Software Engineering II

3 Konfigurationsmanagement-Werkzeuge

SS 2020

Prof. Dr. Dirk Müller

Konfigurationsmanagement

- Versionsmanagement
  - Geschichte und Auswahl
  - Popularität
  - *Subversion*
  - *Mercurial*
- *System Building*
  - Geschichte und Auswahl
  - Popularität
  - *make*
  - *Ant*
  - *Maven*
  - *Gradle*
- Vergleich und Zusammenfassung

Versionsmanagement-Werkzeuge: Geschichte und Auswahl

1972 — Source Code Control System (SCCS) is launched, developed in SNOBOL at Bell Labs by Marc Rochkind.

1982 — Revision Control System (RCS) is released. RCS is still maintained by the GNU Project.

1990 — Initial release of Concurrent Versions System (CVS) version control system

2000 — Subversion (SVN) is launched by CollabNet.

2001 — The first SVN source code is hosted.

2004 — SVN version 1.0 is released.

26 March 2005 — Bazaar (then "Baz") is released under Canonical's sponsorship.

7 April 2005 — Git is released as an open source project headed by Linus Torvalds, the namesake of Linux.

19 April 2005 — Mercurial project is launched a few days after Git, also designed for use with Linux development.

2008 — The final stable release of CVS marks the end of an era (18 years).

https://webinerds.com/version-control-systems-keep-your-code-in-order/

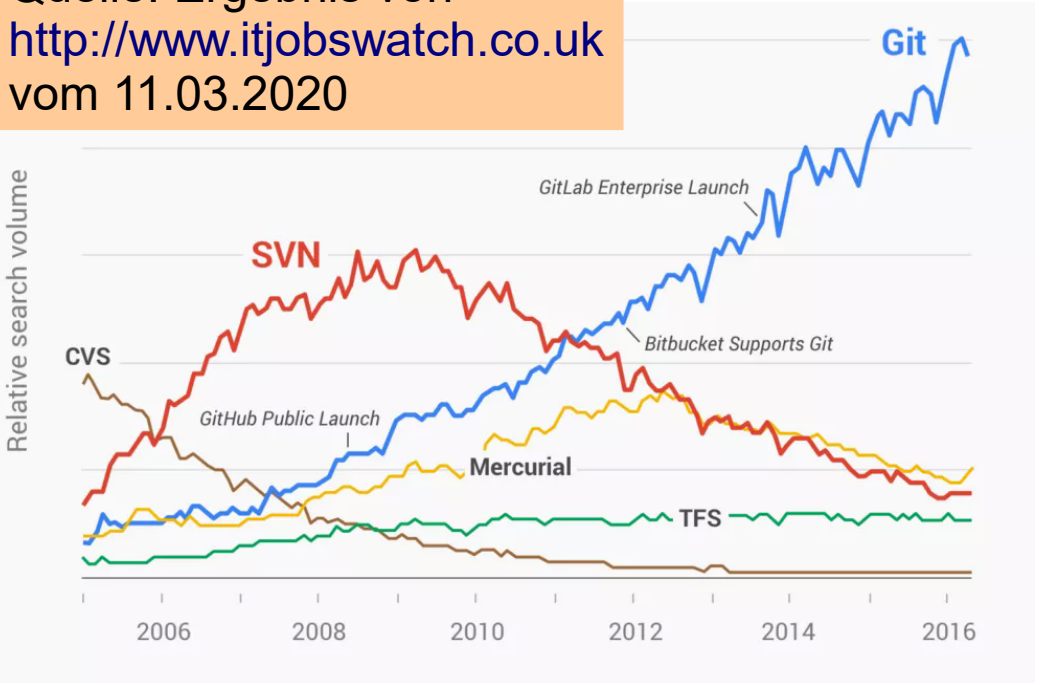
- zwei Repräsentanten des aktuell genutzten zentralen und verteilten Ansatzes ausgewählt
- *Mercurial* statt *Git* gewählt
  - bessere *Windows*-Unterstützung
  - bessere Integration in IDEs
  - leichter erlernbar

