Proseminar: Fortgeschrittene Programmierkonzepte

Build-Systeme

Ein Überblick über Ant, Maven und Gradle

von Cameron Pitsch und Aaron Küsters

1 Einleitung

In dem ersten Teil der Einleitung werden zunächst wichtige Konzepte und Vokabeln eingeführt bevor im zweiten Teil auf den Aufbau der Ausarbeitung eingegangen wird.

1.1 Was sind Build-Systeme?

Unter dem *Builden* (dt. Bauen) versteht man in der Softwareentwicklung den Vorgang des Erstellens einer Anwendung aus vorher entwickeltem Quellcode. Dies enthält typischerweise auch das *Kompilieren* von Quellcode, aber nicht immer, da dies z. B. bei interpretierten Programmiersprachen (wie Python) nicht notwendig ist [1].

Ein weiteres wichtiges Konzept beim Builden ist das sogenannte *Packaging* (dt. Verpacken) von Software. Dabei werden die ggf. zahlreichen (kompilierten) Einzeldateien zusammen mit einigen Informationen in ein Archiv verpackt, um so kompakt bspw. an den Endbenutzer ausgeliefert zu werden. Bei der Programmiersprache Java werden Projekte oft in eine sogenannte *JAR-Datei* verpackt. Die einfachste und kleinste Version hiervon nennt sich *Skinny JAR*, welche nur die kompilierten Klassen des eigenen Projektes enthält. Dies wird z. B. dafür verwendet, um ein Projekt für andere Projekte als Bibliothek zur Verfügung zu stellen. Eine *Fat* oder *Uber JAR* ist die umfassendste, also größte Art von JAR und enthält alles was benötigt wird um das Projekt ohne zusätzliche Dateien auszuführen, also auch alle Java-Biblotheken, von denen das Projekt abhängt [2]. Dabei ist zusätzlich zu beachten, dass eventuell (versionsbedingte) Namenskonflikte zwischen Abhängigkeiten auftreten können und per Umbenennung der Dateien und Ordner, dem sogenannten *Shading*, gelöst werden müssen.

Ein *Build-System* bezeichnet nun eine softwarebasierte Automatisierung des Build-Prozesses, sodass lediglich ein Ziel (wie z. B. package) angegeben werden muss und das Build-System das gewünschte Resultat von selbst erreicht.

Neben den erwähnten Aufgaben werden heutzutage Build-Systeme auch für weitere Aufgaben, wie z. B. das automatisierte Testen von Programmen, verwendet, worauf aber im Nachfolgenden nicht weiter eingegangen wird. Damit ein Build-System den Build-Prozess sinnvoll automatisieren kann, muss das entsprechende Build-System erst konfiguriert werden. Dies passiert meist in einer oder mehreren Textdateien, die *Build-Skripte* oder

auch *Build-Konfigurationen* genannt werden.

1.2 Aufbau

Im Folgenden werden die drei für Java populären Build-Systeme *Ant*, *Maven* und *Gradle* vorgestellt. Dazu gehen wir jeweils zunächst auf die Entstehungsgeschichte ein, geben das Grundkonzept kurz wieder, erläutern wie die Konfiguration aussieht und demonstrieren eine beispielhafte Build-Konfiguration anhand eines Beispielprojektes. Dazu verwenden wir ein einfaches Java-Programm, mit welchem man die verbleibende Zeit zu einem festen Zeitpunkt (in dem Beispiel das Semesterende vom Sommersemester 2019) berechnet und dem Benutzer ausgibt. Dafür verwendet es Objekte und Methoden von einer externen Java-Bibliothek `joda-time`. Die jeweilige Beispielkonfiguration sollte dann das Programm kompilieren, -soweit möglich- die externe Abhängigkeit `joda-time` herunterladen und einbinden, und das Programm als *Uber JAR* verpacken können.

```
1  import org.joda.time.*;
2  public class DayCounter{
3      public static void main(String[] args){
4          System.out.println(tageBisFerien());
5      }
6      public static int tageBisFerien(){
7          ...
8      }
9  }
```

2 Apache Ant

Apache Ant ist das älteste der drei betrachteten Build-Systeme. Die Entwicklung von Ant startete 1999 im Rahmen des TomCat Projektes, welches James Duncan Davidson vorantrieb, weil er ein Build-System wie das populäre *make* für Java suchte. Deshalb ähnelt Ant auch *make* stark. Die Version 1.0 von ANT (Another Neat Tool) erschien im März 2000 [3].

