7 Erweiterung von Gradle

Gradle bietet uns sehr viele Tasks und Plug-ins für die verschiedensten Funktionen. Außerdem gibt es eine große Anzahl von Plug-ins von Dritten, die weitere Funktionen anbieten.

Trotzdem ist es auch für uns interessant zu verstehen, wie diese Erweiterungen geschrieben werden. Im einfachsten Fall erlaubt uns das einen weiteren Einblick in die Funktionalität von Erweiterungen, in vielen Fällen erlaubt uns das Verständnis die bessere Strukturierung unserer eigenen Builds durch Auslagerung von Funktionalität aus dem eigentlichen Build-Skript.

Dieses Kapitel ist zwar dasjenige mit dem komplexesten Inhalt, aber die durch Gradle angebotenen Möglichkeiten sind so einfach zu verstehen, dass es völlig unproblematisch ist, eigene Erweiterungen zu schreiben.

7.1 Arten von Erweiterungen

Gradle bietet als eine zentrale Abstraktion den Task, der als klaren Zweck die Beschreibung eines Schrittes in unserem Build-Prozess hat. Gradle bietet uns von Haus aus eine Menge verschiedener Typen an. Bisher haben wir diese Tasks verwendet und für unsere eigenen Zwecke konfiguriert.

Ein Task ist aber nichts anderes als eine von der Klasse `DefaultTask` abgeleitete Java- oder Groovy-Klasse, die wir sehr einfach auch selbst schreiben können. Dies bietet sich an, sobald wir Tasks in unserem Projekt mehrere Male auf die gleiche oder ähnliche Art konfigurieren beziehungsweise mit eigener Funktionalität erweitern.

Die zweite Art der Erweiterung, das Plug-in, hat einen völlig anderen Zweck. Im Gegensatz zum Task spielt ein Plug-in keine Rolle im Build-Prozess, sondern erweitert und modifiziert diesen in einer plug-in-spezifischen Form. Hierzu kann auch die Einführung neuer Domänenobjekte gehören. Ein Beispiel, das wir schon kennengelernt haben,

ist das SourceSet, das durch das Java-Plug-in eingeführt wird (genauer das Java-Base-Plug-in).

Auch ein Plug-in ist eine Java- oder Groovy-Klasse, die eine Schnittstelle `Plugin<Project>` implementiert. Diese Schnittstelle enthält eine Methode `apply()`, und dies ist die Methode, die aufgerufen wird, wenn wir ein Plug-in in unserem Build-Skript laden.

7.2 Orte für Erweiterungen

7.2.1 Das Build-Skript

Der erste Ort, an dem wir Erweiterungen platzieren können, ist im Build-Skript selbst. Dies ist der offensichtliche Ort für erste Experimente. Der Vorteil ist, dass Gradle innerhalb des Build-Skripts sämtliche API-Klassen zur Verfügung stellt, die wir in anderen Fällen von Hand importieren müssen. Der offensichtliche Nachteil ist, dass die Erweiterungen nur innerhalb des Build-Skripts zur Verfügung stehen.

Es gibt aber ein großes Problem mit diesem Ansatz, das vielleicht im ersten Moment gar nicht so offensichtlich ist. Das Build-Skript soll unseren Build möglichst klar und leicht lesbar beschreiben, ein Aspekt, der in Gradle sehr großen Einfluss auf das Design hat.

> Gradle folgt sehr stark einem der Grundprinzipien vernünftigen Designs, der separation of concerns (Trennung der Anforderungen). Diese fordert nichts anderes, als dass man keine verschiedenen Anforderungen oder Aufgaben innerhalb des Quelltextes vermischt. Die erste Erwähnung dieses Prinzips finden Sie in [Dijkstra 1983].

Wenn wir nun den Quelltext unserer Erweiterungen im normalen Build-Skript belassen, dann haben wir das genaue Gegenteil dessen, was wir haben wollen, ein Build-Skript, das voll von Quelltexten ist, die erst verstanden sein müssen, damit klar ist, was der tatsächliche Zweck ist.

7.2.2 Externe Skripte

Die zweite Möglichkeit der Strukturierung ist das Einbinden externer Skripte, das wir im Rahmen der Einbindung des Emma-Plug-ins bereits kennengelernt haben.

