

# Dependencies

# Worum geht es?

- Java Programme verwenden fast immer mehrere Zusatzbibliotheken
- Für alle Buildumgebungen müssen die gleichen Bibliotheken verwendet werden
- Es soll nachvollziehbar sein, welche Bibliotheken in welcher Version verwendet werden
- Bibliotheken müssen regelmässig ergänzt und aktualisiert werden

=> Dependency Management von Maven

# Zusatzbibliotheken zu einem Maven Build hinzufügen

- in der pom.xml innerhalb eines `<dependencies>` Tags
- mindestens benötigt werden die GAV-Angaben:
  - `groupId`: Gruppe oder Organisation
  - `artifactId`: Eindeutiger Identifier innerhalb der Organisation
  - `version`

# Beispiel: Minimale Konfiguration

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.6.0</version>
    </dependency>
    ...
  </dependencies>
</project>
```

# Wie finde ich die GAV-Angaben zu einer Dependency?

- <http://search.maven.org> oder Firmenrepository
- Webseite der Bibliothek
  - Einbinden weiterer Repositories
  - Hochladen auf ein Firmenrepository
  - Einbinden als lokales JAR
    - Vorsicht: Erschwert Automatisierung des Builds

# Versionsnummern von Dependencies

```
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-databind</artifactId>  
  <version>2.6.0</version>  
</dependency>
```

- Standard: release Version:
  - `<version>2.6.0</version>`
  - immutable
- Version ranges
  - Exclusive quantifiers: `( , )`
  - Inclusive quantifiers: `[ , ]`
  - `<version>[2.0,3.0)</version>`
  - `<version>[,2.7]</version>`
  - Nicht zu empfehlen: verlangsamen Build, Reproduzierbarkeit nicht gegeben

# Dependency Scope

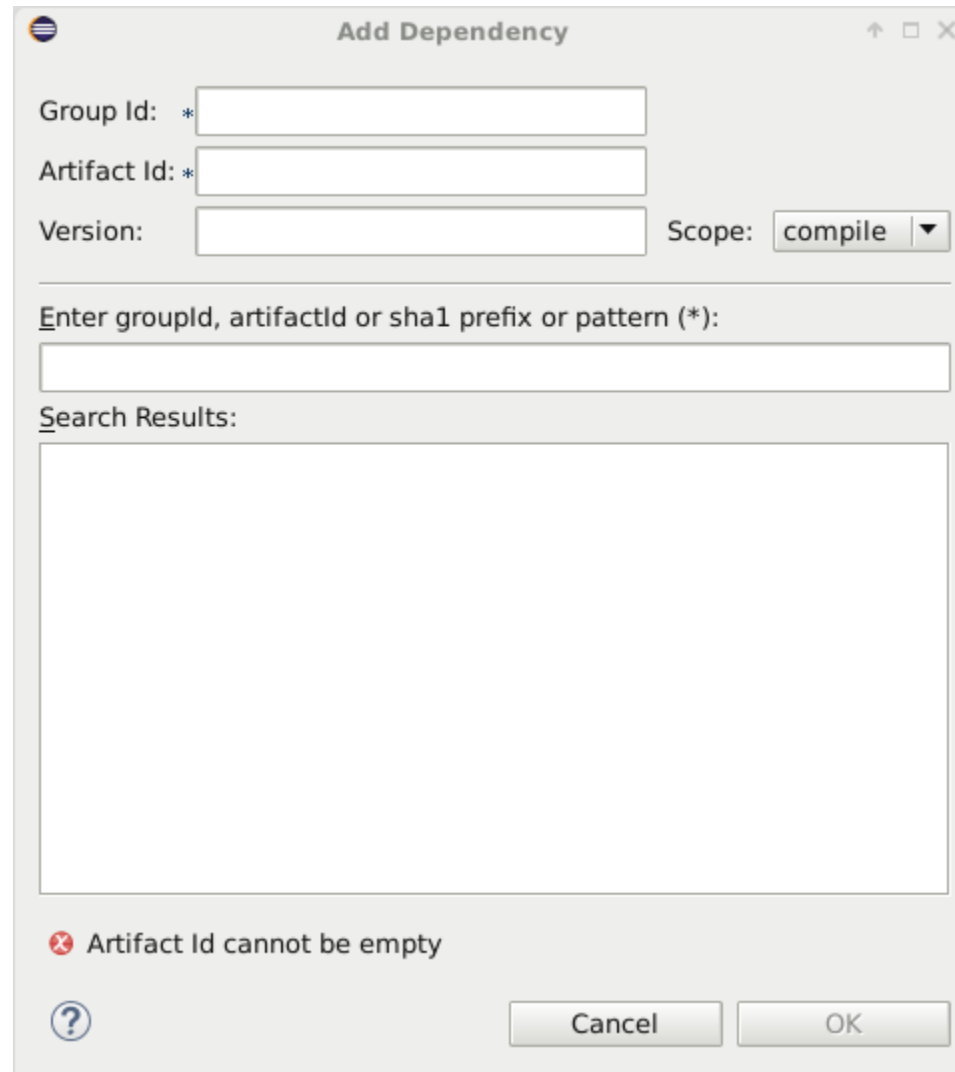
- Nicht alle dependencies werden immer benötigt
- **compile**: Default; Compile Dependencies sind in allen Classpaths verfügbar
- **provided**: Vom JDK oder Container bereitgestellt; nur in den Kompilierungs und Test Classpaths verfügbar
- **runtime**: Dependency wird nicht zur Kompilierung benötigt, sondern nur zur Laufzeit; im Klassenpfad von Runtime und Test aber nicht im Compile Classpath
- **test**: Wird nur zur Kompilierung und Ausführung von Tests benötigt
- **system**: Lokale Bibliothek, ähnlich wie provided

# Beispiel: Erweiterte Konfiguration

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <type>jar</type>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
</project>
```