2.1 Grundkonzept

Ein wesentlicher Bestandteil eines Ant-Projekts ist die Konfigurationsdatei `build.xml`, welche in der populären Auszeichnungssprache XML geschrieben wird. In dieser Datei werden bestimmte *Targets* (dt. Ziele) definiert, welche den Aufgaben entsprechen, die Ant automatisieren soll, wie z. B. das Kompilieren des Projektes oder das Leeren eines Ordners. Ähnlich wie eine Anleitung beinhalten diese Ziele *Tasks*, also Anweisungen, die zur Erfüllung des jeweiligen Ziels nacheinander ausgeführt werden müssen. Der Inhalt, also die Anleitung, der Targets werden vom Nutzer selbst geschrieben, während Tasks

zum Großteil (etwa 150) schon vorimplementiert sind (wie `javac` zum Kompilieren von Quellcode, `mkdir` zum Erstellen eines Ordners, etc.). Nutzer können aber auch selber Targets in Java implementieren [4]. Dadurch, dass der Nutzer den Großteil der Vorgänge selbst implementieren muss, ist dieses Build-System sehr anpassbar. In Ant ist allerdings ohne zusätzliche Erweiterung eine Automatische Abhängigkeitsauflösung nicht möglich. Viele Projekte verwenden als *Dependency-Manager* die Software Apache Ivy um diese Funktionslücke zu schließen.

2.2 Konfiguration

Jedes Target verfügt über einen Namen *name*, welcher vor allem dafür genutzt wird dieses Ziel über die Kommandozeile aufzurufen (z. B. führt `ant compile` das Target *compile* aus). Außerdem dient der Name als Identifikation, um von anderen Targets (als Abhängigkeit) referenziert zu werden. Targets können voneinander abhängen und werden dann nacheinander ausgeführt. Mit dem Schlüsselwort *depends* (dt. abhängen) in einem Target-Tag wird verdeutlicht, dass vor dem Ausführen der Anweisungen in diesem Target zuerst ein anderes Target erreicht werden muss [4]. Um externe Java-Bibliotheken einzubinden, müssen diese zunächst manuell heruntergeladen und in das Projektverzeichnis platziert werden. Dann können diese mit Hilfe bestimmter Task-Attributen eingebunden werden. Das *Shading* beim Packaging ist bei Ant nicht vorimplementiert und muss, wenn nötig, vom Nutzer selbst durchgeführt werden.

2.3 Anwendung an Beispielklasse

```
1  <project name="DayCounter">
2
3    <target name="init">...</target>
4
5    <target name="clean">
6      <delete dir="build"/>
7    </target>
8
9    <target name="compile" depends="clean,init">
10     <javac destdir="build/classes">
11       <src path="src"/>
12       <classpath>
13         <pathelement location="lib/joda-time-2.10.1.jar"/>
14       </classpath>
15     </javac>
16   </target>
17
18   <target name="package" depends="compile">
19     <jar destfile="build/daycounter.jar">
20       <fileset dir="build/classes"/>
21       <zipfileset src="lib/joda-time-2.10.1.jar"/>
```

```
22        <manifest>
23          <attribute name="Main-Class" value="DayCounter"/>
24        </manifest>
25      </jar>
26    </target>
27
28  </project>
```

