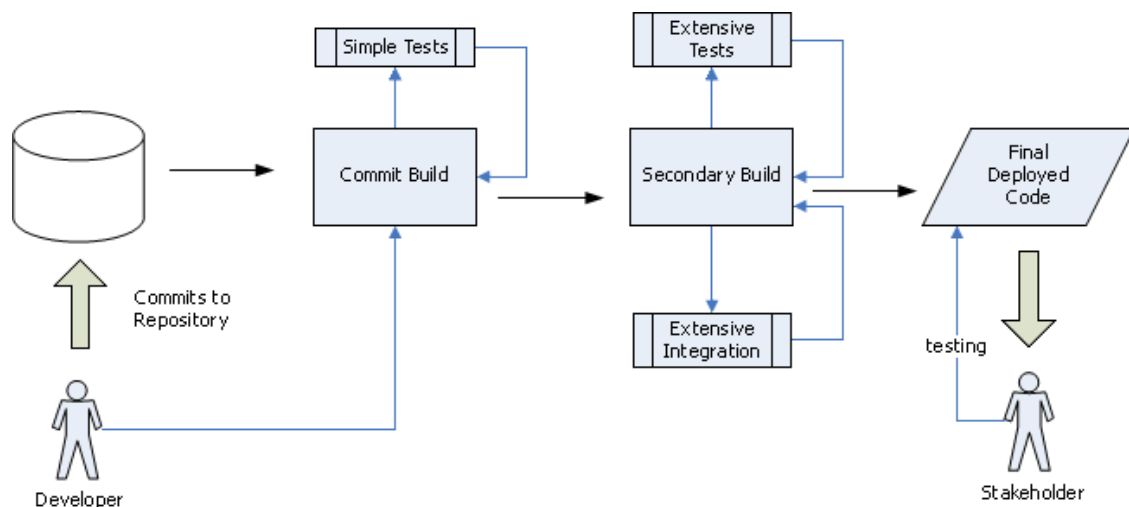


Abb.: Stufen des Buildprozess

## 1.5 Konfigurationsmanagement



Definition

**Die ANSI definiert das Konfigurationsmanagement wie folgt:**

*„Configuration Management [...] is a management process for establishing and maintaining consistency of a product's performance, its functional and physical attributes, with its requirements, design and operational information, throughout its life.“*

Konfigurationsmanagement bezeichnet eine Management-**Vorgehensweise**, bei dem die **Konfigurationseinheiten** / Attribute des Produktes über den gesamten Produktlebenszyklus **verwaltet und definiert** werden. Gemeint sind hier sowohl Hardware, als auch Software und Dienstleistungen. Bestimmte Produkte beispielsweise können nur im Zusammenhang mit einer konkreten Dienstleistung funktionieren.

Konfigurationsteile  
ISO 10007

Die ISO 10007 kennt daher folgende Teilaspekte des Konfigurationsmanagements:

- KMO - Konfigurationsmanagementorganisation und -planung
- KI - Konfigurationsidentifizierung
- KÜ - Konfigurationsüberwachung
- KB - Konfigurationsbuchführung
- KA - Konfigurationsaudit

Das Buildmanagement ist daher in gewisser Weise ein Teil des Konfigurationsmanagements, der sich auf die Software bezieht. Im Buildmanagementskript wird zentral definiert und dokumentiert, wie die Software zu konfigurieren ist. D. h. beispielsweise für welche Plattform oder Anforderungen, welcher Code (bzw. Tests oder Dokumentation) wie zu erstellen und auf dem Server zu installieren ist.

Manchmal wird das Buildmanagement (fälschlicherweise) mit dem Konfigurationsmanagement gleichgesetzt. Dies mag in kleinen Projekten durchaus angemessen sein, ist jedoch bei umfangreicheren Konzepten (z. B. Toll Collect) nicht mehr passend.



## 2 Historie und Tools

Die Historie des Buildmanagements ist eng mit dem Tool **make** verknüpft, das auch heute noch eine bedeutende Rolle spielt.

Seine Ursprünge hat dieses Tool in **UNIX**-Systemen, auf denen Code kompiliert werden musste. Das Kompilieren von Hand war eine mühsame Aufgabe. Oftmals wurden nur wenige Dateien geändert, die erst mühsam identifiziert und dann von Hand übersetzt werden mussten. Selbst mit Kommandos wie „`cc *.c`“ (`cc` ist der Compiler und `*.c` referenziert alle C-Dateien in diesem Verzeichnis), die einfach einzugeben sind, kann der Kompilierungsprozess zu lange dauern, insofern viele Dateien erfasst wurden, aber nur wenige davon geändert werden. Und was, wenn der Sourcecode auf mehrere Verzeichnisse verteilt ist? Es gibt 1000 Gründe, warum Handarbeit sinnlos ist.

Steward Feldman erkannte diesen Missstand und erfand ein Werkzeug, welches diese Vorgänge automatisiert. Die Vorgänge werden einfach in einem **Buildfile**, das wie ein **Scriptfile** aussieht, notiert. Entscheidend waren aber damals schon weitere Features wie **Dependencies** (Abhängigkeiten).



Definition

### Dependencies

Dependencies sind Abhängigkeiten zwischen Code oder Modulen. Beispielsweise kann es sein, dass die Datei A geändert wurde, und anschließend nicht nur A, sondern auch B neu kompiliert werden muss.

Diese Abhängigkeiten können von modernen Werkzeugen erkannt oder auch in Buildfiles direkt notiert / angegeben werden.

Im Folgenden sei das klassische Beispiel für ein Makefile dargestellt:



Quellcode

### makefile

```
helloworld: helloworld.o
    cc -o $@ $<

helloworld.o: helloworld.c
    cc -c -o $@ $<

clean:
    -rm -f helloworld helloworld.o
```

Auch heutzutage wird `make` weitergepflegt und hat in der GNU Foundation seinen Platz gefunden. Durch den guten Support dieses Tools ist GNU `make` <http://www.gnu.org> auch heute noch von großer Bedeutung.

## 2.1 Tools

Einige der wichtigsten make-Derivate sind in der folgenden Abbildung dargestellt:

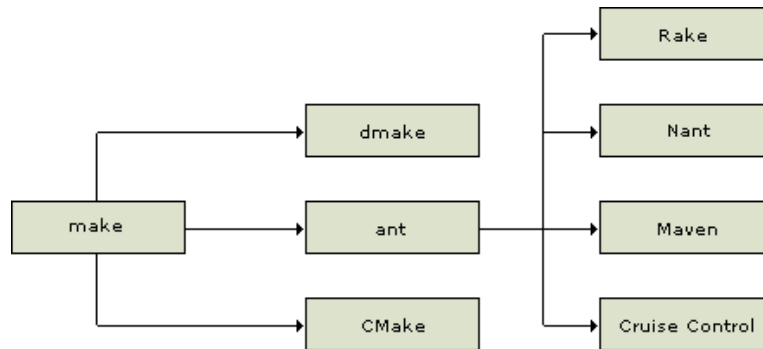


Abb.: Wichtige make-Derivate

Nach dem Erfolg von make entstanden einige Varianten, die entweder wie CMake [www.cmake.org](http://www.cmake.org) auf Plattformunabhängigkeit setzen, oder wie dmake (distributed make) speziell auf eine verteilte Ausführung getrimmt sind und von SUN für Solaris und OpenOffice entwickelt wurden.

