Joachim Baumann

Gradle

Ein kompakter Einstieg in das Build-Management-System

→ Mit einem Geleitwort von Hans Dockter, Erfinder von Gradle

dpunkt.verlag

Joachim Baumann ist bei der codecentric AG in Frankfurt/Main tätig. Er beschäftigt sich seit 17 Jahren mit objektorientierter Programmierung und mit der Erstellung und Wartung großer Softwaresysteme, hauptsächlich in den Rollen des Architekten und Projektleiters. Außerdem unterrichtet er seit über zehn Jahren in Seminaren, Schulungen und Vorlesungen zu Themen, die mit objektorientierter Entwicklung in Verbindung stehen. Er ist Autor des Buchs »Groovy - Anwendungen und fortgeschrittene Techniken« aus dem gleichen Verlag.

Joachim Baumann

Gradle

Ein kompakter Einstieg in modernes Build-Management

dpunkt.verlag

Joachim Baumann
joachim.baumann@codecentric.de

Lektorat: René Schönfeldt
Copy Editing: Sandra Gottmann, Münster-Nienberge
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Buch 978-3-86490-049-5
PDF 978-3-86491-336-5
ePub 978-3-86491-337-2

1. Auflage 2013

Wieblinger Weg 17
69115 Heidelberg

5 4 3 2 1 0

Vorwort

Zielgruppen für dieses Buch

Dieses Buch wendet sich an zwei Zielgruppen: Entwickler, die Projekte auf einfache Weise bauen wollen, und Infrastrukturverantwortliche, die nicht nur mit dem Bauen von Projekten betraut sind, sondern sich auch Gedanken über Software-Verteilung (Deployment) auf verschiedenen Umgebungen machen.

Gradle unterstützt Sie darin, die notwendigen Randbedingungen für ein Projekt in einfacher und flexibler Weise zu beschreiben und es mit verschiedenen Build-Servern beziehungsweise Continuous-Integration-Servern zu integrieren.

Gradle ist in exzellenter Weise geeignet, komplexe und unverständliche Ant- oder Maven-Builds abzulösen, insbesondere in Multiprojektumgebungen. Hierbei kann Gradle auch – anders als andere Build-Werkzeuge – ambitionierten Build- und Projektautomatisierungsanforderungen gerecht werden.

Wenn Sie sich hierdurch angesprochen fühlen und einen kompakten Einstieg in die Arbeit mit Gradle suchen, dann ist dieses Buch für Sie geeignet.

Notwendige Vorkenntnisse für das Buch

Da sich dieses Buch mit der Verwendung eines spezifischen Werkzeugs im Bereich Build-Management beschäftigt, sollten Sie mit dem allgemeinen Thema vertraut sein. Da Sie sich aber mit diesem Thema automatisch beschäftigt haben, wenn Sie zum Beispiel schon mal ein Java-Programm mit mehr als einer Klasse geschrieben haben, ist diese Hürde sehr niedrig. Wenn Sie sich bereits detaillierter mit dem Thema beschäftigt haben, überspringen Sie am besten die Einleitung und starten mit Kapitel 2, um direkt in die Verwendung von Gradle einzusteigen, oder, falls Sie Gradle schon kennen, mit Kapitel 7, um zu erfahren, wie Sie sehr einfach Erweiterungen (eigene Tasks und Plug-ins) mit Gradle realisieren können.

Vertrautheit mit der Sprache Groovy hilft, da alle Build-Skripte Groovy-Skripte sind, die eine domänenspezifische Sprache (DSL) zur Beschreibung des Builds zur Verfügung stellen. Während die Groovy-Kenntnisse bei den einfachen und normalen Beispielen nicht absolut notwendig sind, werden Sie bei den komplexeren Beispielen und bei der Programmierung von Plug-ins interessant, da es vorteilhaft ist, diese in Groovy zu formulieren. Wenn Sie keine Kenntnisse in Groovy mitbringen, sind zumindest Java-Kenntnisse sehr hilfreich, da sich Groovy-Programme im Normalfall von Java-Programmierern lesen lassen (Groovy ist mit Absicht sehr ähnlich wie Java gestaltet). Außerdem sind die Beispielprogramme meistens in Java formuliert, so dass auch hier Java-Kenntnisse helfen.