# Eclipse Plugin: Hinzufügen einer Dependency



**Add Dependency**

Group Id: \*

Artifact Id: \*

Version:  Scope: **compile** ▼

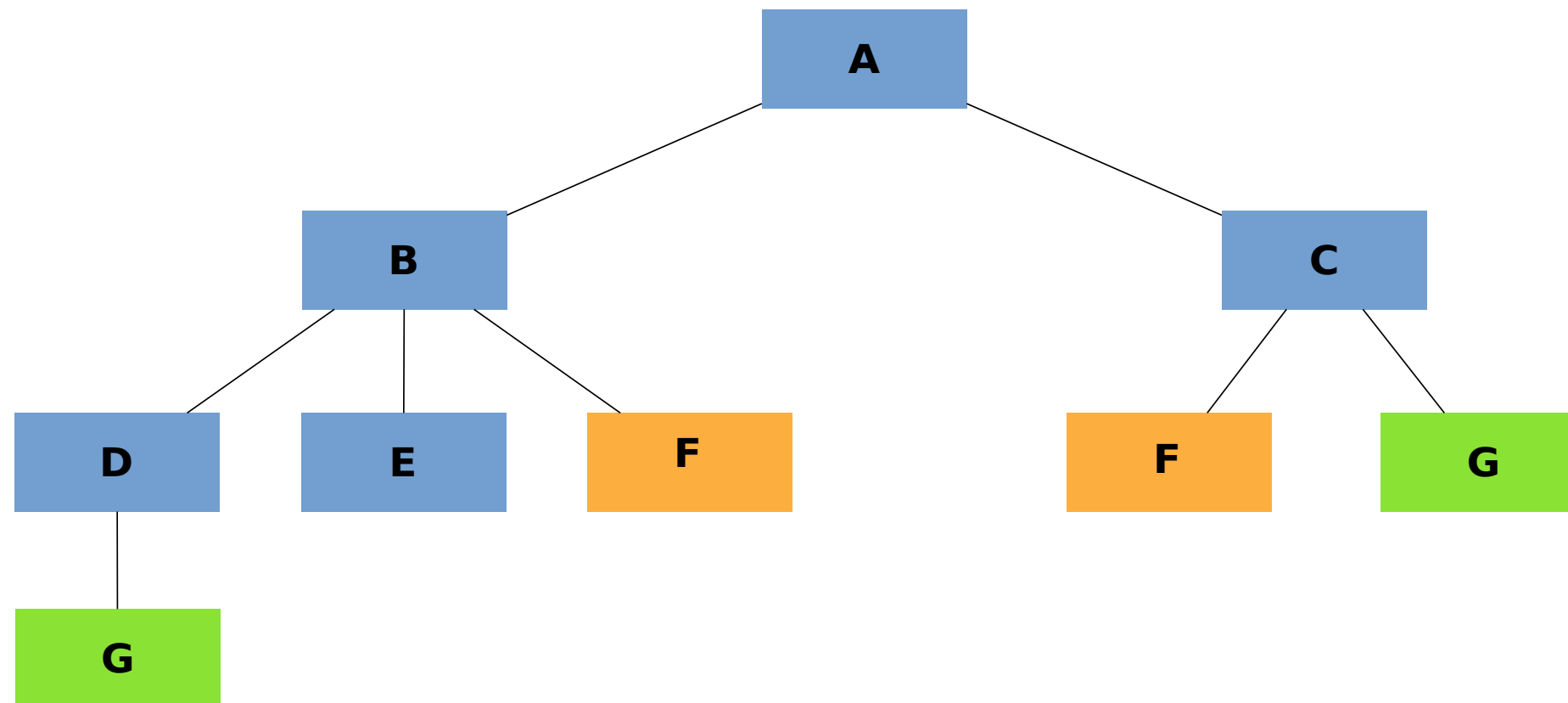
Enter groupId, artifactId or sha1 prefix or pattern (\*):