Versionsmanagement-Werkzeuge: Popularität

Skill / Job Role (Historical trends & salary statistics)	Rank 6 Months to 11 Mar 2020	Rank Change Year-on-Year	Median Salary 6 Months to 11 Mar 2020	Median Salary % Change Year-on-Year	Historical Permanent Job Ads	Live Job Vacancies
Mercurial	848	▼ -46	£50,000	+5.26%	149 (0.12%)	40
Git (software)	29	▲ +3	£52,500	+5.00%	8,260 (6.81%)	2,040
Apache Subversion (SVN)	345	▼ -69	£52,500	+5.00%	1,030 (0.85%)	271

- Verteilte haben zentrale Ansätze 2013 überholt.
- *GitHub* mit dem freien Hosting von Open-Source-Projekten als Erfolgsfaktor
  - Ende 2018 von Microsoft übernommen
  - *GitLab* als Alternative
- *Git* seit 2017 von *Microsoft* für *Windows*-Entwicklung!

Quelle: Ergebnis von http://www.itjobswatch.co.uk vom 11.03.2020

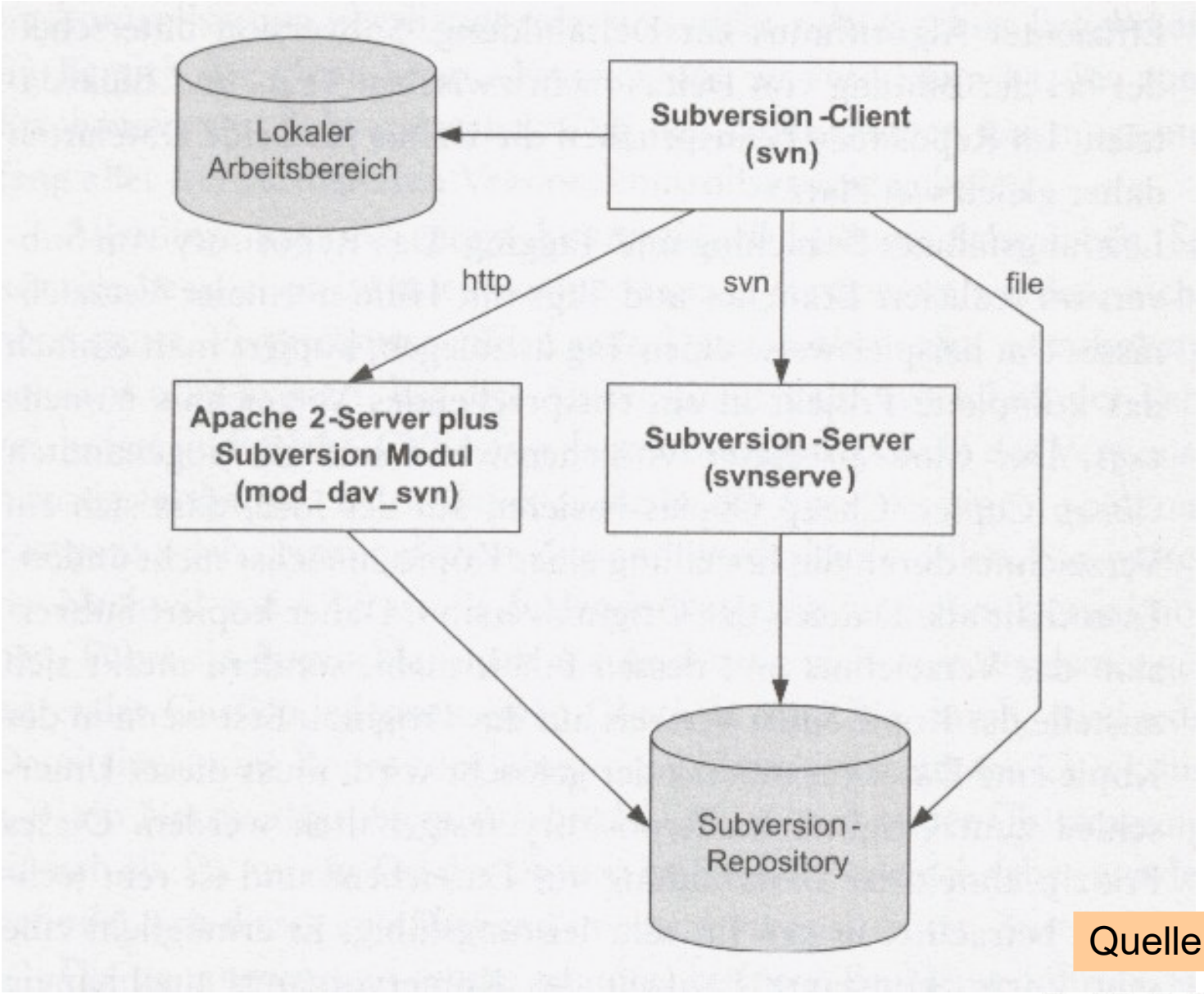


Quelle: [8]