Was dieses Buch nicht enthält

Dieses Buch enthält keine Einführung in das Thema Release-Management/Software-Lifecycle-Management. Auch Berichte aus der Praxis sind nicht der Fokus des Buchs. Genauso wenig kann dies Buch eine Übersicht über alle existierenden Plug-ins geben und die jeweiligen Funktionalitäten umfassend beschreiben. Das Buch ist auch kein allumfassendes Nachschlagewerk für Gradle. Es liefert auch nicht die magische Anleitung, wie Sie Ihren bisherigen Build auf Knopfdruck mit Gradle implementieren.

Dies zu tun, hätte den Umfang des Buchs mehr als verdoppelt.

Was dieses Buch enthält

Dieses Buch ermöglicht Ihnen den schnellen Einstieg in die Verwendung von Gradle.

Nach der Lektüre kennen Sie die Grundlagen und gängige Szenarien, in denen Gradle eingesetzt wird. Außerdem können Sie mit dem erworbenen Verständnis die sehr umfassende Dokumentation von Gradle selbst ([Gradle-Website]) deutlich effizienter verwenden und fortgeschrittene Anforderungen lösen.

In diesem Buch finden Sie für verschiedene Szenarien das beste Vorgehen, um Build-Management aufzusetzen, so dass Sie möglichst schnell ein produktives System erhalten, das Sie dann Schritt für Schritt perfektionieren können.

Zusätzlich erfahren Sie, wie Sie eigene Erweiterungen für Gradle programmieren, um Gradle auch an die komplexesten Umgebungen anzupassen.

Danksagung

Ein sehr großer Dank gebührt der großen Zahl von Reviewern, die das Manuskript nicht nur gelesen haben, sondern auch sinnvolle Änderungsvorschläge und Anregungen lieferten. Sie haben dieses Buch deutlich verbessert.

Auch meinen Kollegen, die mir Feedback gegeben haben, bin ich zu Dank verpflichtet, wie auch meiner Firma, die es fördert, dass jemand in meiner Rolle als Autor tätig ist.

Zu guter Letzt gebührt natürlich meiner Familie mein Dank, die es erträgt, wenn ich stunden- und tagelang von Entdeckungen erzähle, die vielleicht nicht ganz so interessant sind, wie ich es im jeweiligen Moment für absolut offensichtlich halte.

Geleitwort

Als mir Joachim Baumann erzählte, dass er ein Buch über Gradle schreiben wolle, habe ich mich sehr gefreut. Dies aus mehreren Gründen: Joachim Baumann ist ein erfahrener Autor, der zuvor schon sehr gute Bücher geschrieben hat. Zudem ist Joachim schon lange mit der Gradle-Technologie vertraut. Er war unter anderem Teilnehmer an einem der ersten Gradle-Expertentrainings in Deutschland, die ich damals noch selber gehalten habe. Schließlich erscheint das Buch in einem Verlag, den ich schon lange für seine hohe Qualität schätze.

Das Buch bietet eine hervorragende Einführung in Gradle, die auch die Implementierung komplexer Builds umfasst. Anhand einer gelungenen Mischung aus Codebeispielen und Text werden sprachlich und didaktisch gekonnt die Konzepte von Gradle erklärt.

Das Buch kommt zur richtigen Zeit. Im Java-Bereich verzeichnet Gradle schon seit Längerem ein rasantes Wachstum. Nun ist Gradle seit Kurzem auch das neue Standard-Buildsystem für Android-Applikationen und -Bibliotheken. Erstklassige Unterstützung gibt es bald auch für C und C++. Das wird die zunehmende Zahl der Gradle-Benutzer nochmals stark ansteigen lassen.

Ich wünsche allen Leserinnen und Lesern viel Freude mit diesem Buch und mit Gradle. Ich bin überzeugt, dass der Einsatz von Gradle zu erheblichen Produktivitätsgewinnen beim Entwickeln und Ausliefern Ihrer Software führen wird.