Search Results:

✖ Artifact Id cannot be empty

? Cancel OK

# Transitive Dependencies



- Dependencies können selbst wieder Dependencies haben
- Kommt eine Dependency in mehreren Versionen vor, wird nur eine verwendet
- Dependency Mediation
  - Nearest first: Niedrigere Level haben Priorität vor höheren Leveln
  - First found: Auf demselben Level wird die erste gefundene Dependency verwendet

# Problembehandlung

- Bei der Dependency Mediation kann es passieren, dass eine ungeeignete z.B. zu alte Version gewählt wird
- Erste Möglichkeit: Exclude von dependencies
- Zweite (bessere) Möglichkeit: Verwendung von <dependencyManagement>
  - Überschreibt Mediation

# Beispiel Exclude

```
<dependency>
  <groupId>sample.ProjectA</groupId>
  <artifactId>Project-A</artifactId>
  <version>1.0</version>
  <scope>compile</scope>
  <exclusions>
    <exclusion>
      <groupId>sample.ProjectB</groupId>
      <artifactId>Project-B</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

# Beispiel Dependency Management

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>sample.ProjectB</groupId>
      <artifactId>Project-B</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

# Dinge, die man eigentlich nicht machen möchte

- Einbinden als lokales jar (erschwert Automatisierung)
- Einbinden von GitHub Repositories mittels <https://jitpack.io/>

# Plugins

# Worum geht es?

- Jeder Task in Maven wird durch ein Plugin ausgeführt
- Ein Plugin stellt Aktionen in Form sog. `goals` bereit
  - Aufruf `mvn [pluginname]:[goal]`
  - Beispiel: `mvn compiler:compile`
- Einige Plugin Goals können ausserhalb des Lifecycles mittels Kommandozeilentools ausgeführt werden
- Einige Plugins werden bereits mitgeliefert

# Übersicht Plugins

## Core

clean  
compiler  
deploy  
failsafe  
install  
resources  
site  
surefire  
verifier

## Reporting

Changelog  
Changes  
Checkstyle  
Clover  
Javadocs  
PMD

## Packaging

jar war ear  
ejb rar pom  
shade

## Tools

docker  
grunt

## Application Servers

cargo  
jetty  
tomcat

## IDE integration

eclipse  
idea

## Utilities

Help  
Release  
Assembly



# mvn clean install

```
└─$ mvn clean install
...
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ gs-spring-boot ---
...
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ gs-spring-boot ---
...
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ gs-spring-boot ---
...
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ gs-spring-boot ---
...
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ gs-spring-boot ---
...
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @ gs-spring-boot ---
...
[INFO] --- maven-jar-plugin:2.5:jar (default-jar) @ gs-spring-boot ---
...
[INFO] --- maven-failsafe-plugin:2.18:integration-test (default) @ gs-spring-boot ---
...
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ gs-spring-boot ---
```

# Beispiel: Einbinden eines Plugins in die pom.xml

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.5</version>
  <configuration>
    <autoVersionSubmodules>true</autoVersionSubmodules>
  </configuration>
</plugin>
```

# Maven Help Plugin

Kommandozeilenaufruf:

```
mvn help:describe -Ddetail=true -Dcmd=jar:jar  
mvn help:describe -Dplugin=compiler  
mvn help:effective-pom
```

Details: <http://maven.apache.org/plugins/maven-help-plugin/>

# Dependency Plugin

Kommandozeilenaufruf:

```
mvn dependency:help  
mvn dependency:tree  
mvn dependency:list
```

<https://maven.apache.org/plugins/maven-dependency-plugin/>

# Properties

# Worum geht es

- Properties sind Zuordnungen von Werten an Variablennamen
- Können in Projekt Ressourcen verwendet werden
  - `src/main/resources`
  - Muss im Maven Resource Plugin aktiviert werden
- Benutzerdefinierte Properties (frei definiert)
  - Für Angaben die mehrfach in der `pom.xml` verwendet werden
- Benutzerdefinierte Properties (vordefinierte)
  - Als Konfigurationsparameter
- Vordefinierte Properties mit Angaben zur Buildumgebung
  - Von der Laufzeitumgebung bereitgestellt
  - Informationen über Java, Maven, Verzeichnisse