Mit `apply from: 'datei.gradle'` können wir die Datei `datei.gradle` direkt einbinden. Der Vorteil ist, dass wir die gleichen Rahmenbedingun-

gen wie im Build-Skript selbst vorfinden. Da wir das Skript von beliebigen Build-Skripten aus einbinden können, steht uns die Funktionalität auch über die Grenzen des eigentlichen Build-Skripts zur Verfügung.

Es gibt auch hier mehrere Nachteile:

- Wir müssen den Ort, an dem die Datei zur Verfügung gestellt wird, kennen und angeben.
- Gradle führt hier kein Caching der Dateien durch. Während dies im lokalen Dateisystem nicht relevant ist, stellt dies bei entfernten Quellen, die zum Beispiel durch einen URL referenziert werden, ein Problem dar. Neben der deutlich längeren Zeit für einen solchen Zugriff machen wir uns damit abhängig von der Verfügbarkeit des Netzes und des Servers, auf dem das Skript liegt.
- Der größte Nachteil ist aber, dass externe Skripte nicht versioniert sind. Sie sind also davon abhängig, dass ein Skript, das Sie verwenden, auch in zwei Jahren noch die gleiche Semantik hat und mit der dann von Ihnen verwendeten Gradle-Version funktioniert.

Diese Nachteile führen dazu, dass der Build um ein Vielfaches fragiler ist und das Scheitern des Builds deutlich schwerer nachvollziehbar sein kann. Damit sollte auch diese Variante nicht dauerhaft verwendet werden, sondern nur während der Entwicklung der Erweiterungen.

Sollten Sie die externe Datei unter Ihrer Kontrolle haben, dann bietet sich die nächste Variante zur klareren Strukturierung an.

7.2.3 Das Verzeichnis `buildSrc`

Bei jedem Start prüft Gradle, ob ein Verzeichnis `buildSrc` existiert (in Multiprojekt-Builds im Verzeichnis des Wurzelverzeichnisses). Wenn dieses Verzeichnis existiert, dann behandelt Gradle dies als normales Projektverzeichnis für ein Groovy-Projekt (mit `src/main/`- und `src/test`-Verzeichnissen), übersetzt und testet alle Quellen und stellt diese dem Build-Skript zur Verfügung. Wir können in diesem Verzeichnis ein normales Build-Skript `build.gradle` verwenden. Dies ist insbesondere dann notwendig, wenn unsere Erweiterung ihrerseits Abhängigkeiten von anderen Bibliotheken besitzt.

Falls wir kein Build-Skript `build.gradle` zur Verfügung stellen, verwendet Gradle das folgende Build-Skript:

Listing 7–1
Das Build-Skript für das Verzeichnis `buildSrc`

```
apply plugin: 'groovy'

dependencies {
   compile gradleApi()
   groovy localGroovy()
}
```

Interessant sind hierbei die beiden Abhängigkeiten `gradleAPI()` und `localGroovy()`, die uns von Gradle zur Verfügung gestellt werden und die wir auch verwenden können, wenn wir ein eigenes Build-Skript für das Verzeichnis `buildSrc` schreiben. Die Methode `gradleAPI()` stellt die API-Klassen aus der lokalen Gradle-Installation zur Verfügung, `localGroovy()` die Groovy-Version von Gradle.

Anders als bei Erweiterungen im Build-Skript müssen wir in den hier abgelegten Quelltexten die von uns verwendeten API-Klassen selbst importieren. Auch wenn dies folgerichtig ist, sind die durch die fehlenden Importe entstehenden Fehler bei eigenen Erweiterungen, die wir aus unserem Build-Skript in dies Verzeichnis bewegen, ein typisches Problem, das allerdings leicht zu beheben ist.

Ein Nachteil ist auch hier die fehlende Versionierung. Dieser Nachteil wird allerdings dadurch relativiert, dass wir die volle Kontrolle über die Quellen haben. Trotzdem ist die nächste Variante mit einem Minimum an zusätzlichem Aufwand deutlich vorteilhafter.