Hans Dockter
Gründer des Gradle-Build-Systems
Oakland im Juli 2013

Inhaltsverzeichnis

1 Einleitung

Das Thema Build-Management gibt es, seit es zu umständlich ist, alle Quelltexte eines Programms von Hand dem Compiler und in Folge dem Linker zu übergeben, um ein Programm oder eine Bibliothek zu erzeugen. Die erste Lösung für das Problem waren Skripte, die systemabhängig waren und damit spezifisch für jede Systemplattform geschrieben werden mussten.

make

Das erste verbreitete Build-Management-Werkzeug

Das erste Build-Management-Werkzeug, das größere Verbreitung erlangte, war das Werkzeug *make*, das als Teil von Unix mitgeliefert wurde. Dieses verwendete eine Datei (das *Makefile*), die alle Schritte beschrieb, die zur Erzeugung des Programms nötig waren. Auf jedem System, das über das Programm *make* verfügte, konnte damit das Makefile ausgeführt werden. Der Erfolg von *make* war so groß, dass es für fast jede Plattform (inklusive MS Windows) eine Version des Programms gibt.

Was damit jedoch nicht gelöst war, war die Auflösung systemabhängiger Namen von Werkzeugen und Bibliotheken und der Orte, an denen sie auf spezifischen Systemen zu finden waren. Auch systemspezifische Compiler-Optionen wurden nicht erfasst.

Dies führte Anfang der 90er-Jahre zu der Entwicklung und Durchsetzung von weiteren Programmen wie zum Beispiel *autoconf* oder *imake* (Teil von X11), die diese systemspezifischen Werte bestimmen und ein systemspezifisches *Makefile* generieren konnten.

Ant

Als 1995 Java die Bühne betrat, war die Situation einigermaßen stabil, und zu Beginn wurden die zur Verfügung stehenden Werkzeuge zur Übersetzung von Java-Programmen verwendet. Sehr schnell stellte sich allerdings die Frage, ob ein in Java geschriebenes Build-Management-

System nicht vorteilhaft sein könnte, da das JDK für die Übersetzung ja ohnehin vorhanden sein musste.

Ant war richtungsweisend für Java-Umgebungen.

Es gab verschiedenste Ansätze im Java-Umfeld, erfolgreich und richtungsweisend war *Ant* (»Another neat tool«), das eine Beschreibung der Schritte für die Erzeugung des Java-Programms in Form einer XML-Datei akzeptierte (erstes Erscheinen mit der Version 1.1 in 2000). *Ant* setzte sich sehr schnell für alle Java-basierten Programme durch und machte die Komplexität von *make* und den zugehörigen Werkzeugen überflüssig. Außerdem konnte *Ant* durch *Extensions* zusätzliche Funktionalität erhalten.

> Ant entstand im Rahmen der ersten Referenzimplementierung eines Servlet-Containers durch Sun, die später unter dem Namen Apache Tomcat weiterentwickelt wurde.

Was aber *Ant* noch fehlte, waren Funktionen wie eine Abhängigkeits- und Versionsverwaltung für Java-Archive oder die Idee eines Build-Prozesses, der aus verschiedenen, klar getrennten Phasen besteht, um die nötigen Schritte zur Erzeugung des Resultats durchzuführen.

Maven

Maven adressiert fehlende Funktionalitäten und Konzepte in Ant.

Diese Hauptpunkte adressiert *Maven*, das als Build-Management-System sowohl einen Prozess in Form eines Lebenszyklus mit aufeinanderfolgenden Phasen definiert als auch ein ausgefeiltes System zur Verwaltung und Versionierung von Abhängigkeiten (Java-Archiven) hat, die bei Internet-Verbindung automatisch lokal heruntergeladen werden können (Erscheinen der Version 1.0 in 2004). Hierzu stellt Maven ein globales Archiv (das Maven-Repository, siehe [Maven-Repository]) zur Verfügung, in dem seit 2006 so gut wie alle öffentlichen Java-Archive zum Zugriff bereit liegen.

Um die Funktionalität zu erweitern, können Plug-ins für die verschiedenen Phasen registriert werden. Maven führt dann die jeweilige Implementierung in der entsprechenden Phase aus.

Maven verfolgt einen deklarativen Ansatz.