In der passenden `build.xml`-Datei zum Beispiel aus 1.2 wird zuerst in Zeile 5ff. das Target `clean` definiert, welches die Task `delete` aufruft, um den Ordner mit dem Namen `'build'` zu löschen. Darüber hinaus befindet sich in Zeile 9ff. das Target `compile`, bei welchem in dem `depends` Attribut zwei weitere Targets erwähnt werden, nämlich `clean` und `init`. Hier werden also zuerst `clean` und danach `init` erfüllt, bevor die Tasks in `compile` selber ausgeführt werden. Wie man in Zeile 10ff. erkennen kann, können Tasks mehrere Argumente übergeben bekommen, nämlich hier z. B. ein `src` Tag mit dem Attribut `path` welches auf `'src/'` gesetzt wird. Hier dient es dazu, den Ordner festzulegen, wo der Compiler die zu kompilierenden Klassen findet. Weiter wird im Attribut `classpath` die externe Bibliothek `joda-time` per Angabe ihres Pfades eingebunden. Dafür musste sie, wie zuvor beschrieben, erst manuell heruntergeladen und verschoben werden. Wenn man jetzt zum Beispiel `ant compile` im Ordner des Projektes aufruft, wird zuerst das Target `clean`, dann `init` erfüllt, bevor danach alle Tasks, die sich in `compile` befinden ausgeführt werden. Dies führt dann dazu, dass zuerst der 'build' Ordner gelöscht wird, dann in `init` wieder, leer, erstellt wird (inkl. `'build/classes'`) und dann in `compile` das Projekt kompiliert wird und die fertigen .class-Dateien im Pfad `'build/classes'` gespeichert werden. Zum Schluss wird noch der Target `package` definiert. Hierin wird zunächst die Task `jar` aufgerufen um das Projekt als eine JAR-Datei zu verpacken. Dem `jar` Aufruf werden einige Argumente übergeben. Diese sind zum einen `destfile`, der Pfad, inklusive Namen, wo die JAR-Datei gespeichert werden soll, `fileset`, der Pfad wo die zu verpackenden Klassen sich befinden und `zipfileset`, das Verzeichnis wo sich schon verpackte Projekte befinden von denen das aktuelle Projekt abhängt. Schließlich wird noch die zu verwendende Main-Klasse, also `DayCounter` angegeben. Mit dem Aufruf `ant package` wird also im Verzeichnis `build` die Uber-JAR `daycounter.jar` erstellt, welche aus den Klassen in `build/classes` und der angegebenen externen Bibliothek entstanden ist.

Mit der Ausführung von z. B. `clean` wird das build-Verzeichnis (inklusive Inhalt) gelöscht.

```
Projekt-Ordner
├── build.xml
├── build
│   └── ...
├── src
└── ...
$ ant clean
```

```
Projekt-Ordner
├── build.xml
├── src
└── ...
$
```

3 Apache Maven

Etwa vier Jahre nach der Erstveröffentlichung von Ant wurde die erste Version von Apache Maven vorgestellt. Maven enstand als Unterprojekt von dem Jakarta Alexandria Projekt und wurde ebenfalls von der Apache Software Foundation entwickelt. Das Ziel war eine Alternative für Ant zu schaffen, welche dem Benutzer, unter dem Einhalten von bestimmten Konventionen, wesentliche Teile der Konfiguration abnimmt. Seit der ersten Version 2004 wurden 2005 und 2009 je eine neue Hauptversion (Maven 2.0 und Maven 3.0) veröffentlicht [5].

3.1 Grundkonzept

Anders als Ant baut Maven in seinem Grundkonzept auf einigen Konventionen auf. So wird zum Beispiel eine spezielle Verzeichnisstruktur, in dem der Projekt-Quellcode, falls nicht anders spezifiziert, gespeichert werden soll, festgelegt. Zudem sind viele *Ziele* oder *Lebenszyklen* vordefiniert und müssen nicht mehr selbst in der Konfiguration befehlsweise implementiert werden. Dies ermöglicht einen schnellen Einstieg und verkürzt die Build-Konfiguration. Zusätzlich zu den standardmäßig unterstützten Zielen können auch eigene Erweiterungen (sog. *Plug-ins*), die weitere Ziele definieren, geschrieben und genutzt werden. Des Weiteren besitzt Maven echte Unterstützung zur Verwaltung von Abhängigkeiten (sog. Dependencies) und sogar eine zentrale *Repository* (Datenspeicherstelle) für verschiedene Programmbibliotheken und -module. So müssen der Repository bekannte Abhängigkeiten lediglich deklariert werden und können dann automatisch heruntergeladen und eingebunden werden. Maven verwendet ein sogenanntes Project Object Model (kurz *POM*) zur Konfiguration (siehe 3.2) von einem Projekt, welches wie bei Ant in XML angegeben wird. Wie der Name schon verrät, wird mit der POM das Projekt in seinen Grundzügen wie ein Objekt behandelt, welches selbst eigene Attribute (wie z.B. die aktuelle Versionsnummer) besitzt [6]. Ähnlich wie in der Objekt-Orientierten Programmierung, können so auch Projekt-Objekte von einander erben. Zusätzlich kann das Projekt mit den notwendigen POM-Attributen eindeutig identifiziert werden, wodurch das Angeben von Abhängigkeiten auch über die jeweiligen POM-Attribute möglich ist.