7.2.4 Eigene Projekte

Von den Erweiterungen im Verzeichnis `buildSrc` ist es nur noch ein kleiner Schritt bis zum vollständigen eigenen Projekt. Tatsächlich ist es, wenn Sie das in Listing 7–1 angegebene Build-Skript verwenden, nicht mehr als das Kopieren des Verzeichnisses in ein eigenes Projektverzeichnis.

Als einzigen zusätzlichen Schritt müssen Sie noch das Hochladen des erzeugten Archivs in ein Repository oder Verzeichnis konfigurieren, so dass Sie auf die Erweiterung in Folge von beliebigen Build-Skripten aus zugreifen können.

7.2.5 Wahl der Variante

In Ihrem Build-Skript sollten Sie Quelltexte für Erweiterungen nur zur Zeit der Entwicklung haben, danach sollten Sie sie auf jeden Fall auslagern. Hierbei ist der erste Schritt eine externe Datei, in die Sie den Quelltext verschieben.

Wann immer Sie können, sollten Sie Ihre Erweiterungen in eigene Projekte auslagern oder, wenn Ihnen das zu aufwendig ist, dann zumindest in das Unterverzeichnis `buildSrc`. Da der Schritt von diesem Verzeichnis zu einem eigenen Projekt sehr klein ist und die damit verbundenen Vorteile der Versionierung über das Repository sehr hoch sind, sollten Sie eigentlich immer zu dieser Variante tendieren.

7.3 Eigene Tasks

In vielen Fällen reichen uns die Task-Typen aus, die Gradle und die verschiedenen Plug-ins zur Verfügung stellen. Es gibt aber Situationen, in denen wir in mehreren Tasks die immer gleichen Konfigurationen durchführen. In diesen Fällen ist es sehr einfach, einen eigenen Task-Typ zu definieren.

Hierzu deklarieren wir eine Klasse, die von `DefaultTask` oder einem anderen Task abgeleitet ist (in normaler Groovy-Manier). In diesem definieren wir beliebig Eigenschaften und Methoden, die in den abgeleiteten Klassen verwendet werden können.

7.3.1 Annotationen für unsere eigenen Task-Klassen

Zusätzlich stellt Gradle noch einige Annotationen zur Verfügung, um Eigenschaften und Methoden des Tasks zu markieren. Diese dienen dazu, Gradle mitzuteilen, dass die markierten Eigenschaften und Methoden Informationen oder Funktionalität enthalten, die für Gradle relevant ist. Gradle verwendet hierbei die Groovy-eigene Sicht auf Eigenschaften, die entweder durch Attribute oder aber durch Getter- und Setter-Methoden definiert werden können.

Die angebotenen Annotationen werden hauptsächlich für Eigenschaften zur Referenzierung von Eingabe- und Ausgabedateien verwendet, entweder in Form von normalen Java-`File`-Objekten oder in Form von `FileCollection`-Objekten.

Annotation für Dateien	Bedeutung
@InputFile	Die markierte Eigenschaft enthält eine Eingabedatei als `File`-Objekt.
@InputFiles	Die markierte Eigenschaft enthält eine `FileCollection` von Eingabedateien.
@InputDirectory	Die markierte Eigenschaft enthält ein Verzeichnis von Eingabedateien als `File`-Objekt.
@OutputFile	Die markierte Eigenschaft enthält eine Ausgabedatei als `File`-Objekt.
@OutputFiles	Die markierte Eigenschaft enthält eine `FileCollection` von Ausgabedateien.
@OutputDirectory	Die markierte Eigenschaft enthält ein Verzeichnis von Ausgabedateien als `File`-Objekt.
@OutputDirectories	Die markierte Eigenschaft enthält eine `FileCollection` von Ausgabeverzeichnissen.