Maven folgt anders als *Ant* einem deklarativen Paradigma: Anstatt zu sagen, *wie* etwas gemacht werden soll, reicht es aus zu sagen, *was* erreicht werden soll. Hierfür wird eine Spezifikation in Form einer XML-Datei verwendet (das POM oder Project Object Model). Damit dies funktioniert, muss es Konventionen geben, die für die Ausführung der Schritte verwendet werden. Ein Beispiel hierfür sind die Orte, an denen bei *Maven* Quelltexte (`src/main/java`) und Tests (`src/test/java`) zu finden sind. Auch *Maven* bietet die Möglichkeit, die Funktionalität

durch *Plug-ins* zu erweitern, die ihre jeweils eigenen Konventionen mitbringen.

> Da Maven sein Leben als Werkzeug im Umfeld des Apache-Projekts Jakarta Alexandria begann und in Folge für alle Apache-Projekte eingesetzt werden sollte, bilden die durch Maven definierten Pfade die Vorgaben für Apache-Projekte ab.

Ein großer Nachteil von *Maven* ist hierbei, dass sowohl der Prozess als auch die Konventionen sehr rigide sind. Das führt zwar dazu, dass jedes mit *Maven* aufgesetzte Projekt gleich aussieht, es sorgt aber auch dafür, dass das Build-Management früher oder später mit der Infrastruktur kollidiert.

Um das Problem der Abhängigkeits- und Versionsverwaltung bei *Ant* zu adressieren, wurde das Werkzeug *Ivy* implementiert, das seit 2006 ein Apache-Projekt und inzwischen ein Subprojekt von *Ant* ist.

Wir haben also auf der einen Seite das Werkzeug *Ant* (zusammen mit *Ivy*), das sehr große Freiheit, aber keine Unterstützung des Prozesses bietet und eine imperative Beschreibung aller Schritte erfordert.

Auf der anderen Seite haben wir *Maven*, das einen Prozess definiert und deklarative Beschreibung der Schritte durch eine Menge von Konventionen erlaubt. Dies wird allerdings dadurch erkauft, dass *Maven* sehr rigide Vorgaben macht, die das Build-Management früher oder später sehr problematisch machen.

Gradle

Gradle ist eine Antwort auf die Schwächen von Ant und Maven.

Eigentlich würden wir uns für das Build-Management ein Werkzeug wünschen, das die Funktionalität von *Maven* und den deklarativen Ansatz mit der Freiheit von *Ant* verbindet. Und genau dies tut *Gradle*.

Gradle bietet einen deklarativen Ansatz mit vernünftigen Konventionen, die leicht zu ändern sind, und durch die Integration von *Ivy* und *Maven-Repositories* exzellente Verwaltung der Abhängigkeiten und betrachtet *Ant* als einen gleichwertigen Partner, der voll integriert ist.

De facto ist jede Build-Datei für *Gradle* ein Groovy-Skript, das mit den *Gradle*-spezifischen Befehlen angereichert ist. Dies erlaubt prinzipiell die volle Programmierbarkeit des Build-Skriptes. Dies birgt natürlich die Gefahr, dass das Build-Skript, das eigentlich nur den Build beschreiben soll, durch zu viel Quelltext an Lesbarkeit verliert.

Deshalb sollten größere Teile Quelltext grundsätzlich in Plug-ins ausgelagert werden. Dies wir dadurch unterstützt, dass Plug-ins in Gradle sehr einfach zu schreiben und zu verwenden sind.

1.1 Grundsätzliche Aufgaben eines Build-Management-Werkzeugs

Historisch gesehen war die Aufgabe eines Build-Management-Werkzeugs sehr einfach, nämlich die Übersetzung von Quelltexten in der richtigen Reihenfolge und der darauf folgende Aufruf des Linkers, um ein ausführbares Programm oder eine Bibliothek zu erzeugen.

Funktionalitäten eines modernen Build-Management-Systems

Das gehört auch heute noch zu den häufigsten Aufgaben. Zusätzlich wünschen wir uns aber folgende Funktionalitäten:

- Checkout der Quelltexte aus dem Versionsverwaltungssystem und eventuelles Setzen von Tags. Dies kann auch gut in einen Build-Server wie Jenkins ausgelagert werden.
- Ausführen verschiedener Arten von Tests. Entsprechend ihrer Art müssen diese an verschiedenen Stellen im Ablauf des Builds durchgeführt werden. Beispiele sind:
  - Unit-Tests für einzelne Klassen
  - Modul- oder Komponententests
  - Integrationstests
  - Funktionale und nichtfunktionale Systemtests
  - Automatisierte Akzeptanztests
- Detaillierte Test-Reports, die die Testergebnisse übersichtlich zusammenfassen.
- Packen des Resultats. Dies kann allein im Java-Umfeld z.B. eine Jar-, War- oder Ear-Datei sein, je nach Art der Applikation beziehungsweise Bibliothek.
- Transfer des Resultats zu den verschiedenen Testsystemen und Ausführung entsprechender Testarten. Auch dies ist eine Aufgabe, die im Rahmen einer Build-Pipeline gerne durch andere Werkzeuge übernommen wird.
- Unterstützung für polyglotte Projekte, also Projekte, die mehr als eine Sprache und möglicherweise mehr als eine Ausführungsumgebung verwenden.
- Erzeugung von Dokumentation und Release Notes.

Je nach persönlichem Geschmack könnte man hier noch die Provisionierung und Konfiguration entfernter Systeme hinzufügen, um zum Beispiel bei dem Test einer Webapplikation sicherzustellen, dass auf

den Testsystemen bestimmte Versionen des Applikations- oder Webservers installiert sind. Allerdings gibt es Werkzeuge wie Chef oder Puppet, die auf dieses Thema spezialisiert sind und damit in den meisten Fällen die bessere Wahl darstellen (siehe [Chef-Website], [Puppet-Website]).

Da zudem Gradle diese Werkzeuge direkt aus dem Build anstoßen kann, haben wir damit die bestmögliche Trennung der Aufgaben bei gleichzeitiger Unterstützung aller benötigten Funktionalität.

Wünsche an ein modernes Build-Management-System

Die Erwartungshaltung, die wir an ein modernes Build-Management-Werkzeug haben, ist, dass die aufgelistete Funktionalität durch folgende Eigenschaften unterstützt wird.

- Explizite Unterstützung des Workflows, der durch das Build-Management-System implementiert wird.
- Leichte Anpass- und Erweiterbarkeit des Workflows, um sich an lokale Vorgehensmodelle anpassen zu können.
- Leichte Lesbarkeit und selbsterklärende Notation der Sprache des Build-Skripts.
- Verwendung von Konventionen, um vernünftige Vorgabewerte für die verschiedenen Schritte des Workflows zu haben (wie zum Beispiel die schon erwähnten Orte, an denen sich die Quelltexte befinden).
- Leichte Änderbarkeit der Vorgabewerte für die Anpassung an die lokale Umgebung.
- Inkrementeller Build, der erkennt, welche Artefakte schon erzeugt sind.
- Parallelisierung von unabhängig ausführbaren Schritten des Workflows, um die Wartezeit so kurz wie möglich zu halten.
- Zugriff auf alle Artefakte, die durch den Build erzeugt wurden.
- Status-Reports, die den aktuellen Zustand des Builds zusammenfassen.

Grundsätzlich ist der Zweck eines Build-Management-Systems aber nicht auf die Erzeugung von Programmen beschränkt. Ein Beispiel für eine völlig andere Verwendung eines Build-Management-Systems ist das Schreiben eines Buchs, dessen einzelne Kapitel zusammengefügt werden, wobei die Einhaltung bestimmter Regeln geprüft wird (zum Beispiel die Verwendung bestimmter Formate) und ein Inhaltsverzeichnis generiert wird, um zum Schluss gedruckt zu werden. Ein weiteres Beispiel ist das Generieren von Vortragsfolien, die automatisch mit den neuesten Unterlagen aus der Marketing-Abteilung angereichert werden. Auch in diesen Fällen ist ein Build-Management-System sehr wertvoll und hilfreich.

1.2 Gradle – Kurzübersicht

Gradle ist ein Build-Management-Werkzeug mit einem klaren Fokus: Es soll das Thema für den Benutzer so einfach wie möglich machen, ohne dabei die Möglichkeiten einzuschränken.

Um das zu tun, folgt Gradle zwei Prinzipien:

- Convention over Configuration – wann immer es einen sinnvollen Vorgabewert gibt, wird dieser verwendet. Nur wenn hiervon abgewichen wird, muss überhaupt etwas konfiguriert werden. Dies sorgt für einen sehr deklarativen Stil des Build-Skripts.
- Don't Repeat Yourself (DRY) – Gradle folgt soweit wie möglich dem Prinzip, dass man sich nicht wiederholen soll, um überflüssige Redundanz und damit potenzielle Fehlerquellen zu vermeiden. Dies sorgt unter anderem für eine sehr präzise, kurze Syntax in den Build-Skripten.

