Teil V

Bauen und Testen

Warum Build Tools?

Für das Programmierpraktikum:

- Einfaches Ausführen der Tests.
- Herunterladen der JUnit Bibliothek.
- Einfaches Einrichten einer IDE.

Bei größeren Projekten wird Bauen oft komplizierter ...

Warum Build Tools?

Typischer Ablauf ohne Build Tools:

1. Bibliotheken Guava und Gson als jar-Datei herunterladen.
2. Eclipse Projekt-Einstellungen öffnen.
   a. Die zwei jar-Dateien als Bibliothek einfügen.
   b. Ordner `source` und den Ordner `utils` als Ordner für Quelldateien markieren.
   c. JUnit Bibliothek aktivieren.
3. Bash-Script `init.sh` ausführen um weitere benötigten Dateien zu generieren.
4. Um das Programm zu starten, in Eclipse auf Ausführen klicken und die Klasse `harry.hacker.App` als Main-Klasse auswählen.

Build Tools

- Automatisches Bauen (wenn mehrere Schritte zum Bauen notwendig sind, zum Beispiel generierter Sourcecode)
  - Schritte in korrekter Reihenfolge ausführen
  - Schritte nur dann erneut ausführen wenn notwendig
  - Wiederholbare Ergebnisse, feste Prozesse
- Verwalten von Abhängigkeiten (externe Bibliotheken)
  - Genaue Spezifikation der Version
  - Automatisches Heruterladen von transitiven Abhängigkeiten
- Alle Entwickler können Ihren bevorzugten Editor verwenden.
- Gleicher Prozess bei Entwicklern und auf Build-Servern.

Beispiele für Build-Systeme

Build-Systeme für Java:

- Gradle (https://gradle.org/)
- Maven (https://maven.apache.org/)
- Ant (https://ant.apache.org/)

Build-Systeme für andere Sprachen:

- Make (meist C/C++, aber auch z.B. LaTeX)
- sbt (simple build tool; Scala)
- dotnet (C# und F#)

Beispiel: Gradle

Projekt-Beschreibung (`build.gradle`) ist ein Programm
(Programmiersprache: Groovy oder Kotlin)

Vorteile gegenüber Maven und Ant:

- Besser lesbar als XML-Konfiguration in Ant und Maven.
- Flexibler als Maven: Wird ein Build-Schritt nicht direkt unterstützt kann er programmiert werden. In Maven muss ein Plugin geschrieben werden.
- Weniger Konfiguration als Ant: Sinnvolle Standardeinstellungen
- Bessere Performance[1].

[1] Laut eigenen Angaben: https://gradle.org/gradle-vs-maven-performance/

Gradle: Tasks

Ein *Task* ist eine einzelne Aufgabe im Build.

Der Block nach `doLast` enthält die Aktionen des Tasks.

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

Ausführen auf der Konsole mit:

```
gradle hello
```

Gradle: Task Abhängigkeiten

Mit *dependsOn* wird festgelegt, dass ein anderer Task zuerst ausgeführt werden muss.

```
task generator() {
    doLast {
        def generatedFileDir = file("$buildDir/generated")
        generatedFileDir.mkdirs()
        for (int i=0; i<10; i++) {
            new File(generatedFileDir, "${i}.txt").text = i
        }
    }
}

task zip(type: Zip) {
    dependsOn generator
    from "$buildDir/generated"
}
```

Gradle: Inkrementelles Bauen

Der Task *zip* wird nur dann erneut ausgeführt, wenn sich die Eingaben oder Ausgaben geändert haben. (Gradle verwendet Prüfsummen, andere Tools wie *Make* oft einfach das Änderungsdatum der Eingaben)

Aber der Task *generator* wird jedes mal neu ausgeführt.
⇒ Eingaben und Ausgaben spezifizieren:

```
task generator() {
    def fileCount = 10
    inputs.property "fileCount", fileCount
    def generatedFileDir = file("$buildDir/generated")
    outputs.dir generatedFileDir
    doLast {
        generatedFileDir.mkdirs()
        for (int i=0; i<fileCount; i++) {
            new File(generatedFileDir, "${i}.txt").text = i
        }
    }
}
```

Quelle: https://blog.gradle.org/introducing-incremental-build-support

Gradle: Bibliotheken

Externe Bibliotheken werden automatisch heruntergeladen.

```
dependencies {
    // Googles Guava Bibliothek:
    // Format:  Organisation:Bibliotheksname:Version
    implementation 'com.google.guava:guava:27.0.1-jre'

    // JUnit Test Framework:
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.3.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.3.1'
    testRuntimeOnly 'org.junit.vintage:junit-vintage-engine:5.3.1'
}
```

Gradle: Quellen für Bibliotheken

Maven Central (https://search.maven.org/):

- Zentrale Sammlung fast aller Java-Bibliotheken
- Jeder kann eigene Bibliotheken hochladen

Jitpack (https://jitpack.io/):

- Beliebiges Git Repository als Abhängigkeit
- Git Repository braucht Build-Konfiguration (zum Beispiel Gradle)

Lokale Installation[2]:

- Bibliotheken lassen sich lokal installieren

Eigener Repository Manager[3]:

- Ähnlich zu Maven Central, aber unter eigener Kontrolle.
- Beispiel: Firma will nicht von externen Servern abhängig sein.

[2] https://docs.gradle.org/current/userguide/maven_plugin.html
[3] https://maven.apache.org/repository-management.html

Integration von Entwicklungswerkzeugen I

Integrierte Entwicklungsumgebungen stellen Umgebungen dar, mit der sich möglichst viele Aufgaben der Softwareentwicklung zentral steuern lassen.

- Semantischer Texteditor
- Refaktorierung (Umbenennen, Code umstrukturieren)
- Integration von Build-Systemen
- Integration von Versionsverwaltungssystemen
- Debugger (zeigt Zustand des Programms zur Laufzeit)

Viele IDEs unterstützen Plugins, womit sich Funktionalität nachrüsten lässt.

Beispiele für IDEs

IDEs für Java:

- IntelliJ IDEA (https://www.jetbrains.com/idea/)
- Eclipse (https://www.eclipse.org/ide/)
- Netbeans (https://netbeans.org/)

IDEs für andere Sprachen:

- Microsoft Visual Studio (C++, C#, F#, ...; für Windows)
- KDevelop (C, C++; für Unix)
- Apple Xcode (C, C++, Objective-C; für Mac)

Mit den richtigen Plugins können auch Text-Editoren wie Visual Studio Code, Atom, Emacs, Vim, etc. zu einer IDE werden.

IDEs

Empfohlene Features (verfügbar in allen IDEs):

- Dokumentation anzeigen
- Code-Navigation
  - Suchen
  - Zu Datei oder Klasse springen
  - Zu Deklaration springen
  - Variablen-Zugriffe, Aufrufhierarchie, Klassenhierarchie
- Anzeige von Fehlern und Warnungen
- Code Vervollständigung
- Code Generierung (Konstruktoren, Getter/Setter, Equals/HashCode, Interface-Methoden, ...)
- Formatieren von Code
- Refactoring
  - Umbenennen
  - Variablen und Methoden extrahieren
  - Methoden inlinen
- Programme und Tests ausführen
- Debuggen

$\Rightarrow$ Demo : IntelliJ IDEA