➔

Tab. 7–1
Annotationen für eigene Tasks

Weitere Annotationen	Bedeutung
@Input	Wenn eine mit dieser Annotation markierte Eigenschaft den Wert ändert, bewirkt dies, dass Gradle den Task nicht mehr als aktuell (UP-TO-DATE) ansieht.
@Nested	Diese Annotation markiert ein normales Java-Objekt (Java Bean), dessen Eigenschaften die in dieser Tabelle aufgeführten Annotationen haben. Gradle wertet die Annotation und die im Java-Objekt enthaltene Information aus.
@Optional	Wenn wir die Annotationen dieser Tabelle verwenden, dann erwartet Gradle verpflichtend Werte in den annotierten Eigenschaften. Wenn wir diese Annotation zusätzlich benutzen, dann betrachtet Gradle die Werte als optional und meldet keinen Fehler, wenn die entsprechende Eigenschaft keinen Wert enthält.
@SkipWhenEmpty	Diese Annotation verwenden wir zusammen mit den Annotationen für die Markierung von Eingabedateien. Wenn eine mit dieser Annotation markierte Eigenschaft eine leere `FileCollection` oder ein leeres Verzeichnis enthält, wird der Task nicht ausgeführt.
@TaskAction	Eine mit dieser Annotation markierte Methode wird als Task-Action an die Liste der auszuführenden Aktionen angehängt.

7.3.2 Ein einfaches Beispiel

Ein Beispiel demonstriert, wie wenig Aufwand die Erzeugung eines eigenen Task-Typs ist.

Listing 7–2
Erzeugung eigener Task-Typen

```
class HalloTask extends DefaultTask {
   def adressat = 'nobody'

   def setAdressat(neuerAdressat) {
      adressat = neuerAdressat
      if(adressat == "Welt")
         adressat = "Universum"
   }

   @TaskAction
   def helloAdressat() {
      println "Hallo $adressat"
   }
}

task halloWelt ( type : HalloTask) {
   adressat "Welt"
}
```

Wir erzeugen zunächst eine neue Groovy-Klasse, die von dem Typ `DefaultTask` abgeleitet ist. Wir geben ihr eine Eigenschaft `adressat` und

eine Methode `setAdressat()`. Wenn wir die Eigenschaft `adressat` in Folge setzen, wird diese Methode aufgerufen anstelle einer direkten Zuweisung. In der Methode setzen wir den übergebenen Wert, und nur wenn dieser den Wert `Welt` hat, dann ersetzen wir ihn durch den Wert `Universum`.

Dann definieren wir eine zweite Methode `helloAdressat()`, die wir als `TaskAction` annotieren. Damit übernimmt Gradle diese Methode automatisch in die Liste der Aktionen, die jeder abgeleitete Task ausführt.

Zum Schluss deklarieren wir einen Task `halloWelt`, der vom Typ `HalloTask` abgeleitet ist. In der zugehörigen Konfigurations-Closure setzen wir den Wert der Eigenschaft `adressat` auf `Welt`. Dies führt zum Aufruf der `setAdressat()`-Methode und damit der Ersetzung der Zeichenkette. Die Ausführung des Tasks `halloWelt` führt die annotierte Aktion aus und gibt die entsprechende Zeichenkette aus.

```
> gradle -q halloWelt
Hallo Universum
>
```

Listing 7–3
Ergebnis des Build-Skripts

7.3.3 Ein komplettes Beispiel

Im Rahmen der Erzeugung eigener Ant-Tasks haben wir bereits ein etwas komplexeres Beispiel eines Tasks kennengelernt, um Python-Skripte zu übersetzen (siehe Listing 7–4).

Hier verwenden wir auch die Annotationen zur Kennzeichnung der Eingabe- und Ausgabedateien, um Gradle bei der Bestimmung der Aktualität unseres Tasks zu unterstützen.

```
configurations {
   jython
}

repositories {
   mavenCentral()
}

dependencies {
   jython group: 'org.python', name: 'jython', version: '2.5.+'
}

class JythonC extends DefaultTask{
   @InputDirectory
   File sourceDirectory = project.file("src/main/python")
   @OutputDirectory
   File outputDirectory =
      project.file("$project.buildDir/classes/main")
```

Listing 7–4
Der Übersetzung mit jython als eigene Task-Klasse

```
    @InputFiles
    Configuration jythonClasspath = project.configurations.jython

    @TaskAction
    public void compile(){
        ant.taskdef (name: 'jythonC',
                     classname: 'org.python.util.JycompileAntTask',
                     classpath: jythonClasspath.asPath)
        logging.level = LogLevel.INFO
        ant.jythonC(srcdir: sourceDirectory,
                     destdir: outputDirectory)
    }
}

task jythonC(type: JythonC){
    // sourceDirectory = file("src/main/python")
    // outputDirectory = file("$buildDir/classes/main")
    // jythonClasspath = configurations.jython
}

task jythonRun (type: JavaExec) {
    classpath configurations.jython.asPath, jythonC
    main "HalloWelt\$py"
}
```

Wir erzeugen wieder eine eigene Konfiguration `jython`, verwenden das Repository Maven Central und spezifizieren die Abhängigkeit von den `Jython`-Bibliotheken.