Subversion

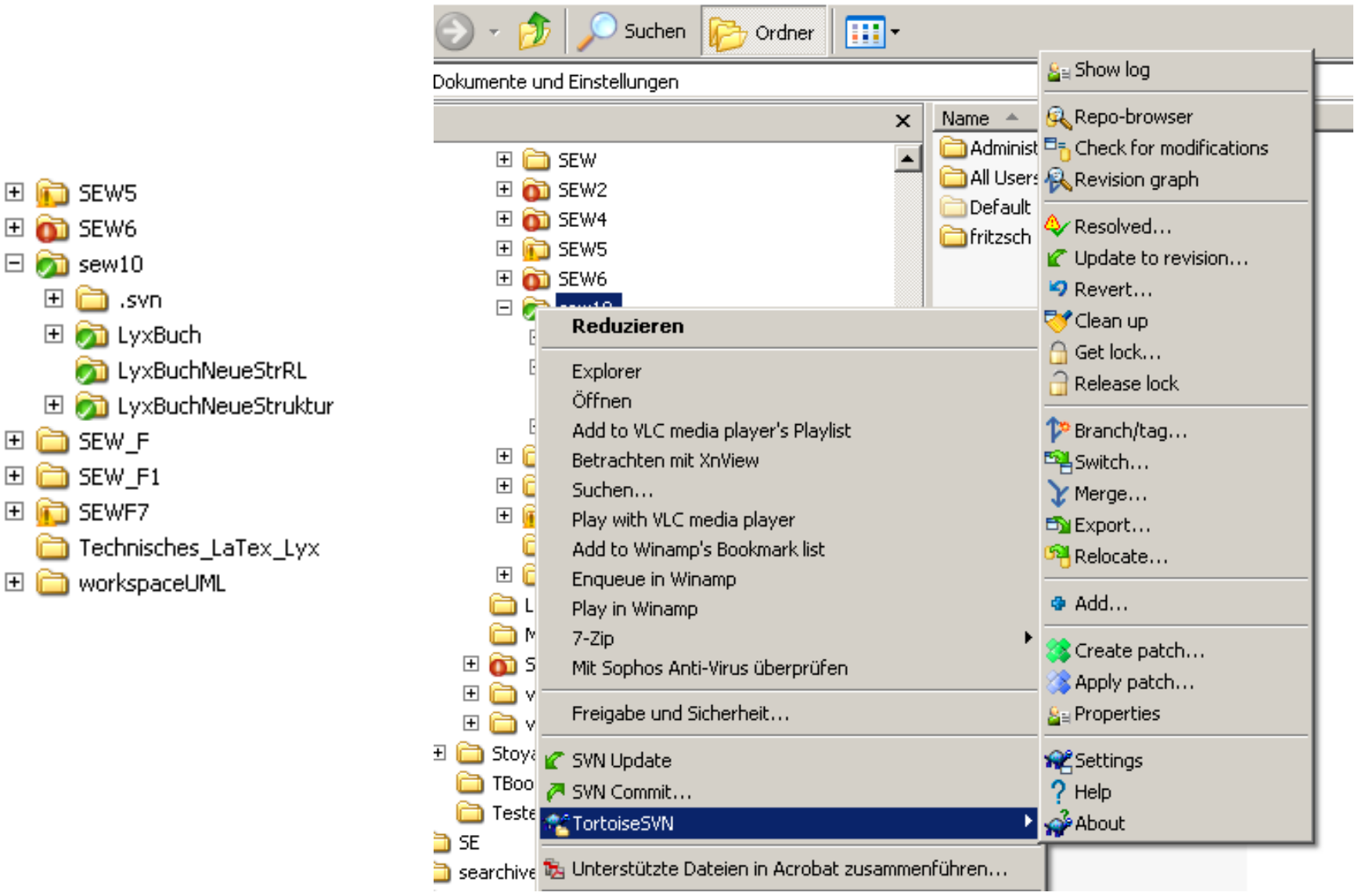
- *Open-Source*-Projekt seit ca. 2000, veröffentlicht 2004
- aktuelle Version: 1.13.0 (vom 30.10.2019)
- Motivation: Vorgänger *Concurrent Versions System*, CVS mit erheblichen Schwächen
  - nur Dateien, keine Verzeichnisse als Konfigurations-Items
  - Dateien umbenennen, verschieben, löschen oder kopieren nicht sauber unterstützt
  - keine Unterstützung von Transaktionen
- alle 3 Schwächen in *Subversion* behoben, Highlights:
  - vollständige Versionshistorie, auch Verzeichnisse, Löschen, etc.
  - Einchecken als Transaktion, erfüllt ACID-Eigenschaften
  - effizienter Delta-Algorithmus für Text- und Binärdateien
  - „billige Kopie“: verzögertes Kopieren, gleiche Teile zunächst über harte Links realisiert, erst bei Änderung neue Dateien angelegt

