Referent:
Thomas Storch (FI für Anwendungsentwicklung, 1. Lj.)

Vortrag:
Apache Maven

Datum:
25.04.2012

Deutsche Software
Engineering & Research GmbH

Steinstraße 11
02826 Görlitz – Germany
Telefon: +49 35 81 / 374 99 – 0
Telefax: +49 35 81 / 374 99 – 99
E-Mail: info@dser.de
Internet: www.dser.de

Inhalt

- Intro: Was ist MAVEN?
- Java-Projekt-Konfiguration mit MAVEN
- Repository (remote)
- Repository (lokal)
- POM
- Parent.pom
- pom.xml
  - artifactId
  - groupId
  - packaging
  - version
  - version ranges
  - SNAPSHOTs
  - scopes
  - exclusions
- Dependencies
- Dependency Management
- Plugins (jar/war/pmd)
- Build-Lifecycle
- Lifecycle-Phase == Goals
- Code-Demo
  - Maven Installation + Integration in Eclipse
  - Utils-Projekt
  - Hauptprojekt
  - Test im Hauptprojekt mit Dependency JUnit
  - executable jar erstellen
- Vorteile
- Nachteile
- Quellen

Intro: Was ist MAVEN? 1/2

- Build-Management-Tool der Apache-Foundation
- Name aus dem Jiddischen → „Speicher des Wissens“
- Ziele:
  - Entwickler können in kurzer Zeit kompletten Entwicklungsstand eines Projekts nachvollziehen
  - build-Prozess einfach gestalten
  - einheitliches build-System anbieten
  - qualitative Projektinformationen zur Verfügung stellen
  - Richtlinien für *best practices*(-Entwicklung) bereitstellen
  - transparente Migration auf neue Funktionen erlauben

Intro: Was ist MAVEN? 2/2

- Abbildung von *Convention over Configuration* für gesamten Zyklus der Softwareerstellung
- Unterstützung des Software-Entwicklers bei:
  - Anlage eines Softwareprojekts
  - Kompilieren
  - Testen und „Packen“
  - Verteilen der Software auf Anwendungsrechnern

  = Automatisierung möglichst vieler Schritte
- Maven-Standard: wenige Konfigurationseinstellungen für viele Aufgaben des Build-Managements, um Softwareprojekt-Lebenszyklus abzubilden

Java-Projekt-Konfiguration mit MAVEN 1/2

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

folder/file	content
src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/scripts	Application/Library scripts
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
NOTICE.txt	Notices and attributions required by libraries that the project depends on
README.txt	Project's readme

Java-Projekt-Konfiguration mit MAVEN 2/2

- manuell: Rechtsklick in Projekt-Explorer
  1. New > Project > General > Project > Next > Namen „xyz“ vergeben > Finish
  2. in *xyz* > New > Folder > „src“ > Finish
  3. in *src* > New > Folder > „main/java“ > Finish
  4. in *src/main* > New > Folder > „resources“ > Finish
  5. in *src* > New > Folder > „test/java“ > Finish
  6. in *src/test* > New > Folder > „resources“ > Finish
  7. in xyz > New > Other > Maven > Maven POM file > Next > Next > Finish
  8. *pom.xml* füllen
  9. Rechtsklick auf *xyz* > Maven > Enable Dependency Management
  10. Rechtsklick auf *xyz* > Properties > Maven > Haken *Resolve dependencies for Workspace projects* raus
  11. im eclipse-Navigator-Fenster: in *xyz* > Dateien & Ordner mit Punkt vorn + *target*-Ordner markieren > Rechtsklick > Team > Add to svn:ignore > damit werden diese Dateien (eclipse-Config~ + temp-Ordner *target*) nicht auf SVN hochgeladen
- ODER: New > Project > Maven-Project > create a simple Project

Repository (remote)

- Artefakt: Produkt, das als Zwischen- oder Endergebnis in der Softwareentwicklung entsteht
- Repository: verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen (Projekt-) Artefakten
- Repository enthält Programmpakete + zugehörige Metadaten, z.B. Beschreibungen der Pakete, Abhängigkeitsinformationen, Change Logs
- Installieren bzw. Aktualisieren der Software aus Repository übernimmt MAVEN

Repository (lokal)

- .m2-Ordner im Home-Verzeichnis des Users
- Kopien aller Dateien und Bibliotheken, aus entfernten Repository heruntergeladen wurden
- Zurückgreifen auf lokale Kopie bei mehrmaliger Nutzung
- neue Bibliotheken oder aktuellere Version von bestehender Bibliothek → herunterladen aus entferntem Repository

POM

- project object model repräsentiert durch *pom.xml*
- zentrale Projektbeschreibungs- und -steuerungsdatei mit Metadaten zum Projekt

```
<project xmlns=http://maven.apache.org/POM/4.0.0
         xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>de.meinefirma</groupId>
        <artifactId>ts.jug.test</artifactId>
        <version>2.0.0</version>
        <name>my test project</name>
        <packaging>jar</packaging>
</project>
```

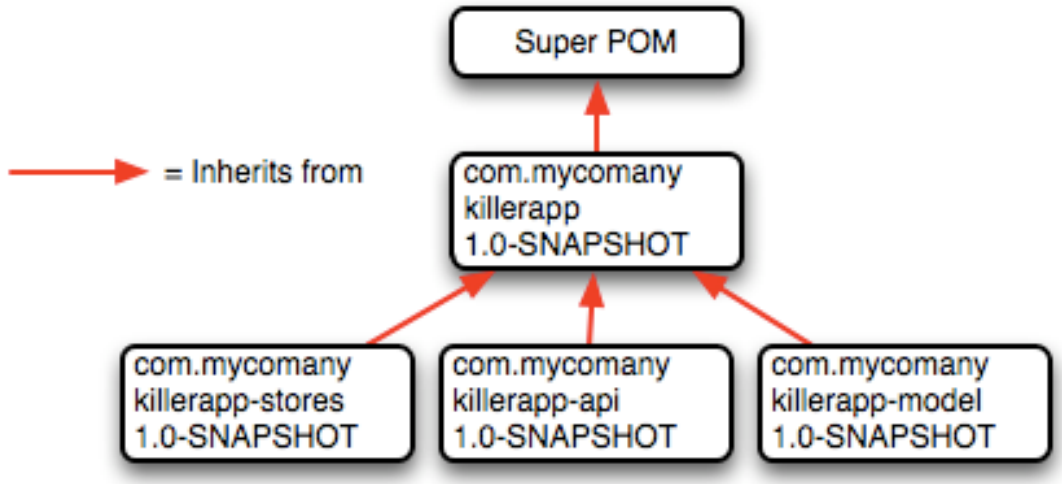
POM

- pom.xml ist Kern einer Maven-Projekt-Konfiguration
- einzelne Konfigurationsdatei, welche Mehrheit der benötigten Informationen enthält
- „POM ist riesig, aber es ist nicht notwendig alle Feinheiten zu verstehen um die Effizienz zu nutzen“ (Apache)
- folgende Elemente werden in der POM zusammengeführt:
  - dependencies
  - developers and contributors
  - plugin lists (including reports)
  - plugin executions with matching ids
  - plugin configuration
  - resources

parent.pom

- für Firma eigene pom definieren, die in allen Projekten eingesetzt wird:

```
<project>
        <mo
        <gr
        <ar
        <ve
        <pa
<project>
```



- in untergeo

```
<parent>
        <groupId>de.meinefirma</groupId>
        <artifactId>company.parent</artifactId>
        <version>1.0</version>
</parent>
```

pom.xml

- `<groupId>de.meinefirma</groupId>`
- pro Punkt existiert ein Unterverzeichnis im Repository
- üblicherweise einzigartig innerhalb einer Firma/eines Projekts
- Punkt-Notation muss nicht der Paket-Struktur des Projekts entsprechen – sollte aber
- Punkte werden durch OS-spezifische Verzeichnisteiler ersetzt welche relativen Pfad ausgehend vom Basis-Repository ergeben
- Zum Beispiel *de.meinefirma* ist im Verzeichnis *$M2_REPO/**de/meinefirma*** aktiv

pom.xml

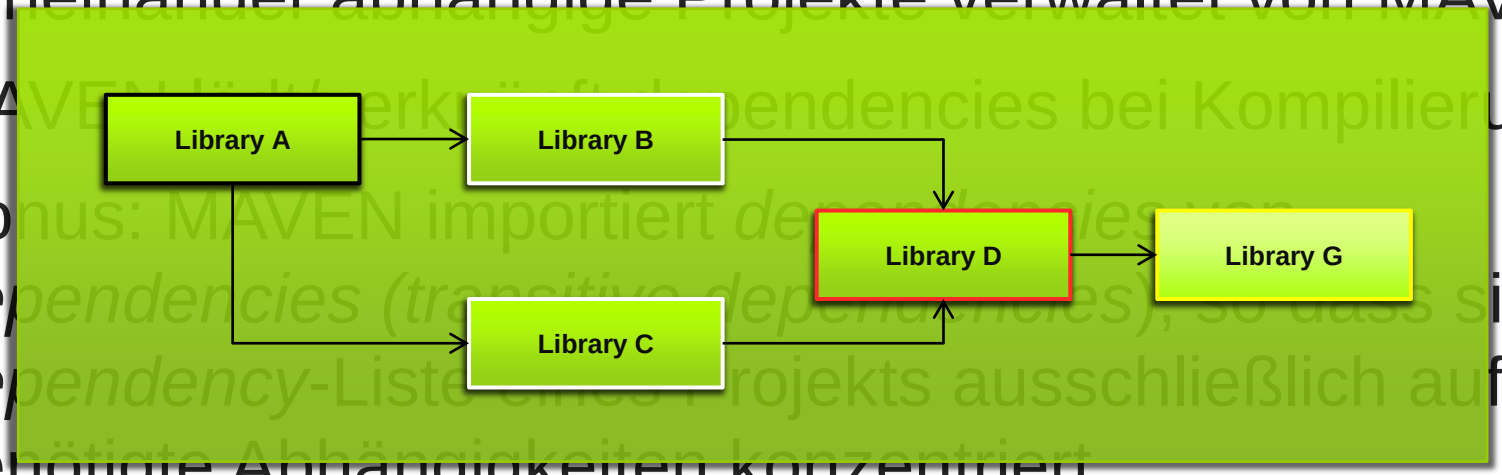
- `<artifactId>ts.jug.test</artifactId>`
- Punkte-Notation irrelevant → *artifactId* ist nur *ein* Verzeichnis
- üblicherweise der Name unter dem Projekt bekannt ist
- erzeugt zusammen, mit der *groupID*, einen Schlüssel der das Projekt von allen anderen Projekten auf der Welt unterscheidet
- zusammen mit *groupID* definiert sie den Artefakt-Bereich innerhalb des Repository
- Zum Beispiel „ts.jug.test“ in Verzeichnis *$M2_REPO/de/meinefirma/**ts.jug.test***

pom.xml

- `<version>0.0.1-SNAPSHOT</version>`
- manuell zu setzen
- *groupId:artifactId* bezeichnet ein einzelnes Projekt → nicht welche Version es verkörpert
- auch genutzt, um innerhalb eines Artefakt-Repositories Versionen voneinander zu unterscheiden
- „ts.jug.test“ *version 0.0.1*-Dateien im Verzeichnis *$M2_REPO/de/meinefirma/ts.jug.test/**0.0.1-SNAPSHOT***

Dependencies

- Grundstein jeder POM → *dependency*-Liste
- voneinander abhängige Projekte verwaltet von MAVEN
- MAVEN löst Verknüpfungen der Dependencies bei Kompilierung
- Bonus: MAVEN importiert *dependencies* von *dependencies (transitive dependencies)*, so dass sich *dependency*-Liste eines Projekts ausschließlich auf benötigte Abhängigkeiten konzentriert



- Anlage von Dependencies sowohl in Projekt als auch in Repository möglich

Dependency Version Ranges

`<version>1.4</version>`
- *version* startet mit “1.4” (bspw. “1.4.0_08”, “1.4.2_07”, “1.4”)

`<version>[1.4]</version>`
- nur *version* 1.4

`<version>(,1.0]</version>`
- *version* ≤ 1.0

`<version>[1.6,)</version>`
- *version* ≥ 1.6

`<version>[1.3,1.5)</version>`
- 1.3 ≤ *version* ≤ 1.5 (1.3, 1.4, 1.5)

`<version>(,1.0],[1.2,)</version>`
- *version* ≤ 1.0 **oder** *version* ≥ 1.2

`<version>(,1.1),(1.1,)</version>`
- schließt *version* 1.1 aus (z.Bsp. wenn 1.1 in Kombination mit dieser Library nicht funktioniert)

- worst practice: `<version>RELEASE</version>`
  - immer aktuellste Version aus Repository geladen→ irgendwann ist eine Änderung enthalten, mit der das aktuelle Projekt nicht klar kommt → Fehler

pom.xml

- `<packaging>jar</packaging>`
- Artefakttyp des Projekts
- WAR (web application archive)
- RAR (resource adapter archive)
- EAR (enterprise archive)
- SAR (service archive)
- APK (android application package)
- … usw.

Snapshots

```
<version>0.0.1-SNAPSHOT</version>
```

- **-SNAPSHOT** solange die Version noch nicht final ist → nur für Entwicklungsprozess
- für sich schnell ändernden Code mit vielen Bug Fixes und Verbesserungen
- **-SNAPSHOT** entfernen um go-live / funktionelle Änderungen zu markieren
- Snapshot-Speicherung in normalem remote-Repository → eigenes Snapshot-Repository möglich
- Apache: “Snapshots are for testing purposes only and are not official releases.”

Scopes

- Verwendungsbereich
- definiert Sichtbarkeit/Zugriff der dependencies
- Auflösung des Scopes übernimmt MAVEN eigenständig → mit Lebenszyklus fest verdrahtet

scope	**dependency**
compile	in allen *classpaths* verfügbar, sprich beim Kompilieren
provided	zum Zeitpunkt von *compile* und der Ausführung von Tests verfügbar
runtime	bei Tests und zur Laufzeit vorhanden, aber nicht bei normaler Kompilierung
test	nur bei Tests

exclusions

- beziehen transitive dependency nicht mit ein
- Beispiel: benötigt *ts.jug.test* *ts.jug.test.utils* und wir wollen *ts.jug.test.utils* und seine dependencies nicht nutzen → als *exclusion* hinzufügen

```
<dependencies>
    <dependency>
       <groupId>de.meinefirma</groupId>
       <artifactId>ts.jug.test</artifactId>
       <version>1.0</version>
       <exclusions>
         <exclusion>
                <groupId>de.meinefirma</groupId>
                <artifactId>ts.jug.test.utils</artifactId>
         </exclusion>
       </exclusions>
    </dependency>
</dependencies>
```

Dependency-Management 1/2

- Definition idealerweise in parent.pom
- Verwendung bei Unterprojekten
- nicht in jeder Unterprojekt-pom.xml vollständige Angabe aller Dependency-Eigenschaften
- verfolgt Gedanken zentraler Dependency-Verwaltung
- Bsp.: Entwickler A trägt Bibliothek in pom.xml ein und lädt Änderungen ins SVN → Entwickler B zieht Update → MAVEN lädt automatisch Abhängigkeiten nach
- Ziel: Verkleinerung der Child-pom.xml

Dependency-Management 2/2

- Parent:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8</version>
    </dependency>
 </dependencies>
</dependencyManagement>
```

- Child:

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
 </dependencies>
```

JAR/WAR/PMD Plugin

- JAR Plugin
  - ermöglicht es *.jar* zu builden und zu signieren
- WAR Plugin
  - verantwortlich um alle Artefakt-Abhängigkeiten, Klassen und Resourcen einer Web Applikation zu sammeln und sie in ein **w**eb **a**pplication a**r**chive zu verpacken
- PMD Plugin
  - startet automatisch das PMD Codeanalyse-Tool auf Projektcode und erstellt Report
  - unterstützt das “Copy/Paste Detector”-Tool (CPD) welches mit PMD ausgeliefert wird
- PMD scans Java source code and looks for potential problems like:
  - Possible bugs - empty try/catch/finally/switch statements
  - Dead code - unused local variables, parameters and private methods
  - Suboptimal code - wasteful String/StringBuffer usage
  - Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
  - Duplicate code - copied/pasted code means copied/pasted bugs

Build-Lifecycle

- Lebenszyklen sind zentrale Konzepte von MAVEN
- Prozess für build und Verteilung eines bestimmten Projekts klar definiert
- POM garantiert verlangtes Ergebnis für jeden lifecycle
- drei eingebaute *lifecycles*
  - *clean* handles project cleaning
  - *default* handles your project deployment
  - *site* handles the creation of your project's site documentation
- beim Ausführen eines bestimmten Lebenszyklus werden alle vorherigen mit ausgeführt

Lifecycle-Phasen == Goals

- gängigste *default*-Lifecycle-Phasen:

Phase	Beschreibung
validate	überprüfen, ob das Projekt korrekt ist und alle benötigten Informationen verfügbar sind
compile	den Quellcode des Projekts kompilieren
test	prüft kompilierten Quellcode mit geeignetem Test-Framework - Diese Test sollten nicht voraussetzen, dass der Code verpackt oder deployed ist.
package	kompilierten Code nehmen und in ein verteilfähiges Format verpacken, z. Bsp. JAR
integration-test	das Paket/Artefakt verarbeiten und wenn notwendig in eine Umgebung deployen in der Integrationstests ausgeführt werden können
verify	Tests ausführen, um zu prüfen ob das Paket/Artefakt gültig (valid) ist und den Qualitätskriterien entspricht
install	Paket/Artefakt in lokales Repository (…/User/EigeneDateien/.m2/repository/de/dser/…) installieren, um es in anderen lokalen Projekten als Abhängigkeit zu verwenden
deploy	wird in Integrations- oder Veröffentlichungs-Umgebung (release environment) erledigt → kopiert finales Paket/Artefakt zum remote-Repository um es mit anderen Entwicklern und Projekten zu teilen

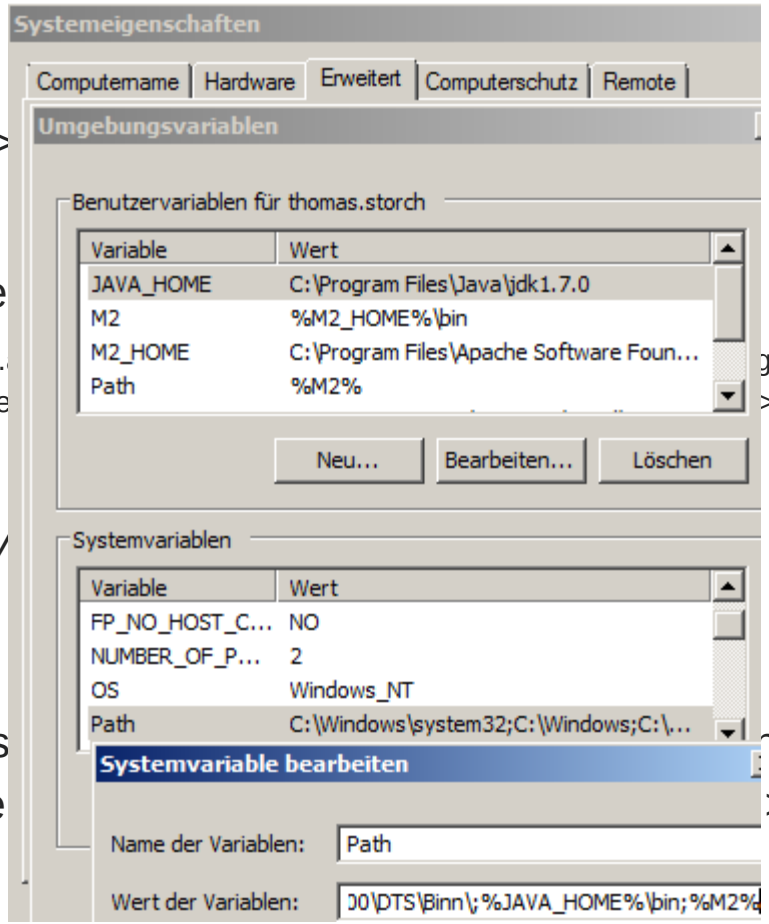
MAVEN-Installation & Integration in eclipse

- Voraussetzungen: aktuelles JDK & Eclipse

1. http://maven.apache.org/ > Download Maven > unten auf der Download Seite
2. in C:\Users\VORNAME.NACHNAME\.m2 Date

<?xml version="1.0"?> <settings xsi:schemaLocation="http://maven. gs-
1.0.0.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance >

3. Eclipse > Help > Install New Software > Add > http://m2eclipse.sonatype.org/sites/m2e > *Mav* > Accept > Finish > Restart Now
4. Eclipse > Help > Install New Software > Add > http://m2eclipse.sonatype.org/sites/m2e-extras nd *Maven Integration for Eclipse Extras* auswähle > Restart Now



5. Eclipse > Window > Preferences > Maven > Installations > Add > Maven-Pfad suchen (z.Bsp.: C:\Program Files\Apache Software Foundation\apache-maven-2.2.1) > OK > OK

Code-Demo

- Utils bauen
  - New > Project > Maven Project > Next > Create a simple project
    - Goup Id: *de.dser* > Artifact Id: *ts.jug.test.utils*
  - New > Class
    - Package: *de.dser.ts.jug.test.utils* > Name: *StringFormatter*

```
public String format(String message) {
        return message.toLowerCase();
}
```