# Beispiel: Benutzerdefinierte Properties

```
<properties>
  <junit.version>4.9</junit.version>
  <hibernate.version>3.3.1.GA</hibernate.version>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
  </dependency>
</dependencies>
```

# Benutzerdefinierte Properties als Konfigurationsparameter

- User Properties für die Konfiguration des Builds
- Liste nicht fix
- Sind sie nicht vorhanden werden sie mit Defaultwerten belegt
- Beispiel
  - `<maven.compiler.source>`
  - Default: 1.5



# Vordefinierte Properties mit Angaben zur Buildumgebung

- Shellumgebungsvariablen
- POM Werte
  - z.B. Projektversion, Projektverzeichnis
- Java System Properties
  - z.B. JAVA\_HOME, Java Version, Betriebssystem, File Separator
- Maven Informationen
  - z.B. Version, Installationsverzeichnis
- Build Informationen
  - z.B. Zeitpunkt des Builds

# Übungsanwendung

- Einfache Webanwendung, die Kalenderdaten in JSON und einem Kalenderformat bereitstellt
  - <http://localhost:8080/events>
  - <http://localhost:8080/events/ical>
- Basiert auf Spring Boot
- Wird im Laufe des Kurses erweitert

# Übung 2

- Die Sourcen für diese Übung liegen im Verzeichnis `uebung - spring1`
- Erstelle eine `pom.xml` für das Projekt
- Verwende als `groupId` `org.informatica`
- Setze die `artifactId` auf einen eindeutigen Wert
- Ergänze eine Dependency zu Spring Boot  
(`org.springframework.boot:spring-boot-starter-web:1.2.5.RELEASE`) und `ical4j` (`org.mnode.ical4j:ical4j:1.0.6`)
- Ergänze das `spring-boot-maven-plugin`. Dieses Plugin sorgt dafür, dass die Anwendung zusammen mit einem Applicationserver in eine `jar` Datei gepackt wird und direkt gestartet werden kann.
- Teile Maven mit, dass das Projekt Java 8 kompatibel sein soll. Setze dafür die properties `maven.compiler.source` und `maven.compiler.target` auf den Wert 1.8
- Teile Maven mit, dass es für die Sourcen das Encoding UTF-8 verwenden soll. Dafür existiert wie für die Java Version ein vordefiniertes Property.
- Baue die Anwendung und starte sie mittels `java -jar <artifactId>.jar`

# Repositories

# Woher kommen Dependencies & Plugins?

- werden zunächst in einem lokalen Maven Cache auf der Festplatte gesucht
  - Lokales Repository
  - befindet sich üblicherweise unter `~/ .m2/ repository`
- wenn dort nicht vorhanden im Internet oder einem Firmenserver
- Maven Central
  - standardmässig eingebunden
  - beinhaltet sehr viele Open-Source Libraries/Plugins