3.2 Konfiguration

Wie schon in 3.1 beschrieben, wird in Maven eine gewisse Verzeichnisstruktur vorgegeben, wobei sich in dem Verzeichnis `src/main/java` die normalen Quellcodedateien befinden. Das erspart in der Konfiguration das Festlegen von den Dateipfaden, sodass in der einfachsten Variante lediglich einige Projekteigenschaften, wie die Gruppen-ID (`groupId`) der entwickelnden Organisation, ein Name (für gepackten Dateien) (`artifactID`) und eine

Versionsnummer (`version`) des Projektes, angegeben werden müssen. Sollen zudem noch Abhängigkeiten deklariert werden, geschieht dies in der Sektion `dependencies`, wo dann jede Abhängigkeit (`dependency`) durch ihre jeweilige `groupId`, `artifactID` und `version` genau identifiziert wird. Ähnlich dazu können auch Plug-ins eingebunden und konfiguriert werden.

```
Projekt-Ordner
├── pom.xml
├── src
│   ├── main
│   │   └── java
│   └── test
│       └── java
└── target
```

3.3 Anwendung an Beispielklasse

```
1  <project>
2      <modelVersion>4.0.0</modelVersion>
3      <groupId>com.example</groupId>
4      <artifactId>daycounter</artifactId>
5      <version>0.1.0</version>
6      <properties>
7          <maven.compiler.source>1.8</maven.compiler.source>
8          <maven.compiler.target>1.8</maven.compiler.target>
9      </properties>
10     <dependencies>
11     <dependency>
12       <groupId>joda-time</groupId>
13       <artifactId>joda-time</artifactId>
14       <version>2.10.1</version>
15     </dependency>
16   </dependencies>
17     <build>
18         <plugins>
19             <plugin>
20                 <groupId>org.apache.maven.plugins</groupId>
21                 <artifactId>maven-shade-plugin</artifactId>
22                 <version>2.1</version>
23                 <executions>
24                     <execution>
25                         <phase>package</phase>
26                         <goals> <goal>shade</goal> </goals>
27                         <configuration>
28                             <transformers>
29                                 <transformer implementation="org.apache.maven.
                                       plugins.shade.resource.
                                       ManifestResourceTransformer">
30                                 <mainClass>DayCounter</mainClass>
31                             </transformer>
32                         </transformers>
33                     </configuration>
34                 </execution>
```

```
35                  </executions>
36              </plugin>
37          </plugins>
38      </build>
39  </project>
```

In der einfachsten Version der Konfigurationsdatei `pom.xml` für die vorgestellte Beispielklasse, werden in Zeile 3-5 zunächst die wichtigsten Projekteigenschaften festgelegt. In Zeile 6-9 wird die für die Kompilierung zu verwendende Java-Version angegeben, die Maven dann als Argumente an den Java-Compiler übergibt. Danach werden in den Zeilen 11-16 noch die verwendeten Abhängigkeiten angegeben, also in dem Beispiel nur die eine Abhängigkeit an die joda-time Bibliothek. Diese wird auch durch die jeweiligen POM-Attribute, die das Projekt der Bibliothek besitzt, genau angegeben. Zur Verdeutlichung hier ein Ausschschnitt der projekteigenen `pom.xml` Konfigurationsdatei des Projektes joda-time, in dem diese POM-Attribute festgelegt werden [7]:

```
1  ...
2  <groupId>joda-time</groupId>
3  <artifactId>joda-time</artifactId>
4  ...
```

Die Möglichkeit eine *Skinny JAR* zu erstellen bietet Maven ohne weitere Konfiguration vordefiniert an. Um ein *Uber JAR* zu erzeugen muss allerdings das Maven Shade Plug-in verwendet werden, welches möglicherweise auftretende Namenskonflikte löst. Im Abschnitt von Zeile 17-38 wird das verwendete Plug-in angegeben und dafür gesorgt, dass bei dem Target `package` zusätzlich, wie in der Einleitung erwähnt, ein *Shade* durchgeführt wird. Dabei wird noch durch die Konfiguration in Zeile 27-33 die zu verwendende Main-Klasse in der JAR-Datei gespeichert.

