HSLU Hochschule Luzern

Verteilte Systeme und Komponenten

Buildautomatisation

Reproduzierbare Buildprozesse

Roland Gisler

FH Zentralschweiz

Inhalt

- Wie wird Software entwickelt?
- Automatisierung des Buildprozesses
- Buildwerkzeuge
- Apache Maven
- Demo

Lernziele

- Sie kennen die Vorteile eines automatisierten Buildprozesses.
- Sie können verschiedene Beispiele von Buildwerkzeugen benennen.
- Sie beherrschen die Anwendung eines ausgewählten Buildwerkzeuges (Apache Maven).
- Sie sind mit den wesentlichen Konzepten von Apache Maven vertraut.

Einleitung

Wie haben Sie bisher Software entwickelt?

- In den Modulen OOP/PLAB und AD haben Sie
  - typisch «alleine»,
  - in einem einzigen (kleinen) Projekt,
  - und meist vollständig in der Entwicklungsumgebung (IDE)

  gearbeitet.
- Wesentliche Schritte waren meist nur die Kompilation (automatisch durch die IDE), das Ausführen von Unit-Tests sowie, das Starten des «Programmes» selber.
  - Das lässt sich alles sehr einfach und direkt in der jeweiligen Entwicklungsumgebung erledigen.
- Die Arbeitsweise ist in einem realen Projekt aber etwas anders…

Wie wird Software typisch entwickelt?

- Typische Situation
  - Arbeit im Team ➔ gemeinsame Codebasis ➔ VCS.
  - Code aufgeteilt in mehrere ➔ Projekte / Module / Komponenten.
  - Verteilung des Endproduktes (Releases) in Form von binären Artefakten (typisch JAR-Dateien).
- Das heisst: Software entwickeln beinhaltet deutlich mehr Aufgaben:
  - Binaries erstellen für Deployment.
  - Testfälle ausführen, Test-Reports erstellen.
  - Qualitätssicherung (Codeanalyse, Metriken etc.).
  - Distributionen zusammenstellen und Deployment etc.
- Die Gesamtheit all dieser Tätigkeiten um aus Quellartefakten ein fertiges Produkt zu erstellen bezeichnet man als ➔Buildprozess.

Der Buildprozess

- Generieren, Kompilieren, Testen, Packen, JavaDoc erzeugen…
  Viele dieser Tätigkeiten kann man problemlos mit einer modernen IDE erledigen!
- Aber: Es sind zum Teil aufwändige und manuelle Schritte!
  - Sie sind somit mühsam und fehleranfällig.
  - Bei grossen Projekten langsam und langweilig (Wartezeiten).
  - Bei langweiligen Task wird er Mensch unzuverlässig.
- Was, wenn Sie das **n**-mal pro Tag machen müssen?
- Bob the Builder?

Automatisation des Buildprozesses

Automatisation von Builds mit Script/Batches

- Idee: Man könnte ein Script schreiben, dass die notwendigen Tools (Kompiler, Packer etc.) sequenziell mit allen notwendigen Parametern ausführt.
- 👍 Vorteile:
  - Vollautomatisierter Ablauf, keinerlei Interaktion mehr.
  - Reproduzierbare Ergebnisse.
  - Lange Builds können über bzw. in der Nacht laufen.
  - Unabhängig von Entwicklungsumgebung (IDE).
- 👎 Nachteile:
  - Eher sturer, unflexibler Ablauf (oder komplizierte Scripte).
  - Abhängigkeit von Shell und Plattform (OS).
  - Aufwändige Wartung und Erweiterung.

Alternative: Ein spezialisiertes Build-Werkzeug!

- Eigenständiges, spezialisiertes Werkzeug mit eigener (Script- oder Definitions-)Sprache.
- Optimiert für die typischen Build-**Aufgaben.**
  - Aufruf von Kompiler, Packer, Tests, Deploying.
  - vereinfachter Umgang mit Ressourcen (Dateimengen).
  - Abhängigkeiten zwischen Dateien überprüfen und steuern.
- Optimiert für die typischen Build-**Abläufe.**
  - Logische Abfolge und Abhängigkeiten zwischen Aufgaben.
- Plattformübergreifend lauffähig.
  - Abstraktion der Plattformspezifika.

➔ Das gibt es natürlich schon lange! Und Sie kennen es schon! ☺

Build-Werkzeuge

Build-Werkzeug: `make`

- Die Mutter aller Build-Tools: **`make`**
  - Hauptsächlich für **`C/C++`** - Projekte verwendet.
  - Existiert seit vielen Jahren, und wird noch immer genutzt.
  - Bietet eine sehr hohe Flexibilität.
- Sehr einfaches Beispiel:

```
program.o: program.c
	gcc -c program.c -o program.o

program: program.o
	gcc program.o -o
```

  - Vielleicht eine etwas gewöhnungsbedürftige Syntax?