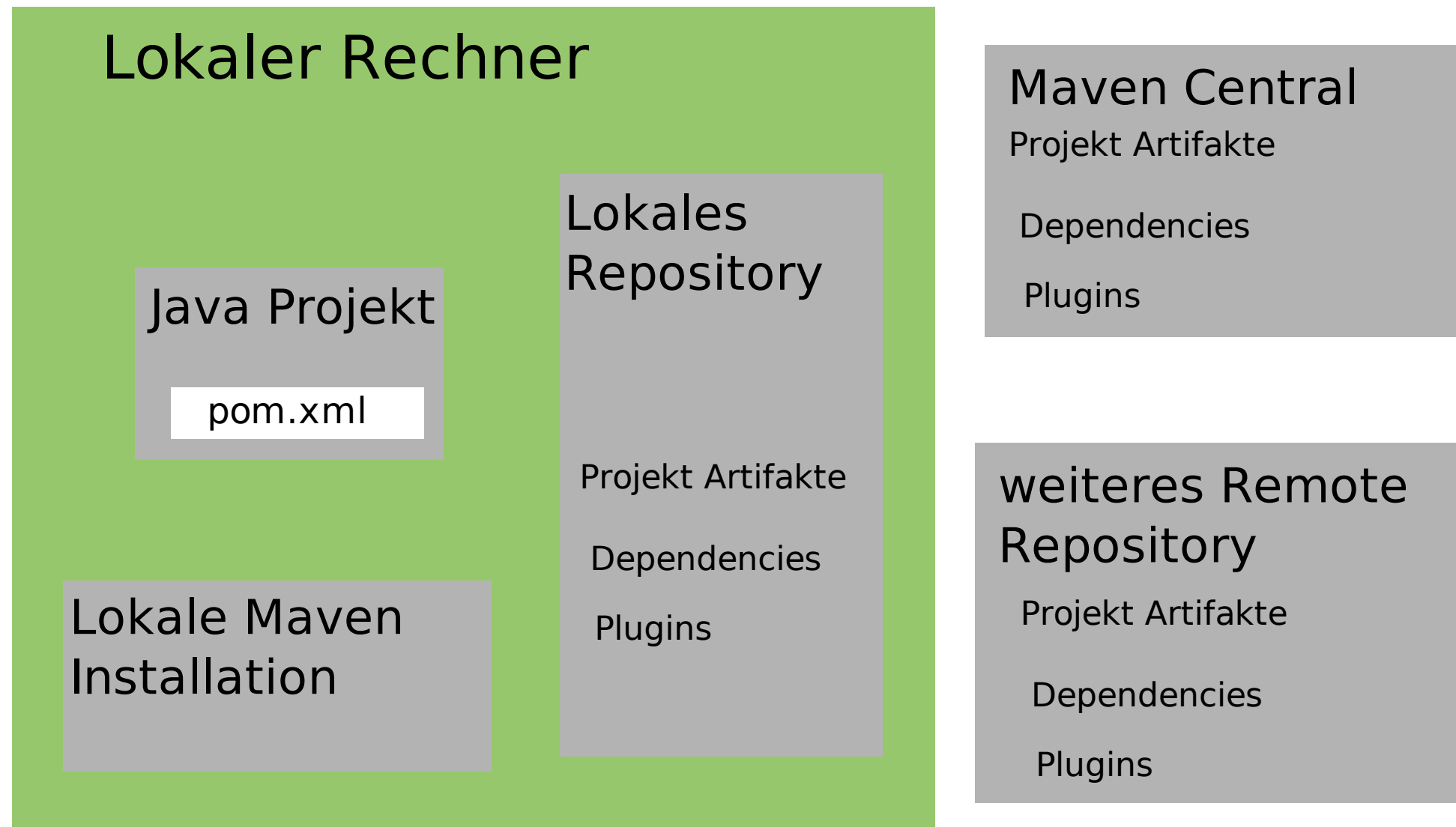
# Wo werden gebaute Artifakte abgelegt?

- `mvn install` kopiert das gebaute Artefakt in das Lokale Repository
- `mvn deploy` lädt das gebaute Artefakt auf einen Repository Server hoch
  - Konfiguriert in `<distributionManagement>`

# Beispiel Lokales Repository

```
.m2
├── repository
│   ├── antlr
│   │   └── antlr
│   │       ├── 2.7.2
│   │       │   ├── antlr-2.7.2.jar
│   │       │   ├── antlr-2.7.2.jar.sha1
│   │       │   ├── antlr-2.7.2.pom
│   │       │   ├── antlr-2.7.2.pom.sha1
│   │       │   └── _maven.repositories
│   │       └── 2.7.7
│   │           ├── antlr-2.7.7.jar
│   │           ├── antlr-2.7.7.jar.sha1
│   │           ├── antlr-2.7.7.pom
│   │           ├── antlr-2.7.7.pom.sha1
│   │           └── _maven.repositories
│   └── aopalliance
│       └── aopalliance
│           └── 1.0
│               ├── aopalliance-1.0.jar
│               ├── aopalliance-1.0.jar.sha1
│               ├── aopalliance-1.0.pom
│               ├── aopalliance-1.0.pom.sha1
│               └── _maven.repositories
└── ...
```

# Bestandteile von Maven





# Privates Repository

- Warum?
  - Firmeneigene Bibliotheken verwalten
  - Verwaltung von Bibliotheken die nicht im Internet verbreitet werden dürfen
  - Zugriffskontrolle
- Repository Software
  - Nexus: <http://www.sonatype.org/nexus/>
  - Artifactory: <http://www.jfrog.com/artifactory/>

# Privates Repository einbinden

Von welchem Server sollen benötigte Dependencies und Plugins heruntergeladen werden?

```
<repositories>
  <repository>
    <id>informatica</id>
    <url>http://52.18.220.227:8081/nexus/content/repositories/releases/</url>
  </repository>
</repositories>
```

# Nexus - Manuelles Hochladen

The screenshot displays the Sonatype Nexus web interface. The top navigation bar includes the Sonatype logo, the text 'Sonatype Nexus', and a user profile 'admin' with a dropdown arrow. The version 'Sonatype Nexus™ 2.11.4-01' is also visible.