Einer der Wegbereiter für ein modernes Buildmanagement ist jedoch Ant <http://ant.apache.org>. Der Durchbruch von Ant liegt in XML begründet. Die **Extensible Markup Language** wurde zwischen 1996 und 1999 standardisiert. Mit XML stand nur eine Beschreibung zur Verfügung, die von Menschen schwer, von Maschinen jedoch besonders gut gelesen werden kann und von Generatoren erzeugt wird. Anders als bei make, liegt hier eine DTD oder ein Schema vor, die die Struktur eines XML-Dokuments genau prüft. Die Entwicklung von Editoren / Eingabehilfen in der IDE ist nun einfach möglich.



Durch die Entwicklungen von XML und Ant wurde ein Quantensprung im Buildmanagement vollzogen. Ant selbst ist schon lange in Version 1.6 stabil und wird in nahezu allen relevanten und größeren Java-Projekten eingesetzt.

Nach diesem Erfolg entwickelten auch andere Firmen und Communities ähnliche Werkzeuge. Nant <http://nant.sourceforge.net> für die .Net Umgebung und Rake <http://rake.rubyforge.org> für Ruby sind hier nur zwei Beispiele.



Schließlich gibt es fortgeschrittene Werkzeuge, die auf Ant aufsetzen (oder es ersetzen) und insbesondere den Continuous Integration Zyklus fördern. Maven <http://maven.apache.org> und Cruise Control <http://cruisecontrol.sourceforge.net> sind die bekanntesten Werkzeuge. Nähere Beschreibungen sind auch in [\[ EB03 \]](#) zu finden.

### 3 Ant Basics

Ant ist mit seinen Derivaten (NAnt, Rake, etc.) bei modernen OO-Projekten sicherlich das meistgenutzte Tool. Alle modernen IDEs wie Eclipse, NetBeans, JBuilder, IntelliJ, etc. haben Ant integriert.

Hands-On

Ziel der folgenden Kapitel ist es daher, nicht nur theoretisches Wissen zu vermitteln, sondern Ant so einzuführen, dass ein leichtes Hands-On möglich ist.

Der Leser sei daher aufgerufen, möglichst einmal alles selbst auszuprobieren!

### 3.1 Ant Start

Ant wurde als Teil von Tomcat bereits im letzten Jahrhundert entwickelt und im Januar **2000** an Apache übergeben. Nach etwas schnelleren Versionssprüngen ist Ant 1.6 jetzt bereits seit 2003 released. Ant 1.6.5 ist seit Juni 2005 stabil. Wie bei JUnit trägt auch die Stabilität von Ant dazu bei, Ant als Standard einzusetzen.

XML Vorteile

Die Idee von Ant war, ein komplett neues Buildmanagement-Werkzeug aufzusetzen, welches keine skriptsprachenbasierte Vergangenheit hat. Auf der anderen Seite sollte eine starke Plattformunabhängigkeit gefördert werden. Und, wie bereits erwähnt, ist die ideale Maschinenlesbarkeit von XML in vielerlei Hinsicht fördernd für die Erstellung von Tools und Wizards auf der Basis von Ant.

Ant erweitern

Bisher konnten makefiles nicht einfach erweitert werden. Unter Ant ist das extrem einfach, da unter jede beliebige Klasse in Ant integriert werden kann, sofern sie ein paar kleine Randbedingungen erfüllt.

Ant ist unter <http://ant.apache.org> zu finden.

Insbesondere das Manual ist ausgereift und enthält sehr viele Beispiele zu den Tätigkeiten (genannt Tasks), die man ausführen möchte.



Ant kann mit oder ohne IDE ausgeführt werden. Speziell für große Builds (evtl. sogar über Nacht) ist die direkte Ausführung von Ant-Skripten nötig.

Unter UNIX und Windows müssen nach dem Download und der Installation lediglich ein paar Skriptzeilen ausgeführt werden, um Buildskripte starten zu können:



Quellcode

#### UNIX (advanced shells)

```
export ANT_HOME=/usr/local/ant
export JAVA_HOME=/usr/local/jdk-1.2.2
export PATH=${PATH}:${ANT_HOME}/bin
```

#### Windows

```
set ANT_HOME=c:\ant
set JAVA_HOME=c:\jdk1.5
set PATH=%PATH%;%ANT_HOME%\bin
```

build.xml

Danach kann **Ant** in der Shell aufgerufen werden. Es wird nach einem **build.xml** file gesucht und dieses ausgeführt.

Bevor nun weiter fleißig Skripte geschrieben werden, hier noch ein Blick auf die bekanntesten ANT-Bücher. Besonders empfehlenswert ist das Buch von Erik Hatcher, welches zwar nicht auf der neuesten Ant-Version basiert, jedoch vom Inhalt her unübertroffen ist und weit über den Tellerrand hinaus schaut.

[ [Ha02](#) ]

[ [Ho05](#) ] - Steve Holzner: „Ant - The Definitive Guide. Complete Build Management for Java“

[ [ES06](#) ] - Stefan Edlich, Jörg Staudmeyer: „Ant - kurz & gut“

[ [Ma05](#) ] - Bernd Matzke: „Ant. Eine praktische Einführung in das Java-Build-Tool“

### 3.2 Ant: Erste Schritte

Die folgenden drei Definitionen sollen den Sprachgebrauch in den nachfolgenden Beispielen erleichtern:



Definition

#### Targets

Targets beschreiben eine globale Aufgabe, die es zu erledigen gilt. Kurz zusammen gefasst:

- Ich möchte das Programm übersetzen, packen und starten.
- Ich möchte die Javadoc erstellen, zippen und per E-Mail versenden.

Dies sind Beispiele für größere Aufgaben, die sich aus kleineren zusammen setzen, die es zu definieren und zu erledigen gilt.



Definition

#### Tasks

Die kleineren Aufgaben aus denen ein Target besteht, heißen Tasks. Ein Task kann beispielsweise das Kompilieren der Sourcen, das Verpacken der class-Files oder das Kopieren des Ergebnisses sein.