Spezialisierte Build-Werkzeuge für Java

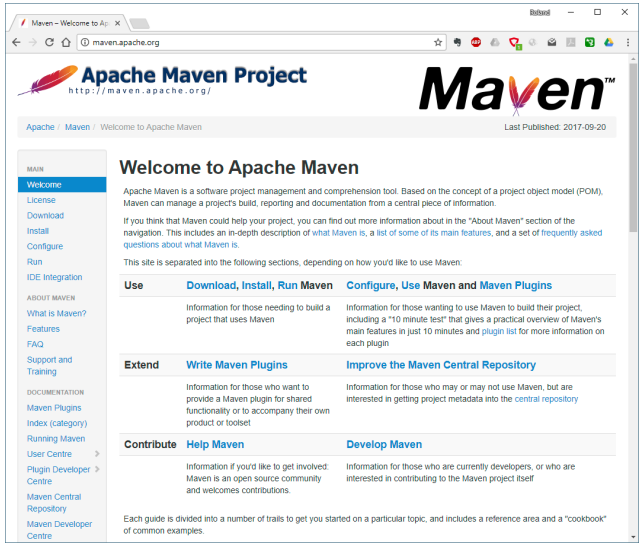
- Im Java-Ökosystem existiert mittlerweile eine ganze Palette von Werkzeugen!
  - Viele davon basieren selber auf Java (mindestens der JRE).
- Ein paar Beispiele:
  - **`Ant`** – «altes» und bewährtes Werkzeug, Java mit XML.
  - **`Maven`** – populäres, etabliertes Werkzeug, Java mit XML.
  - **`Buildr`** – junges Werkzeug, Ruby-Script.
  - **`Gradle`** – populäres, junges Werkzeug, Groovy-Script mit DSL.
  - **`Bazel`** – Buildwerkzeug von Google, Java mit Python-like Scripts.
- Die Qual der Wahl:
  - Implizite Regeln vs. explizite Aufgaben (Imperativ/Deklarativ).
  - Es gibt sehr viele unfaire Vergleiche (wie bei den VCS)!
  - Vergleich zu Datenbanken: Pragmatisch, stabil, langfristig…

Verschiedene Produkte – gemeinsame Ziele und Vorteile!

- Einheitliche, einfache Definition eine Build-Ablaufes.
  - Einfache und doch flexible Syntax/Sprache.
- Anwendung über relativ eingängige Build-Ziele/Goals/Targets.
  - Einfache Anwendung für Entwickler*in.
- Optimierte Abläufe (nur machen, was nötig ist)
  - z.B. optimierte Kompilation: Nur modifizierte Quelldateien kompilieren (wenn Quellen neuer als Kompilate).
- Erweiterbarkeit
  - Flexibel erweiterbar mit neuen Fähigkeiten / Funktionen.
- Möglichst geringer Ressourcenverbrauch (shell, headless etc.).
- Reproduzierbarer Ablauf, technologische Konstanz.

Apache Maven

Buildautomatisation mit Apache Maven



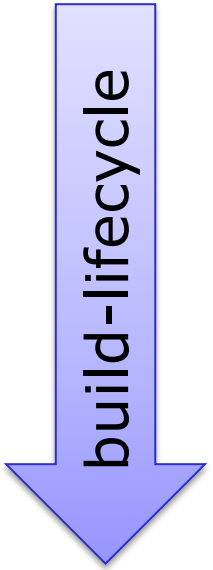
- Apache Maven - http://maven.apache.org/
- Ist selber in Java implementiert und somit plattformunabhängig.
- Deklaration des Projektes in XML.
  - Deklarativer Ansatz (nicht imperativ)!
  - Das zentrale Element pro Projekt stellt das **P**roject **O**bject **M**odel (POM) dar: **`pom.xml`**
  - Definiert alle Metainformationen, Plugins und Dependencies.
- Bewährtes und robustes Buildwerkzeug für Java!
  - Integration in praktisch jede(r) IDE vorhanden.
  - Minimale Ressourcen notwendig (nur JDK).
  - Basiert auf einem globalen, **binären** Repository!