The left sidebar contains a 'Nexus' menu with the following options: 'Artifact Search', 'Advanced Search', 'Views/Repositories', 'Security', 'Administration', and 'Help'. The 'Views/Repositories' section is expanded, showing 'Repositories', 'Repository Targets', 'Routing', and 'System Feeds'.

The main content area is titled 'Repositories' and features a table of repository configurations. The table has columns for 'Repository', 'Type', 'Health Check', 'Format', 'Policy', 'Repository Status', and 'Repository Path'. The '3rd party' repository is selected, and the 'Artifact Upload' tab is active.

The 'Artifact Upload' tab contains the following sections:

- Select GAV Definition Source:** A dropdown menu for 'GAV Definition' with a 'Select...' button and a help icon.
- Select Artifact(s) for Upload:** A section with a 'Select Artifact(s) to Upload...' button, input fields for 'Filename:', 'Classifier:', and 'Extension:', and an 'Add Artifact' button.
- Artifacts:** A table for listing artifacts to be uploaded, with 'Remove' and 'Remove All' buttons.
- Upload Artifact(s):** A button to initiate the upload process, along with a 'Reset' button.

Repository	Type	Health Check	Format	Policy	Repository Status	Repository Path
Public Repositories	group	ANALYZE	maven2			http://localhost:8081/nexus/content/groups/public
3rd party	hosted	ANALYZE	maven2	Release	In Service	http://localhost:8081/nexus/content/repositories/thirdparty
Apache Snapshots	proxy	ANALYZE	maven2	Snapshot	In Service	http://localhost:8081/nexus/content/repositories/apache-snapshots
Central	proxy	ANALYZE	maven2	Release	In Service	http://localhost:8081/nexus/content/repositories/central
Central M1 shadow	virtual	ANALYZE	maven1	Release	In Service	http://localhost:8081/nexus/content/shadows/central-m1
Codehaus Snapshots	proxy	ANALYZE	maven2	Snapshot	In Service	http://localhost:8081/nexus/content/repositories/codehaus-snapshots
Releases	hosted	ANALYZE	maven2	Release	In Service	http://localhost:8081/nexus/content/repositories/releases
Snapshots	hosted	ANALYZE	maven2	Snapshot	In Service	http://localhost:8081/nexus/content/repositories/snapshots

# Repository für Deployment einbinden

Auf welchen Server sollen gebaute Artifakte hochgeladen werden?

```
<distributionManagement>
  <repository>
    <id>informatica</id>
    <url>http://52.18.220.227:8081/nexus/content/repositories/releases/</url>
  </repository>
  <snapshotRepository>
    <id>informatica</id>
    <url>http://52.18.220.227:8081/nexus/content/repositories/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

# Settings

# settings.xml

- Beinhaltet Konfigurationseinstellungen die nicht in der pom.xml gespeichert werden sollten
  - umgebungsabhängig
  - vertraulich
- Ablageorte
  - Global in der Maven Installation: `$M2_HOME/conf/settings.xml`
  - Im User Verzeichnis: `~/.m2/settings.xml`
- Inhalt
  - Username/Passwort zum Zugriff aufs Unternehmens-Repository
  - Proxyeinstellungen
  - Unternehmens-Repository für Plugins und Dependencies
  - Mirror für Repository Server
  - Globale Profile

<https://maven.apache.org/settings.html>

# Beispiel settings.xml