Diese beiden Herangehensweisen führen dazu, dass für Gradle im Regelfall vergleichsweise kurze Build-Skripte ausreichen, um auch komplexere Build-Umgebungen zu beschreiben. Dies führt zu einer geringeren Fehlerrate und zu leichterer Wartbarkeit der Skripte, was wiederum dafür sorgt, dass auch normale Entwickler die Skripte verstehen und erweitern können.

Gradle verwendet ein dynamisches Modell.

Gradle erzeugt ein dynamisches Modell des Workflows, der für den Build zuständig ist. Auch wenn dies wie ein kleiner Schritt wirkt, ist es eine sehr starke Abkehr von dem statischen Lebenszyklus, den zum Beispiel Maven implementiert. Wir können damit in unserem Build-Skript Einfluss darauf nehmen, wie die verschiedenen Phasen des Builds ablaufen, können eigene Phasen hinzufügen und können sogar entscheiden, welche Phasen überhaupt benötigt werden.

Gradle hat eine Konfigurations- und eine Ausführungsphase.

Um dieses dynamische Modell erzeugen zu können, unterscheidet Gradle zwei prinzipielle Verarbeitungsphasen. Die erste Phase, die Konfigurationsphase, verarbeitet das Build-Skript, interpretiert die Inhalte und passt das Modell entsprechend der Anweisungen im Skript an. In der zweiten Phase, der Ausführungsphase, werden die einzelnen Schritte des Builds anhand des erzeugten Modells abgearbeitet.

Gradle verwendet einen gerichteten azyklischen Graph.

Für die interne Darstellung des Modells verwendet Gradle einen gerichteten azyklischen Graph (DAG, für *directed acyclic graph*), in dem die einzelnen Ziele, die Tasks von Gradle, die Knoten darstellen.

Die einzelnen Knoten werden durch Kanten verbunden, die die Abhängigkeiten darstellen. Diese Kanten symbolisieren, welcher Task welchem anderen Task folgen kann.

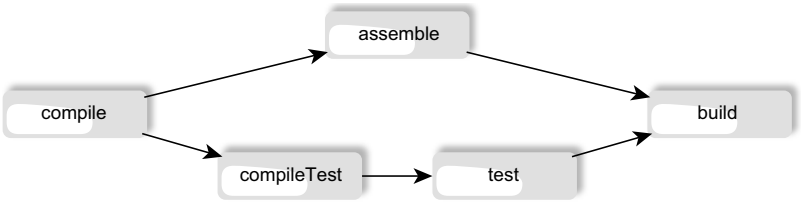

Abb. 1–1
Beziehung verschiedener Tasks (vereinfachtes Beispiel)

Dieser Graph beschreibt für uns Menschen sehr einfach verständlich, wie die einzelnen Schritte des Workflows abgearbeitet werden, um zum Ziel zu gelangen. Man muss nur den Pfeilen folgen. Um aber die Frage zu bearbeiten, welche Tasks in welcher Reihenfolge abgearbeitet werden müssen, um einen bestimmten Task zu erreichen, ist dieser Graph aber nicht wirklich gut geeignet (unter der Voraussetzung, dass wir den Kanten in Pfeilrichtung folgen müssen).

Eine andere Form der Darstellung, in der die Richtung der Kanten umgedreht ist, macht aber die Verarbeitung deutlich einfacher. In dieser Variante des Graphs symbolisieren die Kanten die Abhängigkeitsbeziehung, zeigen also auf die Tasks, die ausgeführt werden müssen, bevor der aktuelle Task bearbeitet werden kann.