Wichtig ist, das ein Task immer genau einen Schritt beschreibt, der in einem Kommando auf der Shell von Hand ausgeführt werden könnte.



Definition

#### Dependencies

Als Dependencies bezeichnet man Abhängigkeiten, die selbst spezifiziert werden können. Die übliche Definition ist, dass man beispielsweise sagt: Bevor das Programm ausgeführt werden kann, muss es übersetzt werden. D. h. **vor** dem „run“-Target sollte das „compile“-Target ausgeführt werden.

Dependencies beschreiben den Ablauf / Workflow für einen Build.

Ein Buildfile kann folgende Targets beinhalten:

```

A
B → A
C → B
X
Y

```

Der Anwender würde hier z. B. C aufrufen. Jedoch müsste vorher das Target B und noch davor A ausgeführt werden. Der Anwender kann genauso gut das Target X und Y aufrufen, welches nicht von einer anderen Aktion abhängig ist.

### 3.3 Praxisbeispiel Hello World (Eclipse)



Beispiel

#### „Hello World“ unter Eclipse

Im Folgenden wollen wir ein einfaches „Hello World“ unter Eclipse ausführen. Dieses Vorgehen funktioniert mit anderen IDEs fast genauso.

#### Schritt 1: Anlegen einer Datei mit dem Namen **build.xml** in einem Projekt

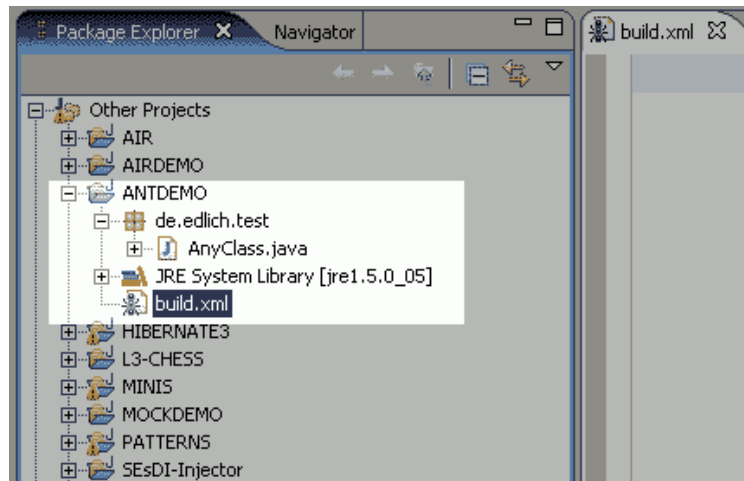


Abb.: Anlegen der Datei build.xml

Man sieht anhand des Icons in Form einer Ameise, das die IDE dieses File als Ant-File erkennt.



Hinweis

Vorsicht: Wenn „build.xml“ falsch geschrieben wird dann wird dieses File nicht gefunden und der Aufruf von Ant erzeugt eine Fehlermeldung!

#### Schritt 2: Öffnen der Ant-View und Übergabe der Datei build.xml aus dem Projekt

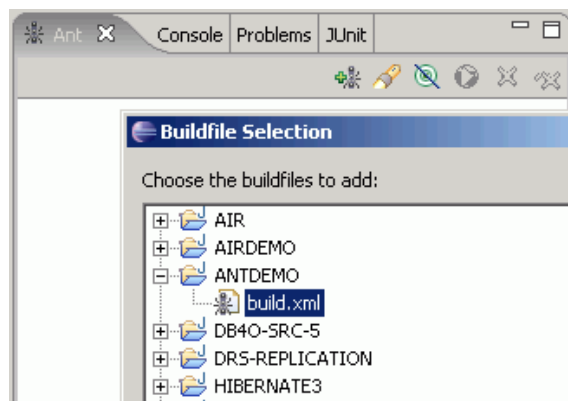


Abb.: Übergabe der Datei build.xml an Ant-View

Wahrscheinlich wird dann Initial die Ant-View einen Fehler anzeigen, da die Datei keinen Inhalt hat, nicht korrekt geparkt und dargestellt werden kann. Das macht jedoch nichts, da wir jetzt die Inhalte hinzufügen.

#### Schritt 3: Einfügen von Inhalt in das geöffnete build.xml

Noch haben wir keine Vorstellung davon, wie die Inhalte für die build.xml-Datei aussehen könnten.

Hier hilft der intelligente Ant-Editor der mit der Taste **STRG (CTRL)** aktiviert wird und umfassende Hilfe anbietet. (Der Ant-Editor wurde anfänglich von Alf Schiefelbein und dem Modulautor als Plug-In für Eclipse 2.\* entwickelt. Das Plug-In wurde später von Erich Gamma übernommen, integriert und weiterentwickelt.)

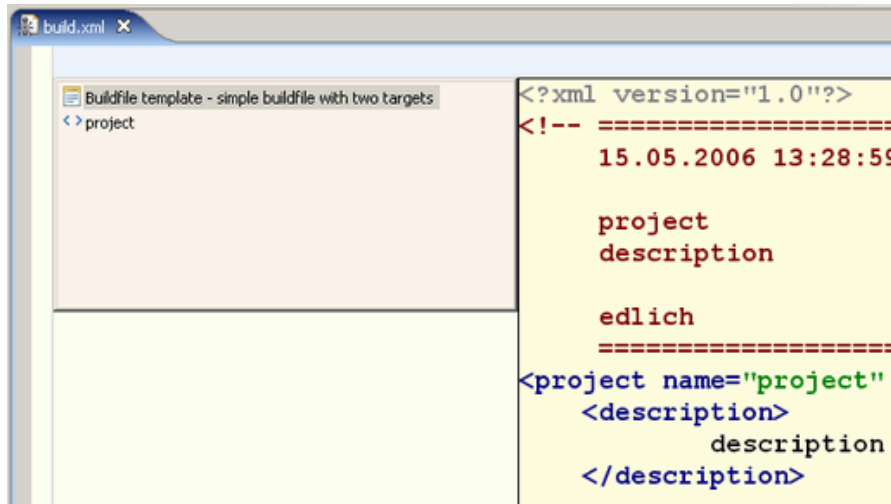


Abb.: Inhalte in die  
Datei build.xml einfügen

### 3.4 Praxisbeispiel Hello World (Fortsetzung)

#### Schritt 4: Eingabe eines einfachen Hello World-Codes

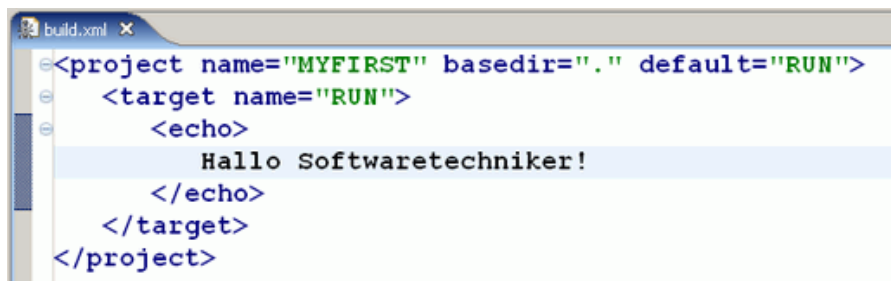


Abb.: Hello World  
Eingabe

Hierbei muss nicht alles von Hand geschrieben werden. Die **STRG-SPACE**-Hilfe kann dabei viel abnehmen und gleichzeitig erklären.

#### Schritt 5 : Ausführung des gewünschten Targets

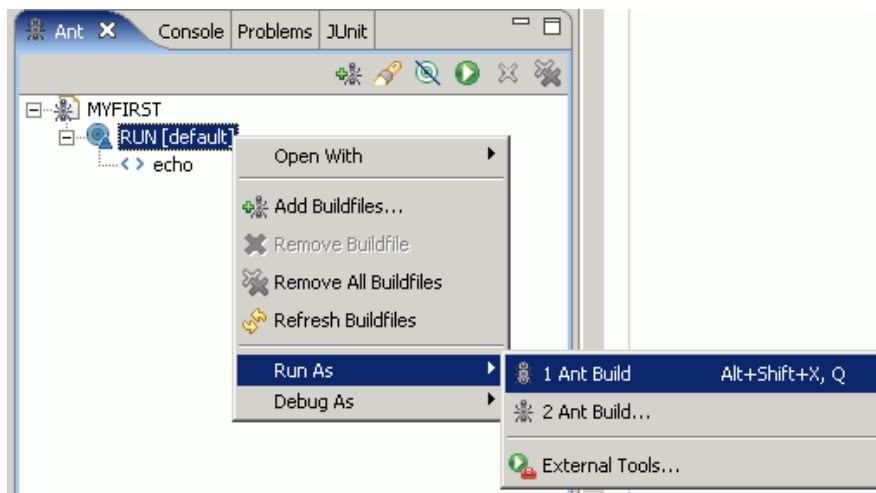
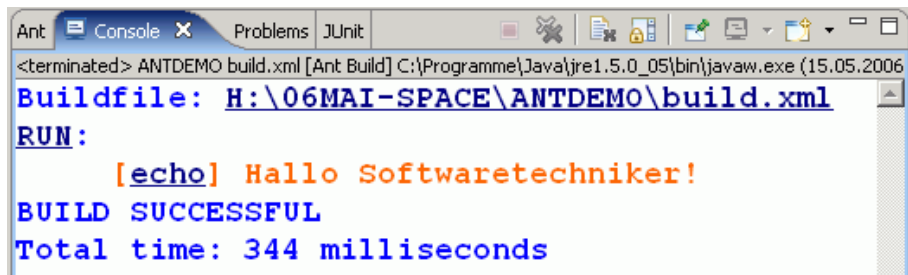


Abb.: Target ausführen

Wenn mehrere Targets enthalten wären, würden diese ebenfalls dargestellt. Alternativ kann der Start über grünen Button erfolgen. Im „Ant build“-Menü können sehr viele Voreinstellungen für Ant getroffen werden.

**Schritt 6:** Bewunderung des Ergebnisses


```

<terminated> ANTDEMO build.xml [Ant Build] C:\Programme\Java\jre1.5.0_05\bin\javaw.exe (15.05.2006
Buildfile: H:\06MAI-SPACE\ANTDEMO\build.xml
RUN:
    [echo] Hallo Softwaretechniker!
BUILD SUCCESSFUL
Total time: 344 milliseconds

```

Abb.: Ergebnis

Zugegebenermaßen, bisher haben wir kaum etwas geleistet / gebuildet. Etwas zum Laufen zu bekommen ist jedoch auch schon ein schöner Erfolg.

**Schritt 7:** Starten eines Buildfiles, das etwas leistet

Quellcode

**Simple build.xml**

```

<project default="run">

    <target name="compile">
        <javac srcdir="." />
    </target>

    <target name="jar" depends="compile">
        <jar destfile="hello.jar" basedir="." includes="**/*.class" />
    </target>

    <target name="run" depends="jar">
        <java classname="hello" classpath="hello.jar" fork="true" />
    </target>

</project>

```

**3.5 Sonstiges und Properties****Was ist das Starttarget?**

Default Target nötig

Das Starttarget kann individuell ausgewählt werden. Üblicherweise enthalten Build-Skripte mehrere Targets. Der Buildmanager wählt dann aus, welches er starten will: Also z. B. **clean**, **run** oder **compile**, etc.

Wird Ant direkt ohne Parameter von der Konsole aus aufgerufen oder das Buildfile per Doppelklick im Fenster gestartet, so wird standardmäßig der default-Task ausgewählt, der seit Ant 1.6.0 immer vorhanden sein muss:

```
<project default="run">
```

Es ist üblich alle drei Parameter anzugeben. Ein korrekter Aufruf wäre daher:

```
<project basedir="." name="MyFirstBuildfile" default="run">
```

**Können Targets mehrere Male aufgerufen werden?**

Jedes Target nur einmal aufrufen

Nein. Jedes Target wird nur einmal aufgerufen. Der Einbau von programmiersprachlichen Schleifen war nie vorgesehen. Es gibt jedoch so etwas ähnliches wie **if / else** in Ant. Ein Beispiel ist unter „Eigene Tasks“ zu finden.



Properties sind  
Variablen unter Ant

`${myVar}`



Quellcode

## Properties

Properties sind Variablen. Ant kennt jedoch nur den Typ **string**. Diese sind unter anderem wichtig, damit Redundanzen im Buildfile vermieden werden können.

Properties werden mit dem Task **property** definiert und mit `${ }` referenziert.

Hier ein Beispiel für deren Verwendung:

### Properties-Beispiel

```
<project name="TestProperties" default="run" basedir=".">
  <property name="sourceDir" value="source"/>
  <property name="deployDir" value="c:/tmp"/>
  ...
  <property name="wlsHome" value="c:/bea/wlserver9.1"/>
  <property name="domainName" value="myDomain"/>
  <property name="sourceDir" value="source"/>
  <property name="targetDir"
    value="${wlsHome}/config/${domainName}/applications"

  <mkdir dir="${sourceDir}"/>
```

Wie das Beispiel zeigt, sollten zunächst einmal alle Variablen definiert werden. Danach werden diese im Code referenziert. Namen oder Pfade sollten nie doppelt im Code angegeben werden. In großen Projekten wird sogar im Buildfile initial Quellcode aus einem anderen Buildfile importiert, in dem lediglich Definitionen stehen, wie z. B.:



Quellcode

### Import von anderem Buildfile.xml-Code

```
<project name="import1" basedir="." default="...">
  <import file="${importPath}/properties.xml"/>
</project>
```

Daher sollten Buildfiles sobald sie zu groß sind, aufgeteilt werden.