```
<settings>
  <servers>
    <server>
      <id>server001</id>
      <username>my_login</username>
      <password>my_password</password>
    </server>
  </servers>
  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
    </proxy>
  </proxies>
</settings>
```

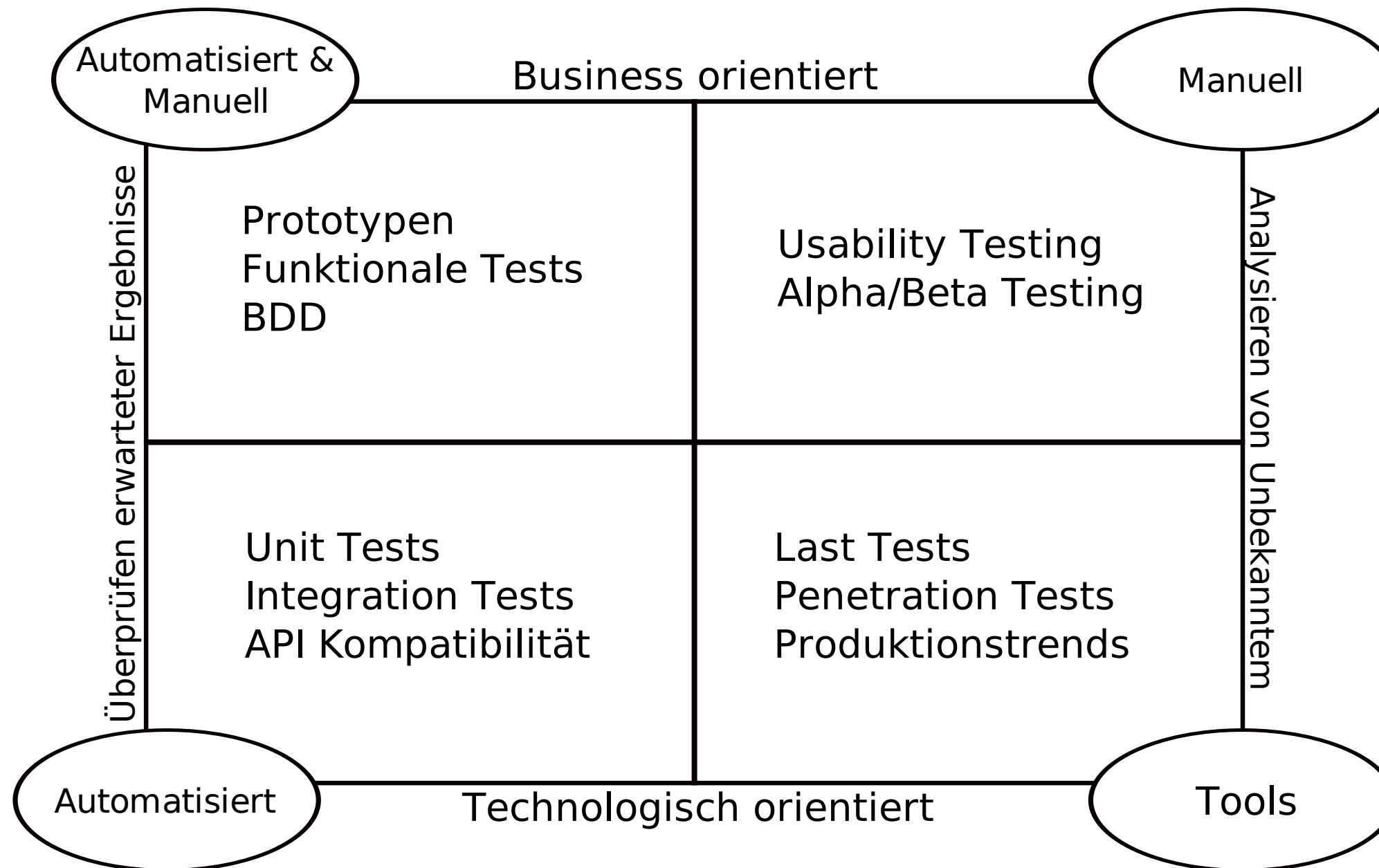
# Übung 3

- Erweitere die Konfiguration aus der vorherigen Übung um alle notwendigen Angaben um die Anwendung zu einem Repository Server zu deployen (URL `http://52.18.220.227:8081/nexus/content/repositories/releases/`)
- Deploye deine Anwendung mittels `mvn deploy` auf den Repository Server
- Melde dich unter `http://52.18.220.227:8081/nexus` am Server an und prüfe ob deine Anwendung hochgeladen wurde



Testen

# Testarten



Angelehnt an: The Agile Testing Quadrant

# Tests in Maven

- Standardphasen im Lifecycle
  - `test`: (Unit-)Tests die schnell ausgeführt werden können
  - `integration-test`: langsame Tests, für die eine komplexe Integrationsumgebung erforderlich ist (z.B. Datenbank, Applikationsserver)
  - `verify`: Abschließende Checks

# test Phase

- Surefire Plugin (Default)
- Tests im Verzeichnis `src/test`
  - Package Struktur analog zu `src/main`
- Schlagen Tests fehl, wird der Build nach Beendigung der Test Phase abgebrochen

# Surefire Plugin

<https://maven.apache.org/surefire/maven-surefire-plugin/>

- Default-mässig werden die folgenden Klassennamen berücksichtigt
  - `**/*Test.java`
  - `**/Test*.java`
  - `**/*TestCase.java`
- Ergebnis Reports: `target/surefire-reports`
- JUnit oder TestNG
  - Müssen als dependency hinzugefügt werden (Scope: test)

# nützliche Kommandozeilenoptionen