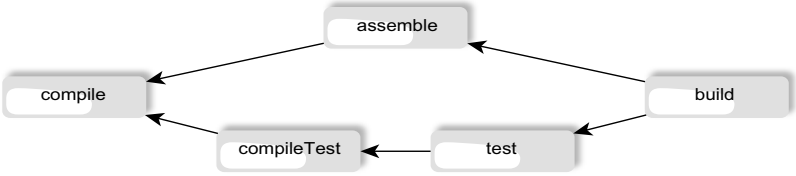

Abb. 1–2
Beispiel für einen Abhängigkeitsgraph (vereinfacht)

Bestimmung der Ausführungsreihenfolge in einem DAG

Um in diesem Graph zu bestimmen, welche Tasks (Knoten) als Vorbedingung ausgeführt werden müssen, um einen bestimmten Task ausführen zu können, müssen wir nur allen Pfeilen ausgehend von diesem Task folgen (so lange bis wir bei Knoten angelangt sind, die keine weiteren Kanten mehr haben) und erhalten damit die Antwort.

Betrachten wir zum Beispiel den Knoten *test*, so sehen wir, dass, um ihn bearbeiten zu können, zuerst der Knoten *compileTest* erfolgreich abgeschlossen werden muss, der wiederum vom Erfolg der Ausführung des Knotens *compile* abhängt.

Dies ist eine topologische Sortierung, und es gibt Algorithmen, die einen gerichteten azyklischen Graphen sehr einfach in eine solche Sortierung überführen. Mit dieser topologischen Sortierung lassen sich alle Fragen, die sich in einem Build-Prozess ergeben, elegant beantworten. Insbesondere die Ausführungsreihenfolge lässt sich sehr leicht bestimmen. Gradle verwendet genau diese Repräsentation für die interne Verwaltung der Tasks.

Die Idee ist übrigens nicht neu, auch *make* verwendet einen DAG, um dieses Problem zu lösen.

1.3 Migrationspfade von anderen Build-Management-Werkzeugen

Eine wichtige Frage im Zusammenhang mit Gradle ist die Frage, wie man von den etablierten Build-Management-Systemen und den teilweise sehr langen und komplexen Build-Szenarien zu Gradle wechseln kann. In vielen Fällen fühlt man sich in einem *Ant*- oder *Maven*-Build seit Jahren gefangen, hat eine sehr hohe und schwer beherrschbare Komplexität und fürchtet potenziell hohe Investitionen, um das Werkzeug zu wechseln.

Aus diesem Grund bietet Gradle für *Ant* und *Maven* verschiedene Migrationsmöglichkeiten an.

Migration von Ant-Skripten

Gradle integriert Ant nicht nur vollständig, sondern ist auch in der Lage, die Build-Skripte von Ant vollständig zu lesen und die Ant-Targets als Gradle-Tasks zu integrieren. Dies macht eine kontrollierte Migration in kleinen Schritten unproblematisch.

Gradle-Tasks entsprechen am ehesten den Targets in Ant und nicht den Ant-Tasks. Dies kann am Anfang für etwas Verwirrung sorgen.

In der Integration von Gradle und Ant führt dies dazu, dass Gradle-Tasks in Ant als Ant-Targets verwendet werden können und Ant-Targets in Gradle als Gradle-Tasks zur Verfügung stehen. Damit bietet Gradle eine vollständige Interoperabilität an, was natürlich bei der Einführung von Gradle in Ant-Umgebungen sehr hilfreich ist.

Migration von Maven-Skripten

Die Migration von Maven zu Gradle führt über das MavenImport-Plug-in von Gradle, das seit der Version 1.1 zur Verfügung steht. Hiermit ist es möglich, Maven-Build-Strukturen in erste Versionen von Gradle-Skripten umzuwandeln. Hier ist generell noch Handarbeit notwendig, aber gerade die Entwicklung dieses Plug-ins schreitet rasch voran, was in der Zukunft deutlich einfachere Migrationen verheißt. In einfachen Fällen ist es aber nicht einmal notwendig, dieses Plug-in zu verwenden, da Gradle in der Voreinstellung den meisten Konventionen von Maven folgt und damit sehr einfach anstatt des Maven-Builds verwendet werden kann.

In vielen Fällen können die entstehenden Skripte noch weiter vereinfacht werden, so dass nur sehr kurze und simple Skripte übrig bleiben.

Da Gradle zusätzlich eine vollständige Kompatibilität zu allen gängigen Repository-Implementierungen anbietet, kann eine derartige Infrastruktur mit Gradle unverändert weiterverwendet werden.