Ein besseres Beispiel für den Import von projektspezifischen Properties ist jedoch die Fileoption von **property** direkt zu verwenden:



Quellcode

### Quellcode

```
<project name="import2" basedir="." default="...">
  <property file="myProperties.prop"/>
</project>
```

Oftmals ist die Angabe einer Location bei Properties sinnvoller, als die eines Values. Beispielsweise wenn es sich um einen Pfad handelt. Location wandelt den Pfad intern in einen absoluten um. D. h. man spart sich hier viel Schreibarbeit und vermeidet damit auch potenzielle Fehlerquellen.

## 4 Tasks

Tasks sind die wichtigsten Grundbausteine eines jeden Buildfiles. Früher waren dies die grundlegenden Operationen in Makefiles, die man unter UNIX ausführen kann. Dazu gehört beispielsweise:

- Aufrufen des Compilers
- Erstellen eines Verzeichnisses
- Kopieren von Dateien

Unter Ant sind dies vordefinierte Kommandos, die eine Vielzahl von nützlichen Aktivitäten abbilden, die beim Buildmanagement benötigt werden. Diese Tasks sind in der Version 1.6.5 in etwa **85** sogenannte **core Tasks** und ca. **60 optional Tasks** unterteilt.

Unter Core findet man die wichtigsten produktunabhängigen Aufgaben. Die optionalen sind meist selten benutzte Tasks, die häufig produktabhängig sind. Nachfolgend sind ein paar der wichtigsten Beispiele gelistet (fett markierte Tasks sind besonders wichtig):



Beispiel

### Core Tasks (Auszug)

**BuildNumber**, Checksum, Chmod, Concat, **Copy**, **Cvs**, **Delete**, **Echo**, Exec, Fail, Filer, Get, **Import**, Input, **Jar**, **Java**, **Javac**, **Javadoc**, Mail, **Mkdir**, **Move**, Nice, Parallel, **Property**, Rmic, Sleep, Sql, Tar, Taskdef, **Tempfile**, Touch, **TStamp**, Typedef, Unjar, Untar, Unzip, Waitfor, War, Xslt, **Zip**



Beispiel

### Optional Tasks (Auszug)

Cab, Chgrp, Chown, Depend, EJB Tasks, Echoproperties, FTP, IContract, Image, JavaCC, Javah, JspC, JDepend, JProbe Coverage, **JUnit**, Perforce Tasks, PvcS, Rpm, ServerDeploy, **Script**, Sound, Splash, Telnnet, Microsoft Visual Source Safe Tasks, XMLValidate

Abschließend müssen lediglich noch Targets im Projekt erstellt und diese mit den vorliegenden Tasks gefüllt werden. Dabei sind die einzelnen Parameter der Tasks wichtig. Diese lassen sich leicht in der IDE-Hilfe (z. B. CTRL-SPACE), im Manual unter <http://ant.apache.org> oder in den dort enthaltenen Beispielen ablesen.

## 4.1 Tasks Beispiele

Im Folgenden sind einige Beispiele für den Aufruf typischer Tasks gelistet.



Quellcode

### copy & delete

```
<copy file="myWeb.xml" tofile="root/WEB-INF/web.xml"/>
<copy file="Web.xml" todir="root/WEB-INF"/>

<delete file="myFile.txt"/>

<delete dir="myproject/temp"/>
```



Quellcode

### javac

```
<javac srcdir="${src}"
  destdir="${build}"
  classpath="mytools.jar"
  debug="on"
  source="1.5"
/>
```

Das oben stehende Beispiel kompiliert alle Sourcen in `${src}` und schreibt das Ergebnis \*.class in `${build}`.



Quellcode

### zip (ähnlich wie jar)

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

## 4.2 Patternsets, Filesets & Co.

Komplexere Themen

In Ant gibt es nur zwei komplexe und schwierige Themen:

1. Die korrekte Angabe der Parameter / Attribute von Tasks  
(Hilfestellungen gibt es unter [www http://ant.apache.org](http://ant.apache.org))
2. Definition von Mengen  
Unter Ant müssen oft Mengen definiert werden, z. B. eine Menge von Dateien, eine Menge von Verzeichnissen, eine Menge von regulären Ausdrücken u.s.w.

Zur Definition von Mengen sind folgende Strukturen hilfreich:

### Patternsets

Patternsets dienen dazu eine **Menge von Dateien** zu referenzieren, diese mit einer ID zu versehen, um sie später wiederverwenden zu können. Ein Beispiel:



Quellcode

#### Patternset

```
<patternset id="javaFiles">
  <include name="**/*.java"/>
  ...
</patternset>
```

### Filesets

Filesets dienen dazu eine Menge von Dateien zu definieren - sie referenzieren auf Patterns von Patternsets. Hier zuerst mit Referenz auf das vorherige Pattern:



Quellcode

#### Fileset

```
<fileset id="arolproject/sources">
  <patternset refid="JavaFiles"/>
</fileset>
```

Schwierig sind dabei u.a. die vielen Regeln und Beispiele für die inneren Attribute:

#### Quellcode

```
<fileset dir="WEB-INF/classes">
  <include name="**/*.class"/>
  <exclude name="de/edlich/demoapp/*.class"/>
  <exclude name="org/apache/jsp/*.class"/>
</fileset>
```

Das Attribut `dir` definiert das Root-Verzeichnis, von dem aus das Fileset definiert wird.

Weiterhin sind **includes** (Komma oder Leerzeichen separierte Liste mit Dateien, die inkludiert werden müssen) und **excludes** (Komma oder Leerzeichen separierte Liste mit Dateien, die exkludiert werden müssen) möglich.

Es gibt weitere Strukturen, die im Ant Manual unter Concepts and Types

[www http://ant.apache.org](http://ant.apache.org) genauer beschrieben sind:

- Filelists: Dateien ohne Wildcards
- Dirsets: Wie Filelists, jedoch für Ordner
- Patternsets: Wie Filesets jedoch mit einer Referenz-ID
- Path-Structures: Für PATH und CLASSPATH
- File Mappers: Regeln für die Endungen bei Mengentransformationen (z. B. bei Dateien)

## 5 Fallbeispiel



Beispiel

### Entwickeln eines Buildfiles

In diesem Fallbeispiel soll ein umfangreiches Buildfile entwickelt werden, dass so viele Targets wie möglich enthält.