Client-Server-Architektur von *Subversion*

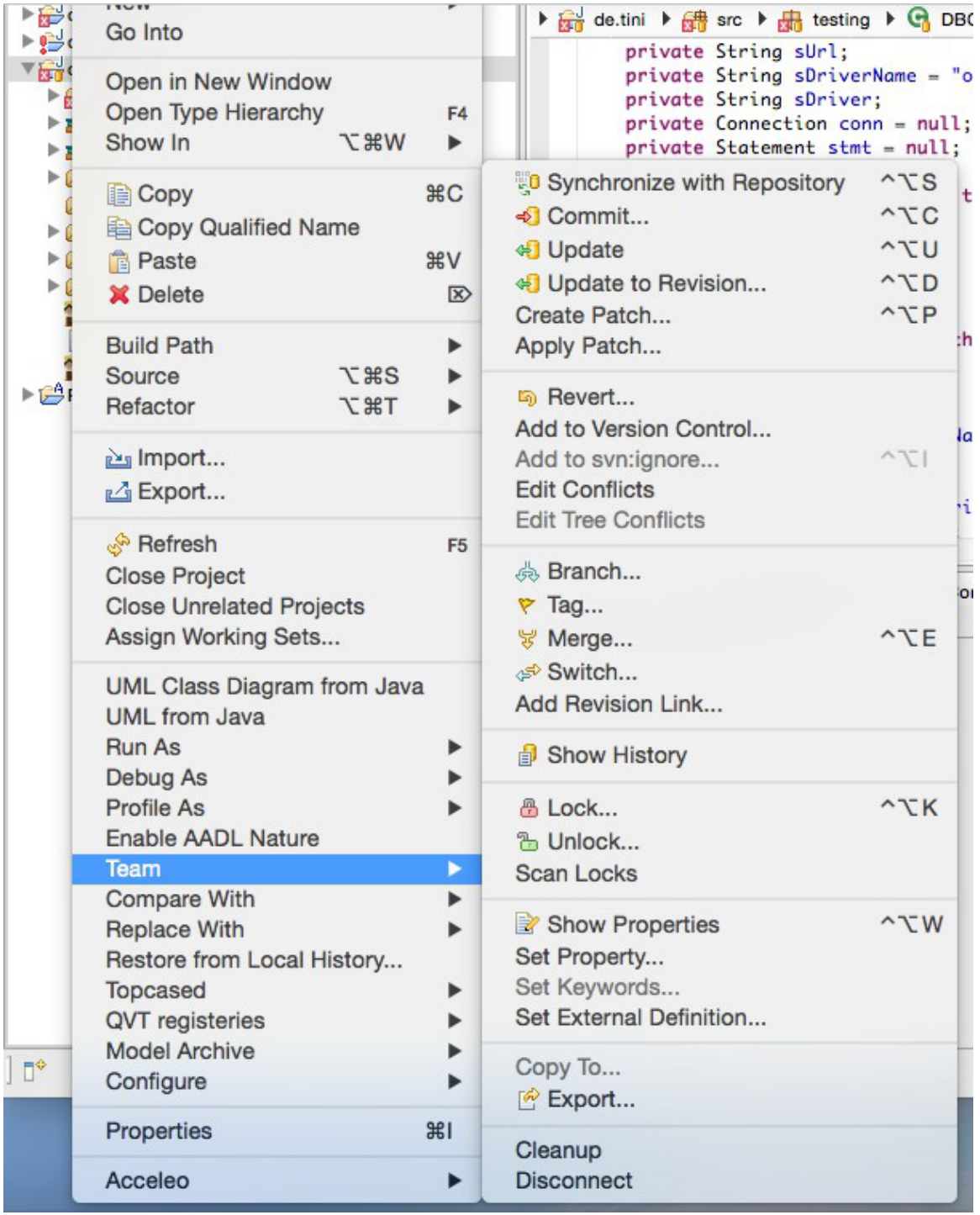


Quelle: G. Popp