1.4 Installation

Die Installation von Gradle ist völlig problemlos. Wenn Sie sich auf Ihrem System vollständig auskennen, sollte die folgende Kurzanleitung genügen. Sicherheitshalber finden Sie detailliertere Anleitungen für die verbreitetsten Systeme im Anschluss.

In allen Fällen installieren wir Gradle einfach in einem Verzeichnis `gradle`. Wenn Sie häufig die Version von Gradle wechseln wollen (zum Beispiel weil Sie Build-Skripte mit verschiedenen Versionen testen müssen), dann empfiehlt es sich, versionsspezifische Installationen mit entsprechenden Namen unterhalb eines Verzeichnisses `gradle` einzurichten. In diesem Fall können Sie durch einfaches Ändern der Variablen `GRADLE_HOME` die Version wechseln. In allen normalen Fälle reicht aber das beschriebene Vorgehen vollständig aus.

Gradle kann für Fälle, in denen Sie mit verschiedenen Versionen arbeiten, versionsspezifische Start-Dateien generieren, die in das Versionsverwaltungssystem gelegt werden können (dies ist der Gradle-Wrapper). Diese Dateien sind sogar in der Lage, bei Bedarf die spezifizierte Version von Gradle automatisch herunterzuladen.

1.4.1 Kurzanleitung

Die Installation von Gradle ist auf allen normalen Systemen sehr einfach, indem Sie den folgenden Schritten folgen:

- Laden Sie die Archivdatei mit der aktuellen Version herunter. Wählen Sie hierzu am besten die Version, die mit `all` bezeichnet ist (diese enthält neben den für die Ausführung nötigen Dateien alle Quelldateien). Sie finden sie auf der Download-Seite der Gradle-Website:

  ```
  http://www.gradle.org/downloads
  ```

- Entpacken Sie die Archivdatei (ein Zip-Archiv) an einem Ort Ihrer Wahl.
- Setzen Sie die Umgebungsvariable `GRADLE_HOME` auf das Installationsverzeichnis.

- Fügen Sie der Pfadvariablen Ihres Systems den Pfad `GRADLE_HOME/bin` hinzu.
- Wenn Sie später eine neuere Version installieren möchten, löschen Sie einfach das Installationsverzeichnis und verschieben die neue Version an die exakt gleiche Stelle. Die Variablen brauchen Sie dann nicht zu ändern. Alternativ ändern Sie den Wert der Variablen GRADLE_HOME. Eine dritte Variante wäre das Setzen eines symbolischen Links `current`, der auf die aktuelle Version zeigt.

> Tatsächlich wird die Umgebungsvariable `GRADLE_HOME` schon seit einiger Zeit von Gradle selbst nicht mehr benötigt. Aber viele andere Werkzeuge wie zum Beispiel Build-Server verwenden diese Variable zur Identifikation der zu verwendenden Gradle-Version, und ihre Verwendung macht gerade die Verwendung verschiedener Versionen von Gradle sehr viel einfacher.
>
> Wenn Sie auf diese Umgebungsvariablen verzichten wollen, ergänzen Sie einfach direkt Ihren Pfad mit `<Gradle-Installationsverzeichnis>/bin`.

Jetzt können Sie die Funktion von Gradle mit einem Aufruf testen. Geben Sie dazu auf der Kommandozeile `gradle –v` ein:

Listing 1–1
Erster Test für die Verifikation der Gradle-Installation

```
$ > gradle –v

------------------------------------------------------------
Gradle 1.x
------------------------------------------------------------

Gradle build time: Montag, 28. Januar 2013 03:42 Uhr UTC
Groovy: 1.8.6
Ant: Apache Ant(TM) version 1.8.4 compiled on May 22 2012
Ivy: 2.2.0
JVM: 1.7.0_17 (Oracle Corporation 23.7-b01)
OS: Mac OS X 10.8.2 x86_64


$ >
```

Diese Kommandozeile gibt die Versionen sämtlicher integrierter Software-Pakete aus. Wenn dies funktioniert, können wir davon ausgehen, dass Gradle erfolgreich installiert ist.

1.4.2 Windows

Laden Sie zuerst das Zip-Archiv der aktuellen Version herunter (mit dem Browser oder dem Kommandozeilenwerkzeug Ihrer Wahl). Diese finden Sie auf der Download-Seite der Gradle-Website: