Buildwerkzeuge für Javaprojekte

Christian Bunse

Institut für Informatik

03.07.2008

Inhalt

Freie Universität Berlin

- Der Build
- Besonderheiten von Javaprojekten
- Ziele von Buildwerkzeugen
- Continuous Integration
- Vorstellung von Buildwerkzeugen
  - Shellscripte / Batchfiles
  - Make
  - Scons
  - Ant
  - Maven

Der Build - Definition

Freie Universität Berlin

„A build is the process in which a build tool uses other tools to convert the source code into a working product that can be used by other people.“ [2]

„Als Build bezeichnet man den Prozess, in welchem ein Buildwerkzeug andere Werkzeuge benutzt, um aus dem Quellcode ein funktionierendes Produkt zu schaffen, das von anderen verwendet werden kann.”

[2] *Doar, Matthew B.: Practical Developement Environments. p.78, 2005.*

Wirtschaftliche Bedeutung des Build

Builds produzieren zeitlichen Overhead, dieser ist abhängig von Projektgröße und -struktur.

Eine Studie der University of California in Kooperation mit dem US-Departments of Energy aus dem Jahr 1992 [1] über den durch den Build entstehenden Overhead ergab einen durchschnittlichen Wert von 12%, aber Fälle von 20-30% waren nicht ungewöhnlich.

Trotz immenser wirtschaftlicher Bedeutung sind kaum Studien zu diesem Thema vorhanden.

[1] *https://e-reports-ext.llnl.gov/pdf/244668.pdf*

Typische Schritte während des Build

Freie Universität Berlin

- Dateien löschen / anlegen
- Resourcen filtern
- Quellcode erzeugen
- Code Compilieren
- Bibliotheken laden
- ....
- Tests ausführen
- Dokumentation erzeugen
- Container starten / stoppen

Schwierigkeiten beim Bauen

Freie Universität Berlin

- Abhängigkeiten
  - sind das Kernproblem
  - sind z.T. hoch komplex
- Portabilität
- langsame Builds

Besonderheiten von Javaprojekten

Freie Universität Berlin

- plattformunabhängig
- Abhängigkeiten nicht so klar wie bei C / C++

  → Class-Files müssen ausgewertet werden
- häufig automatisch generierter Code
- Reflection
- Inversion of Control

Ziele von Buildwerkzeugen

Freie Universität Berlin

- Verringern des Arbeitsaufwands für die Entwickler
- Verkürzen der Zeitspanne zwischen Änderung am Code und sichtbarem Ergebnis
- automatisches Entdecken entstandener Fehler
- Continuous Integration
- Dokumentation

Ziel:

- Das Projekt sollte sich zu jedem beliebigen Zeitpunkt in stabilem Zustand befinden.

Vorgehen:

- Entwickler speichern Änderungen regelmäßig in SCM (z.B. CVS).
- Das Projekt wird an zentraler Stelle regelmäßig und automatisch gebaut.
- Bei dem zentralen Build werden alle Tests ausgeführt.

Benötigte Werkzeuge:

- SCM -Tool
- Integration Tool
- Buildtool mit den Eigenschaften
  - baut vollautomatisch
  - führt automatisch Tests durch
  - ermöglicht verschiedene Profile
  - Dependency Management

Shellscripte / Batchfiles

- Intuitiver Ansatz zur Automatisierung des Builds

- Vorteile:
  - keine Einarbeitungszeit
  - hohe Flexibilität

- Nachteile:
  - plattformabhängig
  - alle Dateien werden kompiliert
  - keine Fehlererkennung
  - debuggen mühsam

Make

Freie Universität Berlin

- Urvater der Buildwerkzeuge
- entstand 1977
- noch immer weit verbreitet
- Konzepte werden bis heute von den meisten Buildtools verwendet
- vielfältige Implementierungen, u.a. gmake

- Vorteile
  - erlaubt globale Definition von Variablen
  - definiert einzelne Ziele des Builds
  - Deklaration von Abhängigkeiten

  → Zeit zum Bauen des Projekts kann verkürzt werden

Definition von Targets

```
Ziel: Bedingung1 Bedingung2 ...
      Regel 1
      Regel 2 ...
```

Beispiel

```
application.class: application.java
      javac application.java
```

wird überprüft an Hand von Zeitstempeln.

Make - Beispiel

Freie Universität Berlin

```
JFLAGS  = -d /classes
SOURCES = Main.java Sub1.java Sub2.java
CLASSES = Main.class Sub1.class Sub2.class

%.class: %.java
   javac $(JFLAGS) $<

all: $(CLASSES)

clean:
   /classes/rm $(CLASSES)

doc: $(SOURCES)
   javadoc -author -d /docs $(SOURCES)

Main.class: Sub1.class Sub2.class
```

- Nachteile
  - Dependencies und Rekursion
  - Nutzen von Zeitstempeln
  - Debugging ist schwierig
  - schwer zu portieren

SCons

Freie Universität Berlin

- entstanden 2001
- neuste Version: 0.98.5
- deklarativ

- Vorteile
  - automatisches Erkennen von Abhängigkeiten (für C)
  - plattformunabhängig
  - arbeitet mit Checksum statt Zeitstempel
  - Scriptsprache (Phyton) für Buildfiles
  - unterschtützt hierarchische Builds
  - (gute Dokumentation)