Interessant ist, dass man Ant sehr gut **für andere Dinge außer Softwarebuilds** einsetzen kann. Der Modulautor setzt Ant für viele Dinge im täglichen Leben ein: Einkaufen, Wäsche waschen, Kinder erziehen, ...

Nein, Scherz beiseite: Obwohl es leicht möglich ist, derartige Tasks zu definieren, wird es wohl noch einige Jahrzehnte dauern, bis dafür auch die nötigen Tasks zu Verfügung stehen.

Backup-Tool mit Ant

Dennoch wird im Folgenden ein Beispiel aufgeführt, das zeigt, dass die **Anwendungsgebiete für Ant viel breiter sind, als gedacht**. Der Modulautor besitzt einen USB-Stick, der ab und zu gesichert werden sollte. Was liegt näher als ein kleines Ant-Skript zu schreiben, welches diese lästige Aufgabe übernimmt. Das folgende Skript packt daher Verzeichnisse auf dem USB-Stick als ZIP-Datei und speichert diese auf der Festplatte in einem Directory, welches das aktuelle Datum enthält. Dies ist oft einfacher und praktischer als die besten **Backup-Tools**:

### Quellcode

```
<project name="HANDYSAVE" basedir="G:/" default="save">
  <target name="save">
    <echo message="Starting to save your Laptop..." />
    <echo message="Bitte Eclipse schliessen!" />
    <echo message="> Setting the date ..." />

    <tstamp/>

    <echo message="> Creating the target dir on C: ..." />
    <mkdir dir="C:/HANDYDRIVE/${DSTAMP}" />
    <echo message="> Created the target dir on C: ..." />

    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/BERUF.zip" basedir="G:/BERUF" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/BUECHER.zip" basedir="G:/BUECHER" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/CENTRAL-3.0-SPACE.zip" basedir="G:/CENTRAL-3.0-SPACE" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/DB40.zip" basedir="G:/DB40" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/FORSCH.zip" basedir="G:/FORSCH" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/LITERATUR.zip" basedir="G:/LITERATUR" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/PRIV.zip" basedir="G:/PRIV" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/WEB_PROJEKTE.zip" basedir="G:/WEB_PROJEKTE" />
    <zip destfile="C:/HANDYDRIVE/${DSTAMP}/XHACK-libs.zip" basedir="G:/XHACK/libs" />

    <echo message="Successfully finished saving your USB-Stick!" />


  </target>
</project>
```

Probieren Sie dieses Beispiel einfach mal aus und passen sie es auf die eigenen Bedürfnisse an.

## 6 Buildmanagement für Fortgeschrittene

In diesem Kapitel werden ein paar fortgeschrittene Funktionen - Advanced Features - vorgestellt, die für das Verständnis der Konzepte des Buildmanagements relevant sind.


Ant bietet bei weitem mehr Möglichkeiten, als hier angesprochen werden. Allein die später vorgestellten Zugriffe auf Skriptsprachen sind extrem hilfreich.

Beispielsweise können Entwickler auch selbst von Ant profitieren und Ant Tasks einfach unter Java  <http://ant.apache.org> verwenden.

### 6.1 Was Profis verwenden

#### Checksum

MD5 in einer Zeile

Für viele Distributionen ist es nötig eine Checksumme zu bilden. Beispielsweise enthält der Download der Tomcat Distribution  <http://tomcat.apache.org> immer eine **MD5** Checksumme. Mit diesem Task kann daher automatisch im Build z. B. eine MD5 Checksumme gebildet werden. Die aktuelle 5.5.17 Distribution (Bitte die 17 für später merken!) hat die Checksumme **87ce0084ff43d7d97120fb3af0cfe4ff**. MD5 gibt also eine 128 Bit Zahl an, gegen die der Download geprüft werden kann.

Der Aufruf ist entsprechend trivial:

```
<checksum file="foo.bar"/>
```

#### Temporäre Dateien

Task tempfile

Im Build kommt es immer wieder vor, dass man eine temporäre Datei erstellen muss, die nur Daten aufnimmt, damit sie anschließend weiterverarbeitet werden kann. Die Datei selbst spielt keine Rolle und kann nach dem Build sofort wieder gelöscht werden. Hierfür ist der Task **tempfile** geeignet. Ein Beispiel:



Quellcode

##### Temporäres File

```
<project name="TEST" basedir="D:" default="all">
  <target name="all">
    <tempfile property="temp.file" destdir="D:\tmp\"/>
    <touch file="${temp.file}" />
  </target>
</project>
```

Probieren Sie dieses Buildfile einfach mal aus. Das Ergebnis ist z. B.:

```
[touch] Creating D:\tmp\null-978523213
```

## 6.2 Buildnummern vergeben

Ein wichtiger Aspekt beim Erstellen und Packen von Anwendungen ist die Angabe des aktuellen Releases. Sicherlich ist jedem schon aufgefallen, dass viele Programme eine interessante oder verschlüsselte Art haben, das Release im Dateinamen auszudrücken.

Was bedeutet es überhaupt, wenn Tomcat in der Version 5.5.17 heruntergeladen werden kann?

Hier gibt es einige Vorgehensweisen, die es sich anzueignen lohnt:



Quellcode

### Ausführliches Versioning

```
Version.MajorRelease.FixPack BuildNumber als
1.1.3 Build 42

oder

Version.MajorRelease.MinorRelease.BuildNumber als
1.1.3.42
```

oder etwas schmaler:

### Kleinere Versionierung

```
Version.MajorRelease.BuildNumber als
1.0.21
```

Gut zu sehen ist ein derartiges Vorgehen auch bei Eclipse. Dort werden mehrere Endstufen eingebracht:

- Eclipse wandert zuerst in kaum sichtbaren sub-Versionen wie 3.1.\*  
Diese werden dem Anwender mit einem Datum zur Verfügung gestellt. Z. B. als N20060522-0010 (ein **Nightly Build**) oder als I20060519-1206 (ein **Integration Build**). Offensichtlich ist der Nightly Build weniger oft (automatisch) getestet als der Integration Build.
- Irgendwann scheint eine gute Version erreicht zu sein. Normalerweise ist dies dann die Nummer 3.1.XXX, wobei XXX eine hohe (Build)Nummer / Minor Nummer ist. Eclipse stellt diese dann zuerst als **Milestone Builds** der nächst höheren Version (z. B. 3.2 M3) und dann als **Release Candidate** (z. B. 3.2 RC5) zur Verfügung.
- Erst wenn die Änderungen der Release Candidates zu klein sind, wird Eclipse 3.2 **final** herausgestellt.

Natürlich ergeben Milestonebuilds oder Release Candidates in internen Projekten manchmal nicht viel Sinn, sie können aber (gerade wenn es auch externe Bedeutung hat) durchaus **Motivations-** und **Marketingzwecke** erfüllen.

Ant zählen lassen

Die Frage ist nun, wie kann das unter Ant automatisiert werden?

Ganz einfach: Man definiert eine Property, die die Versionen bezeichnet, und lässt die Buildnummer automatisch von Ant hochzählen:



Quellcode

### Buildnummernvergabe unter Ant

```
<property name="VERSION" value="1.2."/>
<buildnumber/>

...produkt${VERSION}${build.number}.jar...
```

Die erste Zeile definiert das Release und eine Major Version. Diese kann von Hand gesetzt werden, da sie sich nur langsam ändert.

Die zweite Zeile initialisiert die **Buildnummer** mit Null (0).

In der letzten Zeile wird nun beispielsweise das JAR zusammengebaut. Und hier zählt Ant mit jedem Build selbst hoch. Jedes erstellte Jar kann damit eindeutig identifiziert werden.

Eine Datei dient als  
Nummernzähler



Quellcode

Das Ergebnis der von Ant erstellten Zähldatei mit dem Namen `build.number` ist:

#### Quellcode

```
<buildnumber [file="mybuild.number"]/>
#Build Number for ANT. Do not edit!
#Thu May 15 13:56:57 GMT+01:00 2003
build.number=19
```

Analog kann man einen Zeitstempel mit `<tstamp/>` in den Dateinamen oder in sonstige Variablen einbauen.

## 6.3 Weitere Profi-Tasks für Buildmanager

### Ant

Buildfiles strukturieren

Der Task `ant` ruft ein anderes Buildfile auf.

```
<ant dir="subproject"/>
```

Tasks wie `import` oder `ant` sind also sehr wichtig, um dem Buildfile eine Struktur zu geben. Builddateien wachsen in größeren Projekten immer so schnell, dass man „*refactoren*“ muss. Hier bietet es sich also an, das aktuelle Buildfile mit `import` zu erweitern, oder mit `ant` die Kontrolle an das Subbuildfile abzugeben.

### Parallel, Sequential, Waitfor

Parallelverarbeitung

Ant Buildfiles können mittels Tasks parallel ausgeführt werden. `sequential` führt Tasks strikt nacheinander aus. Mit `waitfor` kann auf eine parallele Ausführung gewartet werden, indem bestimmte Bedingungen geprüft werden.

`sequential` wird meist zu Dokumentationszwecken parallel eingesetzt, um abhängige Tasks zur parallelen Ausführung zu zwingen.

In der Praxis ergibt die parallele Ausführung in den seltensten Fällen einen Sinn. Handelt es sich jedoch um hochgradig unabhängige Arbeitsschritte und unterstützt das System Parallelausführung, so kann Parallelisieren wiederum hilfreich sein. Hier lohnt also ein einfacher Test: mit und ohne `parallel` ausführen und dann die Ant-Endausführungszeit in der Console ablesen.



### Exec

Externe Programme ausführen

Mit **Exec** kann jedes beliebige Executable ausgeführt werden. Damit wird Ant wieder so mächtig wie die früheren Buildfiles.

#### Exec Beispiel

```
<exec dir="${src}" executable="cmd.exe" os="Windows 2000" output="dir.txt">
  <arg line="/c dir"/>
</exec>
```

Quelle: Ant Tutorial

Hier wird einfach eine Command-Shell gestartet und ein `dir` als Parameter übergeben. Damit sind auch alle Shell-Kommandos unter Ant verfügbar.

### Skriptsprachen

Es wurde bereits erwähnt, dass unter Ant viele Skriptsprachen verfügbar sind. Hier ein gutes Beispiel aus dem Manual, das die Verwendung von einigen dieser Sprachen zeigt:



Quellcode





Quellcode

**Ruby, Groovy oder Beanshell verwenden**

```
<property name="message" value="Hello world"/>

<script language="groovy">
    println("message is " + message)
</script>

<script language="beanshell">
    System.out.println("message is " + message);
</script>

<script language="judoscript">
    println 'message is ', message
</script>

<script language="ruby">
    print 'message is ', $message, "\n"
</script>

<script language="jython">
    print "message is %s" % message
</script>
```

Auch hier wird die Mächtigkeit von Ant durch den sprachspezifischen Zugriff auf Systemressourcen anschaulich demonstriert.

**SQL**

Datenbankzugriff



Quellcode

**SQL-Task**

```
<sql driver="org.database.jdbcDriver" url="jdbc:database-url"
    userid="sa" password="pwd">
    INSERT INTO person VALUES ('Tim',42);
</sql>
```

## 6.4 Weitere Profi-Tasks für Buildmanager (Fortsetzung)

### Kontrolle mit if / unless

Unter Ant kann eine gewisse Ablaufsteuerung implementiert werden. Dazu sind die Schlüsselwörter `if` und `unless` geeignet. Hier ein Beispiel:



Quellcode

```
<target name="EndModul" if="propertyXvorhanden" />

<target name="TestModul" unless="propertyYvorhanden"/>
```

Es können also in beliebigen Tasks Properties definiert werden, die dann später abgefragt werden können. Ein weiteres gutes Beispiel ist im Kapitel 7 [„Eigene Tasks“](#) zu finden.

### Input

Ant interaktiv

Mit Hilfe des `input` Tasks können Interaktionen mit dem Buildmanager durchgeführt werden. `input` fragt dabei nach einer `property`, die danach gesetzt und ausgewertet werden kann. Dies können beliebige Strings wie z. B. Namen oder Pfade sein. Hier ein paar Beispiele:



Quellcode

#### Input Beispiele

```
<input>Bitte weiter mit *Return*...</input>

<input message="Namen der neuen Datenbank:"
  addproperty="FRONT_DB"/>

<input message="Update fortsetzen:"
  validargs="y,n"
  addproperty="update.cont"/>
```

In diesem Beispiel ist zu erkennen, welche Möglichkeiten diese Abfrage eröffnet.



Achtung

So schön und sinnvoll diese Abfrage auch sein mag: in vielen Fällen **torpediert** dies **den Gedanken der Continuous Integration**:

Buils, die diese Abfrage enthalten, benötigen immer einen Menschen vor dem Rechner. Ziel des Continuous Integration ist es jedoch, dass der Buildprozess mit nur einem Klick gestartet werden kann. Anschließend sollte anhand eines Outputs sofort klar sein, ob dieser Build erfolgreich war oder nicht.

Aus diesem Grund wird der Task `input` nur zu Debug- oder Testzwecken eingesetzt, wenn garantiert ein Buildmanager vor dem System sitzt.

## 7 Eigene Tasks

Wenn man an die Grenzen von Ants stößt und den gesuchten Task nicht findet, kann dieser in der Programmiersprache Java selbst geschrieben werden. Der nachfolgend dargestellte Prozess ist einfach gehalten, damit Personen, die neue Tasks schreiben, diese auch wieder in die Open-Source-Gemeinde zurückgeben können.