  - Run As > Maven install (kompilieren)

→ wird später unsere Abhängigkeit

Code-Demo

- Hauptprojekt bauen
  - New > Project > Maven Project > Next > Create a simple project
    - Goup Id: *de.dser* > Artifact Id: *ts.jug.test*
  - New > Class
    - Package: *de.dser.ts.jug.test* > Name: *Main* + main()

```
String message = "HaLlO WeLt";
message = new StringFormatter().format(message);
System.out.println(message);
```

  - pom.xml > Dependency (Tab) > Add > *ts.jug > utils > pom.xml zeigen
  - in Main > Strg + 1 auf *StringFormatter*
  - Run As > Maven install > in m2-Ordner & in ProjectExplorer > Maven Dependencies

→ MAVEN holt aus lokalem Repository utils-dependency

Code-Demo

- Hauptprojekt: New > Class
  - Package: *de.dser.ts.jug.test* > Name: *EvenChecker*

```
public boolean isEven(long value) {
   long modulo = value % 2;
   boolean isEven = (modulo == 0);
   return isEven;
}
```

- *EvenChecker > MoreUnit > Jump to …*
  - *Source folder* `ts.jug.test/src/`**`test`**`/java`
- Dependency zu Junit in pom.xml

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.10</version>
   <scope>test</scope>
</dependency>
```

Code-Demo

- Plugins in pom.xml > kopiert referenzierte Libraries (=dependencies) in *target*-folder → erstellt Manifest-file Eintrag (*Class-Path*) mit lib-Folder (**`<outputDirectory>`**) → erspart zusammenkopieren der verschieden Dependencies
- ausführbare *jar* wird erstellt → Namen (**`<finalName>`**) änderbar
- → nur *compile*, nicht *test*
- cmd > target > *java -jar myMain.jar*
- danach: mvn clean in cmd zeigen

Vorteile

- Fördert:
  - Standardisierungen
  - *Convention over Configuration* und die Realisierung von *Best Practices*
  - Wiederverwendung
  - einheitliche Verzeichnisstrukturen
  - einheitliche Organisation der Abhängigkeiten
- vereinfaches Handling bei *vielen* Abhängigkeiten und benötigten Artefakten
- durch Definition der *goals* in Plug-ins wird Arbeitsteilung zwischen Konfigurationsmanagement und Softwareentwicklung gefördert
- erzeugt nicht nur Javadoc, sondern auch weitere hilfreiche Dokumentationen
- bietet Unterstützung und Anbindung für weitere Anwendungen (Fehlerverfolgung, Reporting-Systeme, Integrationssysteme)

Nachteile

- Installation und Konfiguration
- Einarbeitungszeit
- Build-Prozess wird abstrakter

Quellen:

Thema	URL
Was ist MAVEN?	http://de.wikipedia.org/wiki/Apache_Maven http://de.wikipedia.org/wiki/Konvention_vor_Konfiguration http://maven.apache.org/what-is-maven.html
Java-Projekt-Konfiguration mit MAVEN	http://maven.apache.org/guides/getting-started/index.html http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html
Repository	http://www.ordix.de/ORDIXNews/4_2007/Java_J2EE_JEE/maven_build_management.html
POM / pom.xml	http://maven.apache.org/pom.html
Dependency Version Ranges	http://docs.codehaus.org/display/MAVEN/Dependency+Mediation+and+Conflict+Resolution#DependencyMediationandConflictResolution-DependencyVersionRanges
parent.pom	http://www.sonatype.com/books/mvnref-book/reference/figs/web/pom-relationships_pom-inherit-simple-super.png
JAR/WAR/PMD Plugin	http://maven.apache.org/plugins/maven-jar-plugin/ http://maven.apache.org/plugins/maven-war-plugin/ http://maven.apache.org/plugins/maven-pmd-plugin/ http://pmd.sourceforge.net/
Dependency-Management	http://adelio.org/softwareentwicklung-im-team-teil-3-dependency-management/
Build-Lifecycles	http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
Lifecycle-Phasen == Goals	http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html http://www.slideshare.net/thorque/maven2-die-nchste-generation-des-buildmanagements (Seite 18)
Nachteile	http://www.ordix.de/ORDIXNews/4_2007/Java_J2EE_JEE/maven_build_management.html

- http://www.torsten-horn.de/techdocs/maven.htm