Apache Maven - Konzept

- Maven selber kann eigentlich sehr wenig! Es definiert im wesentlichen nur die POM-Struktur und die ➔Lifecycle-Phasen.
- Alles wird dann von dynamisch geladenen Plugins (mit eigenen Releases) erledigt! Extreme grosse Vielfalt an Funktionen!
  - Entfernt vergleichbar mit der Eclipse Platform und (JDT-)Plugins.
- Somit ist Maven extrem **flexibel** und nahezu beliebig **erweiterbar**.
- Vorsicht: Neben den Core-Plugins (die von Apache Maven selber entwickelt werden), existieren sehr viele weitere (Dritt-)Plugins, deren Qualität sehr unterschiedlich ist.
  - Vorsicht: Maven-Plugins sind auch Abhängigkeiten!
- Man kann natürlich auch eigene Plugins schreiben.
  - Aber bitte nur wohlüberlegt: Lohnt es sich wirklich?

Apache Maven – «lifecycle phases» und «goals»

- Maven definiert einen generalisierten Ablauf eines Buildprozesses – den so genannten «lifecycle» (Auszug, Vereinfacht):
  - **`validate`** – Validiert die Projektdefinition.
  - **`compile`** – Kompliation der Quellen.
  - **`test`** – Ausführen der Unit-Tests.
  - **`package`** – Packen der Distribution (JAR, EAR etc.)
  - **`verify`** – Ausführen der Integrations-Tests.
  - **`install`** – Deployment im lokalen Repository.
  - **`deploy`** – Deployment im zentralen Repository.



- Die per Deklaration aktivierten Plugins registrieren sich und ihre Aufgaben (häufig) automatisch in den jeweiligen Phasen!
  - Buildprozess passt sich quasi dynamisch der Projektart an.

Apache Maven – Binäre Repositories

- Maven integriert auch das ➔Dependency Management.
- Im POM deklarierte Abhängigkeiten sowohl von Plugins als auch von Libraries werden aus einem zentralen Repository geladen.
  - The Central Repository: https://search.maven.org/
- Alle heruntergeladenen Artefakte werden in einem lokalen Repository auf dem Rechner gespeichert (gecached).
  - Lokales Repository: **`$HOME/.m2/repository`**
  - Nebenbei: Im **`$HOME/.m2`** befindet sich auch die Konfigurationsdatei **`settings.xml`** - haben sie sie kopiert?
- Firmen und Organisationen nutzen typisch ein eigenes Repository als lokaler Speicher und Mirror von öffentlichen Repositories.
  - HSLU RepoHub Nexus: https://repohub.enterpriselab.ch/

Apache Maven – Single- und Multimodulprojekte

- In den Modulen OOP/PLAB und AD haben wir bisher nur einfache «Single-Modul»-Projekte genutzt: Ein Verzeichnis enthielt genau ein einziges Projekt.
  - Einfache Struktur für kleine in sich abgeschlossene Projekte.
  - Kriterium: **Releaseeinheit**, d.h. was wollen wir einzeln unter einer Version «releasen» (veröffentlichen) können.
- Maven unterstützt aber auch Multimodulprojekte, d.h. ein Projekt kann beliebig viele Submodule enthalten ➔Modularisierung!
  - Übergeordnete Konfiguration wird an Submodule «vererbt».
  - Abhängigkeiten zwischen Submodulen können definiert werden.
- Diese Technik setzen wir im Modul VSK ein!

Demo / Screencast's

- Maven – Einsatz in der Shell:
  EP_12_SC01_MavenConsole.mp4
- Maven – Integration in IDE (NetBeans, Eclipse, IntelliJ):
  EP_12_SC02_MavenIDE.mp4

Apache Maven Dokumentation

- Ausführliche Dokumentation auf der Projektseite:
  http://maven.apache.org/
- Wichtige Konzepte (zur Vertiefung für Interessierte):
  - Lifecycles, Phases und Goals
  - Dependencies / Dependency-Resolution
  - Repositories
  - Deployment
  - Plugins
  - Site

Installation von Maven – Einsatzszenarien

- **Variante 1**, minimalistisch: Maven ist in vielen IDEs bereits enthalten!
  - z.B. Eclipse, Netbeans, IntelliJ - nicht immer aktuellste Version!
  - Vorteil: sehr einfache Anwendung ohne Aufwand.
  - Nachteil: **unnötige** Abhängigkeit zur IDE.
- **Variante 2**, **empfohlen**: Installation von Maven auf das System
  - Distribution als ZIP, einfach entpacken!
  - **`$MAVEN_HOME/bin`** in Suchpfad aufnehmen, damit **`mvn`**-Kommando gefunden wird
  - Vorteil: **Unabhängig** von IDE, Build in einer Shell möglich
  - Nachteil: Installation (nicht wirklich ein Nachteil, oder?)
- **Wichtig**:
  `settings.xml` in jedem Fall lokal in **`$HOME/.m2`** kopieren.

Quellen und Links

- Apache Maven – http://maven.apache.org/
- Gradle – https://gradle.org/
- Apache Ant – http://ant.apache.org/

HSLU Hochschule Luzern

Fragen?