Dann definieren wir unseren eigenen Task-Typ `JythonC`, abgeleitet vom Typ `DefaultTask`. Wir deklarieren drei Eigenschaften, die wir für die Synchronisation mit Gradle als Eingabe- und Ausgabedateien annotieren.

Das `sourceDirectory` ist als Eingabeverzeichnis annotiert und enthält die Python-Quelldateien, `outputDirectory` spezifiziert das Verzeichnis für die Ausgabedateien, und zusätzlich definieren wir noch mit der Eigenschaft `jythonClasspath`, dass die in der Konfiguration spezifizierten Abhängigkeiten als Eingabedateien zu sehen sind. Auf diese Weise wird der Task auch dann als nicht mehr aktuell betrachtet, wenn sich die Abhängigkeit (zum Beispiel durch eine neue Version der Bibliothek) ändert.

Dann folgt die auszuführende Aktion als Groovy-Methode, die mit der Annotation `@TaskAction` markiert ist, so dass Gradle diese automatisch an die Liste der auszuführenden Actions anhängt.

Jetzt können wir diesen Task-Typ `JythonC` genau wie die durch Gradle vorgegebenen Task-Typen verwenden. Wir definieren einen Task `jythonC`, der unseren Task-Typ verwendet. Da die vorgegebenen

Werte für die verschiedenen Verzeichnisse den von uns benötigten entsprechen, brauchen wir hier nichts mehr zu ändern.

Wenn wir im nächsten Schritt unseren Task in das Verzeichnis `buildSrc` verschieben, dann können wir uns mit dem Task prinzipiell in einer Multiprojektumgebung befinden. In dieser kann (und wird) das aktuelle Projekt, in dem der Task verwendet wird, ein anderes sein als das, in dem der Task ursprünglich verwendet wurde. Daher ist es also prinzipiell sinnvoll, die Werte auf jeden Fall explizit zu setzen.

Die an dieser Stelle zu bevorzugende Alternative ist eine defensive Programmierung, in der wir die Werte als optional markieren und erst in der Task-Action prüfen, ob die Werte definiert sind. Falls diese ungesetzt sind, können wir auf das momentan aktive Teilprojekt zugreifen und die Werte entsprechend setzen.

Allgemein gilt, dass wir mit Gradle so konservativ wie möglich programmieren sollten, um die Nutzer unserer Erweiterungen so weit wie möglich zu unterstützen. Hierzu gehört auch, verschiedene Möglichkeiten für das Setzen unserer Eigenschaften vorherzusehen. In unserem Fall wäre dies die Möglichkeit, die Ein- und Ausgabedateien sowohl als Zeichenkette als auch als `File`-Objekt oder als `FileCollection` setzen zu können.

7.3.4 Unser Task im Verzeichnis `buildSrc`

Die bisher gezeigten Beispiele für eigene Tasks platzierten die Tasks im Build-Skript `build.gradle`. Der nächste Schritt ist, den Task in das Verzeichnis `buildSrc` zu verschieben.

Die Task-Klasse

Wir legen die Datei `JythonC.groovy` für unsere Task-Klasse `de.gradlebuch.JythonC` im Verzeichnis `buildSrc/src/main/groovy/de/gradlebuch/` ab. Natürlich könnten wir die Klasse problemlos auch ohne Paket definieren, aber die Verwendung eigener Pakete sollte jedem Entwickler in Fleisch und Blut übergegangen sein.

Die Definition der Klasse sieht sehr ähnlich aus wie bei der direkten Definition im Build-Skript, wir haben aber jetzt zusätzlich die Paket-Deklaration und die Import-Befehle für die Gradle-Klassen, die wir verwenden.