Vorgehensweise

- Zunächst wird die eigene Java-Klasse von **org.apache.tools.ant.Task** abgeleitet und in **BuildException** importiert. Dieses Ableiten ist nicht zwingend notwendig, jedoch zu empfehlen. Dagegen ist die Implementierung der **execute**-Methode Pflicht.
- Als nächstes wird eine public void-Methode **execute** implementiert, die eine **BuildException** erzeugen kann. Wurde bei der Übergabe der indirekten Parameter beispielsweise ein falscher Pfad übergeben, so kann der Buildlauf mit dieser Exception abgebrochen werden.



### Frage von Melanie

Um welches Design Pattern handelt es sich hier bei der Verwendung der Execute Methode? (Ausflug in Design-Patterns)

📖 Antwort (Siehe Anhang)

Parameter und Return Values

- Jeder „**Parameter**“ wird als ein (privates) Feld mit einem **setter** implementiert. Da `execute()` keine Parameter hat, können Attribute in diesem neuen Task angegeben werden. Diese können dann von Ant mit selbst geschriebenen **settern** gesetzt werden, so dass man im Java-Code Zugriff auf diese Property-Variablen hat. Return-Values setzt man als User Properties.



Beispiel

Im Folgenden dazu ein Beispiel (Quelle: 📖 [ ES06 ] ):



Quellcode

### Beispiel Schreibzugriff

```
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;
import java.io.*;

public class PruefeZugriff extends Task {
    private String filename; // Task Attribut
    public void setFilename(String filename)
    {this.filename = filename;}
    public void execute() throws BuildException {
        // Vorher ggf. filename prüfen und BuildException werfen...
        File myFHandle = new File(filename);
        if (myFHandle.canWrite()) // Hier die eigene Business-Logik
            project.setUserProperty("myFile", "canWrite");
    }
}
```

Was hier passiert: Es soll ein Task definiert werden, der prüft, ob eine Datei Schreibzugriff besitzt. Dazu wird all das getan, was auf der vorigen Seite beschrieben wurde. Felder und execute-Methode werden implementiert und die Return-User-Property wird gesetzt.

In Java wird dann mit `.canWrite` gearbeitet, um festzustellen, ob die Datei geschrieben werden kann. Kann sie nicht geschrieben werden, bleibt die Return Property undefiniert. Ansonsten ist sie definiert.



Quellcode

Wie kann das Ganze nun verwendet werden?

```
<project name="Beispiel" default="main" basedir=".">
<taskdef name="zugriffOK" classname="PruefeZugriff"/>

<target name="main" depends="PruefeZugriff" if="myFile">
  <replace file="E:/ant-labor/newTask2/datei.txt"
    token="Arol" value="SuperArol" summary="true"/>
</target>

<target name="pruefeZugriff">
  <zugriffOK filename="F:/Builds/nightly/log.txt"/>
  <echo message="Property myFile = ${myFile}!" />
</target>
</project>
```

In der zweiten Klasse wird zunächst der neue Task definiert. Er muss sich im Classpath des zu startenden Ants befinden und daher über das erweiterte Ant Menü der meisten IDEs gestartet werden.

Abfrage der neuen Property

Initial wird der „main“ Task aufgerufen. Dieser hängt von „PrüfeZugriff“ ab. Und genau dort wird der Zugriff im ersten (neuen) Task geprüft. Danach wird die neu gesetzte Property mit `echo` testweise ausgegeben.

Viel wichtiger ist jedoch, dass die Property nun gesetzt ist und erst dann im ersten Target „main“ abgefragt werden kann. Dies geschieht mittels `if`. In diesem Beispiel kann dann danach die Datei ersetzt werden. Ohne den vorigen Test wäre dies nicht möglich gewesen.

## Zusammenfassung

- Es wurde dargestellt, welche Bedeutung der Prozess des Buildmanagements im Zusammenhang mit Continuous Integration hat und welche Buildformen es gibt.
- Der Begriff des Konfigurationsmanagements und die Buildmanagement Historie sowie gängige Tools am Markt wurden vorgestellt.
- Die wichtigen Grundbegriffe von Ant wie Target, Tasks, Dependencies sind definiert worden. Sie sollten in der Lage sein, Ant Skripte selbst zu erstellen. Die IDE Integration, sowie das Property-Handling wurden behandelt.
- Sie sollten die wichtigsten Tasks für das Buildmanagement und die Bedeutung von Pattern und Filesets kennen.
- Fortgeschrittene Konzepte wie Buildnummernvergabe, Strukturierung mit Ant oder Include oder Parallelausführung unter Ant sind Ihnen kein Fremdwort mehr.
- Sie sollten das nötige Handwerkszeug haben, um eigene Tasks zu erstellen.

## Wissensüberprüfung



Formulieren

### Übung ANT-01

#### Wissensfragen zu Buildmanagement

1. Was unterscheidet **Target** und **Tasks**? Definieren Sie bitte beide Begriffe!
2. Was hat Ant mit Continuous Integration zu tun?

Bearbeitungszeit: 15 Minuten



Programmieren

### Übung ANT-02

#### Projektaufgabe - Ant-Skript

Schreiben Sie ein Ant-Skript für Ihr nächstes Programmierungsprojekt. Es soll einige **Targets** und ein **Clear Target** enthalten. Integrieren sie Buildnummern für jeden Build. Versuchen Sie auch Tests (z. B. JUnit) und eine Dokumentation (z. B. Javadoc) in den Build zu integrieren!

Lassen Sie Ihr Programm von Ant oder mit einem eigenen Task starten, damit Sie sofort Akzeptanztests durchführen können.

Bearbeitungszeit: 30 Minuten



Programmieren

### Übung ANT-03

#### Eigene Tasks schreiben

Schreiben Sie einen eigenen Task und testen Sie diesen in Ihrem Buildfile!

Idee: Wie wäre es mit einem Task, der die Länge einer Datei in eine Property schreibt. Falls Sie bessere Ideen haben, dann realisieren Sie bitte diese.

Bearbeitungszeit: 45 Minuten

## Appendix

---

### Melanie:



Um welches Design Pattern handelt es sich hier bei der Verwendung der Execute Methode? (Ausflug in Design-Patterns)

### Antwort:

Es handelt sich um das Command-Pattern. Es wird ein Objekt erstellt, welches Ant zur Verfügung gestellt wird. Ant selbst soll aus diesem Objekt vorwiegend nur eine Methode aufrufen. Wie diese Methode heisst ist egal. Hauptsache das Objekt (=der neue Tasks) hat einen neuen Namen und die Argument / Return Übergabe ist irgendwie gelöst.