Scons - Beispiel

Freie Universität Berlin

```
import os
env = Environment(ENV = os.environ)
test = Environment(ENV = os.environ)

test.Append(JAVACLASSPATH = 'main/classes')

print 'compile files in main'
env.Java('main/classes', 'main/src')

print 'compile files in test'
test.Java('test/classes', 'test/src')
```

- Nachteile
  - Buildfile ist z.T. Deklarativ und z.T. imperativ
  - nicht für Java gedacht: nur 1 von 26 Kapiteln der Dokumentation beschäftigt sich mit Java
  - keine Dependecy-Erkennung für Java
  - geringer Funktionsumfang für Java

Ant

Freie Universität Berlin

- Entstanden 2000

- Vorteile
  - plattformunabhängig
  - Implementierung in Java
  - deklarativ (XML)
  - enormer Umfang an Tasks
    - core: > 80
    - optional: > 60
    - weitere frei verfügbar ...
  - stärkere Trennung zwischen Buildimplementierung und Builddefinition
  - gute Dokumentation

Ant – Beispiel (1)

Freie Universität Berlin

```
<project>

   <property name = "src.dir" value = "src" />

   <target name = "clean">
      <delete dir = "build" />
   </target>

   <target name = "compile">
      <mkdir dir = "build/classes" />
      <javac srcdir  = "${src.dir}"
             destdir = "build/classes" />
   </target>

</project>
```

Ant – Beispiel (2)

Freie Universität Berlin

```
<project>
   ...
   <target name = "jar" depends = "compile">
      <mkdir dir = "build/jar"/>

      <jar destfile = "build/jar/myApp.jar"
           basedir  = "build/classes">
         <manifest>
            <attribute name  = "Main-Class"
                       value = "Main" />
         </manifest>
      </jar>
   </target>
   ...
</project>
```

- Nachteile
  - spezielle Tasks z.T. aufwändig zu implementieren
  - sehr statisch
  - bei großen Projekten hoher deklarativer Aufwand
  - Probleme bei hierarchischen Projekten
  - mehrfaches kompilieren von Klassen

Entstand aus dem Versuch, den Build Prozess des Jakarta Turbine Projekts zu vereinfachen.

Ziele:

- Standard für Projektbau
- einfacher, standardisierter Weg um Projektinformationen zu veröffentlichen
- einfache Möglichkeit um JARs zwischen verschiedenen Projekten zu teilen

Maven 1

Freie Universität Berlin

- entstand 2001
- basiert auf Ant-Tasks
- ist extrem flexibel
- weiterhin hoher deklarativer Aufwand
- Entwicklung wurde zu Gunsten von Maven 2 eingestellt.

- Entstanden 2005
- Goals direkt in Java implementiert

- Standards vorgegeben:
  - Convention over Configuration
  - klarer Build Lifecycle
  - einheitliche Verzeichnisstrukturen
  - einheitlicher Buildprozess
  - Dokumentation wird automatisch erstellt
  - werden durch Vererbung durchgesetzt

- Repository – System
  - selbstständiges Herunterladen von Bibliotheken / Plug-ins
  - vereinfacht das Handling bei vielen Abhängigkeiten
  - kann die Build-Dauer reduzieren
  - fördert die Trennung von Konfigurationsmanagement und Softwareentwicklung
- Unterstützung für Multiprojekte
- Profile
- Plug-ins
- weitreichende Dokumentation

Default Build Lifecycle (Auszug)

- process-resources
- compile
- process-test-resources
- test-compile
- test
- package
- install
- deploy

Maven 2 - Standard Projektstruktur

Freie Universität Berlin

```
-- src
        -- main
                -- java
                -- resources
                -- (webapp ...)
                -- (filters ...)
        -- test
                -- java
                -- resources
        -- site
                -- ...
-- target
        -- ...
-- pom.xml
```

Maven 2 – Beispiel (1)

Freie Universität Berlin

```
<project ... >

   <modelVersion>4.0.0</modelVersion>

   <groupId>de.fu-berlin.inf</groupId>
   <artifactId>my-app</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>

   <name>First Maven Application</name>

</project>
```

Maven 2 – Beispiel (2)

Freie Universität Berlin

```
<project ... >
   ...
   <dependencies>
      <dependency>
         <groupId>de.fu-berlin.inf</groupId>
         <artifactId>my-app</artifactId>
         <version>1.0-SNAPSHOT</version>
      </dependency>
      ...
   </dependencies>
   ...

</project>
```

Maven 2 – Beispiel (3)

Freie Universität Berlin

```
<project ...>
   ...
   <parent>
      <groupId>de.fu-berlin.inf</groupId>
      <artifactId>my-app-parent</artifactId>
      <version>1.0-SNAPSHOT</version>
   </parent>
   ...
```

Maven 2 – Beispiel (4)

Freie Universität Berlin

```
<project ...>
   ...
   <groupId>de.fu-berlin.inf</groupId>
   <version>1.0-SNAPSHOT</version>
   <artifactId>my-app-parent</artifactId>
   <packaging>pom</packaging>

   <modules>
      <module>my-app</module>
      <module>my-webapp</module>
   </modules>
  ...
```

- Nachteile
  - Plug-ins teilweise schlecht dokumentiert
  - Plug-ins teilweise als generisch gekennzeichnet obwohl sie einem speziellem Zweck dienen
  - hat noch nicht die Reife von Ant

Zeitanteile am Build (Beispiel)