4 Gradle

Gradle ist das neuste der drei Projekte und baut auf den Konzepten von Maven weiter auf, mit dem Ziel den vom Nutzer geforderten Aufwand weiter zu verkleinern und die Stärken vieler vorheriger Build-Systems zu vereinen [8]. Entstanden ist Gradle aus Frustration des Gründers Hans Dockter mit den von ihm bisher verwendeten Build-Systemen. 2007 veröffentlichte er die erste Version (0.1) des Open-Source Projektes Gradle [9].

4.1 Grundkonzepte

Wie bereits bei Maven beschrieben, wird auch bei Gradle viel Konfiguration durch Konventionen ersetzt. So wird unter anderem auch die von Maven genutzte Verzeichnisstruktur standardmäßig festgelegt. Im Gegensatz zu Ant und Maven basiert Gradle nicht auf XML, um die Build-Konfiguration abzubilden. Stattdessen wird eine sogenannte Domain-specific language (zu dt. Domänenspezifische Sprache, *DSL*), welche auf Groovy basiert, verwendet.

Diese sorgt dafür, dass das geschriebene Build-Skript als ein Objekt der Klasse `Project` in Gradles API gelesen werden kann und somit dann die Eigenschaften des Projektes, wie z. B. die Abhängigkeiten, von Gradle referenziert werden [8]. Die Verwendung von einer DSL anstelle von XML soll zudem die Lesbarkeit der Konfiguration verbessern. Ein weiteres wichtiges Konzept von Gradle ist die Erweiterbarkeit mit Hilfe von Plug-ins, welche auch selber neue Attribute zu der DSL von Gradle hinzufügen können [10]. Während einige Plug-ins (wie z. B. java) vordefiniert sind, werden die meisten von Nutzern geschrieben und für andere bereitgestellt. Dazu existiert, ähnlich wie bei dem zentralen Repository für Abhängigkeiten von Maven, ein sogenanntes *Plug-in-Portal* wo diese Plug-ins gesammelt werden. Ebenfalls wie bei Maven sind so mit der Wahl geeigneter Plug-ins viele Ziele vordefiniert, was wie bereits erwähnt einen schnellen Einstieg ermöglicht und für kleinere Konfigurationsdateien sorgt.

Ein weiterer Unterschied zu den bereits vorgestellten Build-Systemen liegt in den vielen Verwendungsmöglichkeiten von Gradle. So löst sich Gradle von der starken bis exklusiven Bindung an Java der Vorgänger und bietet bspw. auch Unterstützung für native Applikationen, welche in C++ oder C geschrieben werden. Zwar können auch Ant oder Maven mit entsprechender Konfiguration C oder C++ Programme kompilieren, allerdings ist dies nur eher mühsam und mit der Hilfe von externen Plug-ins möglich, während Gradle dies standardmäßig unterstützt. Die Unterstützung von verschiedenen Sprachen macht Gradle vor allem bei neueren Projekten beliebt, welche für verschiedene Elemente des Projektes unterschiedliche Programmiersprachen verwenden. Dies wird *polyglot programming* (dt. mehrsprachiges Programmieren) genannt [8]. Inzwischen wird Gradle, auch aus diesem Grund, als standardmäßiges Build-System für die Entwicklung von Android-Applikationen in der Entwicklungsumgebung Android Studio verwendet.

4.2 Anwendung an Beispielklasse

```
1  plugins {
2      id 'java'
3      id 'application'
4      id 'com.github.johnrengelman.shadow' version '5.0.0'
5  }
6  repositories {
7      mavenCentral()
8  }
9  dependencies {
10     implementation 'joda-time:joda-time:2.1'
11 }
12 mainClassName = 'DayCounter'
```