```
mvn test  
mvn test -Dtest=MyUnitTest  
mvn install -Dmaven.test.skip=true  
mvn install -DskipTests
```

# integration-test Phase

- längerdauernde Tests
  - Hochfahren des Spring Containers
- Zusammenspiel mit anderen Diensten testen
  - Testen der Webanwendung via HTTP, z.B. in Tomcat
  - Testen der Datenspeicherung, z.B. in PostgreSQL

# Failsafe Plugin

- Per Konvention werden Klassen die mit `IT` beginnen oder enden als Integrationstests ausgeführt
- Phasen im Detail
  - `post-integration-test`: Vorbereitungen, z.B. Tomcat oder Datenbank starten
  - `integration-test`: Testausführung
  - `post-integration-test`: Aufräumen, z.B. Tomcat oder Datenbank wieder herunterfahren
  - `verify`: Auswerten der Testergebnisse



# nützliche Kommandozeilenoptionen

```
mvn verify  
mvn -DskipTests  
mvn -DskipITs
```

Prüfen

# Idee

- Build fehlschlagen lassen, wenn bestimmte Bedingungen nicht erfüllt sind
  - Buildvoraussetzungen
    - z.B. Java-Version, Maven Version
  - Qualität des Buildskripts
    - z.B. fehlendes Dependency Management
  - Codequalität
    - z.B. Codeduplizierung
- Lifecycle-Phasen
  - validate
  - verify

# Maven Enforcer Plugin

- Stellt Tasks zur Verfügung um die Umgebung und die pom.xml zu prüfen
  - Umgebung: Existenz von Dateien, Java-Version, Maven-Version etc.
  - pom.xml: Dependency Convergence

<https://maven.apache.org/enforcer/maven-enforcer-plugin/usage.html>

# Beispiel: Build nur mit Maven Version 3.0

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.4</version>
  <executions>
    <execution>
      <id>enforce-maven</id>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <requireMavenVersion>
            <version>3.0</version>
          </requireMavenVersion>
        </rules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

# Codequalität prüfen

- Formatierungsstandards
- Testabdeckung
- Komplexität des Quellcodes
- Code Duplizierung
- Toter Code

# Statische Codeanalyse - Standardtools

- PMD
  - Findet v.a. "unsauberen" Code, z.B. leere try-catch Blöcke oder ungenutzten Code
  - Erweiterung: Copy-Paste-Detector
- Checkstyle
  - Prüft hauptsächlich den Programmierstil, z.B. Einrückungen oder Namenskonventionen
  - Oftmals Anpassung der Regeln notwendig (Zeilenlänge, final)
- Findbugs
  - Sucht nach Fehlern, z.B. null Referenzen
  - Kompiliert Projekt ein zweites Mal und analysiert & instrumentiert Bytecode
  - Verlangsamt Build deutlich
- Jacoco
  - Testabdeckung

# Codequalität - PMD

## Konfiguration mit Standard Ruleset

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <version>3.4</version>
  <configuration>
    <verbose>true</verbose>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



# Testabdeckung

- `jacoco-maven-plugin`:
  - komplexe Konfiguration
  - benötigt zwei `<execution>` Einträge
    - der erste bestimmt wann der Jacoco Agent mit der Erfassung beginnen soll
    - der zweite wann sie beendet werden soll und welche Regeln ausgeführt werden sollen
  - benötigt viel Speicher
  - arbeitet mit einigen Mockframeworks (z.B. PowerMock) nicht zusammen

<http://www.eclemma.org/jacoco/trunk/doc/maven.html>

# Beispiel Konfiguration

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.4.201502262128</version>
  <executions>
    <execution>
      <id>prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>check</id>
      <goals>
        <goal>check</goal>
      </goals>
      <configuration>
        <rules>
          ???
        </rules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

# Konfiguration

- Hilfe zur Konfiguration
  - `mvn help:describe -Dplugin=org.jacoco:jacoco-maven-plugin -Ddetail`
- Coverage Rules
  - Können auf Zeilen, Branches oder Methodenebene angewandt werden
  - Einzelne Klassen können von der Prüfung ausgeschlossen werden

# Übung 4

- Ergänze eine dependency zu `spring-boot-starter-test` mit dem Scope `test`
- Definiere ein Property für die Spring Version
- Es gibt insgesamt 7 Tests für die Anwendung. Sorge dafür dass alle ausgeführt werden.
- Der Build soll nur dann erfolgreich sein, wenn 90% Testcoverage erreicht werden.
- Verwende das Maven Enforcer Plugin um ein DependencyManagement zu erzwingen (Rule: `dependencyConvergence`)