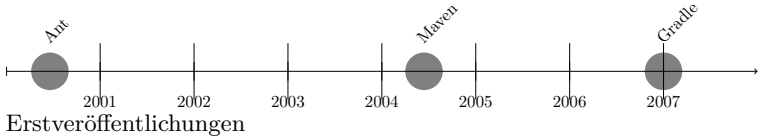
Die Datei `build.gradle` enthält zunächst in der `plugin`-Sektion eine Auflistung der verwendeten Plug-ins, im Beispiel `java` und `application` um das Projekt zu kompilieren und auszuführen. Allerdings kann Gradle, genau wie bereits Maven, durch diese Konfiguration lediglich Skinny-JARs erzeugen. Um auch Uber-JARs zu erzeugen muss, ebenfalls wie

bei Maven, ein Plug-in verwendet werden. Das Pendant zu Mavens `shade` Plug-in ist das `shadow` Plug-in für Gradle, dessen Verwendung in Zeile 4 angegeben wird. Der Eintrag in der `repositories`-Sektion setzt fest, woher Gradle Abhängigkeiten beziehen soll. Dort wird im Beispiel die zentrale Repository von Maven verwendet (siehe 3.1). Schließlich wird noch die Abhängigkeit mit Name und Version benannt. Am Ende ist noch die zu verwendende Main-Klasse angegeben.

5 Vergleich

5.1 Zeitlich

Während bei der (ersten) Entwicklung von Ant der Fokus vor allem darauf lag, ein `make`-ähnliches Build-System für Java zu erstellen, wurde bei der Entwicklung von Maven versucht eine eigene Herangehensweise zu erkunden und selber Standards festzulegen. Das Ziel für die Entwicklung von Maven war vor allem die bis dahin oft genutzten sehr individuellen aber trotzdem repetitiven Ant Build-Konfiguration zu ersetzen [11]. Gradle, als das neuste der drei Projekte, baut viel auf den Konzepten von Maven auf.



Erstveröffentlichungen

5.2 Konzeptionell

Die wesentlichen konzeptionellen Unterschiede zwischen Ant, Maven und Gradle liegen vor allem in der Berufung auf Konventionen oder Standards. Während Ant zu großem Teil auf Annahmen verzichtet, gehen Maven und Gradle in wesentlichen Punkten, wie Verzeichnisstruktur, von bestimmten Voraussetzungen aus. Dazu kommt, dass bei Ant und Maven der Hauptfokus auf Java gelegt wird, während Gradle ohne externe Erweiterung bereits viele andere Sprachen unterstützt und so attraktiv für mehrsprachige Softwareprojekte ist. Ein weiterer Unterschied liegt in der Verwendung der Konfigurations- oder Skriptsprachen. Ant und Maven greifen dafür auf XML zurück, während Gradle auf eine eigene DSL baut. Wenn man die Konzepte von prozedurale und objektorientierte Programmierung verallgemeinert auf Build-Systeme überträgt, dann fällt Ant mit der Aneinanderreihung von verschiedenen Befehlen (`tasks`) eher in den prozeduralen Bereich, während Maven, mit dem POM Konzept, und Gradle, mit der DSL, ein Projekt eher als ein Objekt auffassen.

Literaturverzeichnis

[1] Lee, Kevin A.: *The Buildmeister's Guide*. Lulu.com, 2006

[2] James Falkner: *The Skinny on Fat, Thin, Hollow, and Uber*. https://dzone.com/articles/the-skinny-on-fat-thin-hollow-and-uber. Version: April 2017

[3] Holzner, Steve: *Ant: The Definitive Guide: Complete Build Management for Java*. O'Reilly Media, Inc., 2005

[4] The Apache Software Foundation: *Apache Ant™ 1.10.5 Manual*. https://ant.apache.org/manual/index.html. Version: April 2019

[5] Apache Maven Project: *Maven Releases History*. https://maven.apache.org/docs/history.html. Version: April 2019

[6] O'Brien, Tim ; Moser, Manfred ; Casey, John ; Fox, Brian ; Van Zyl, Jason ; Redmond, Eric ; Shatze, Larry: *Maven: The complete reference*. 2008

[7] joda-time: *Github: joda-time/pom.xml*. https://github.com/JodaOrg/joda-time/blob/master/pom Version: April 2019

[8] Muschko, Benjamin: *Gradle in action*. Manning, 2014

[9] InfoQ: *Breaking Open: Gradle (Interview)*. https://www.youtube.com/watch?v=XXoIzzcJr80

[10] Gradle: *Using Gradle Plugins*. https://docs.gradle.org/current/userguide/plugins.html. Version: April 2019

[11] Apache Maven Project: *What is Maven?* https://maven.apache.org/what-is-maven.htmll. Version: April 2019