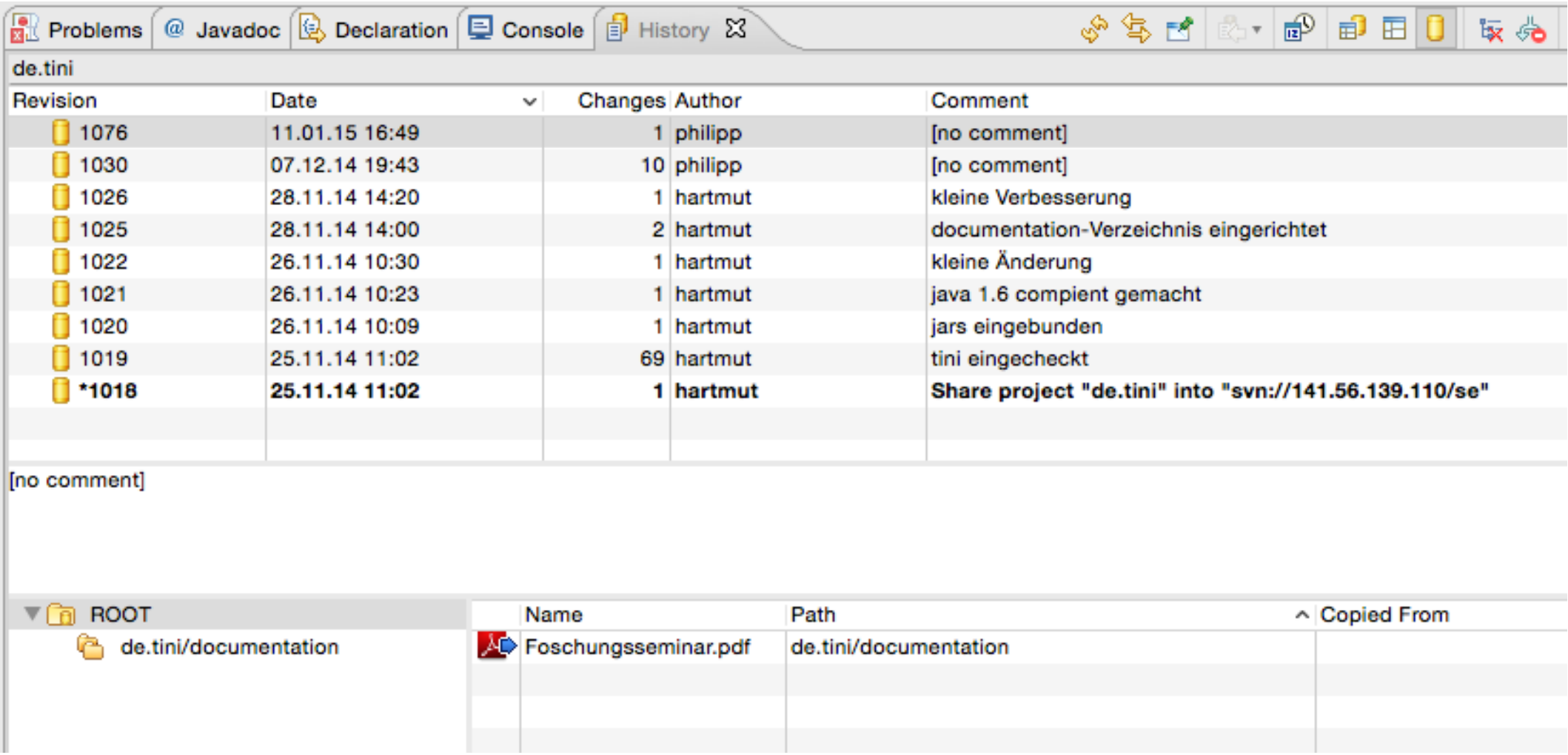
Frontend *TortoiseSVN* unter Windows



Subclipse als svn-Client unter *Eclipse*



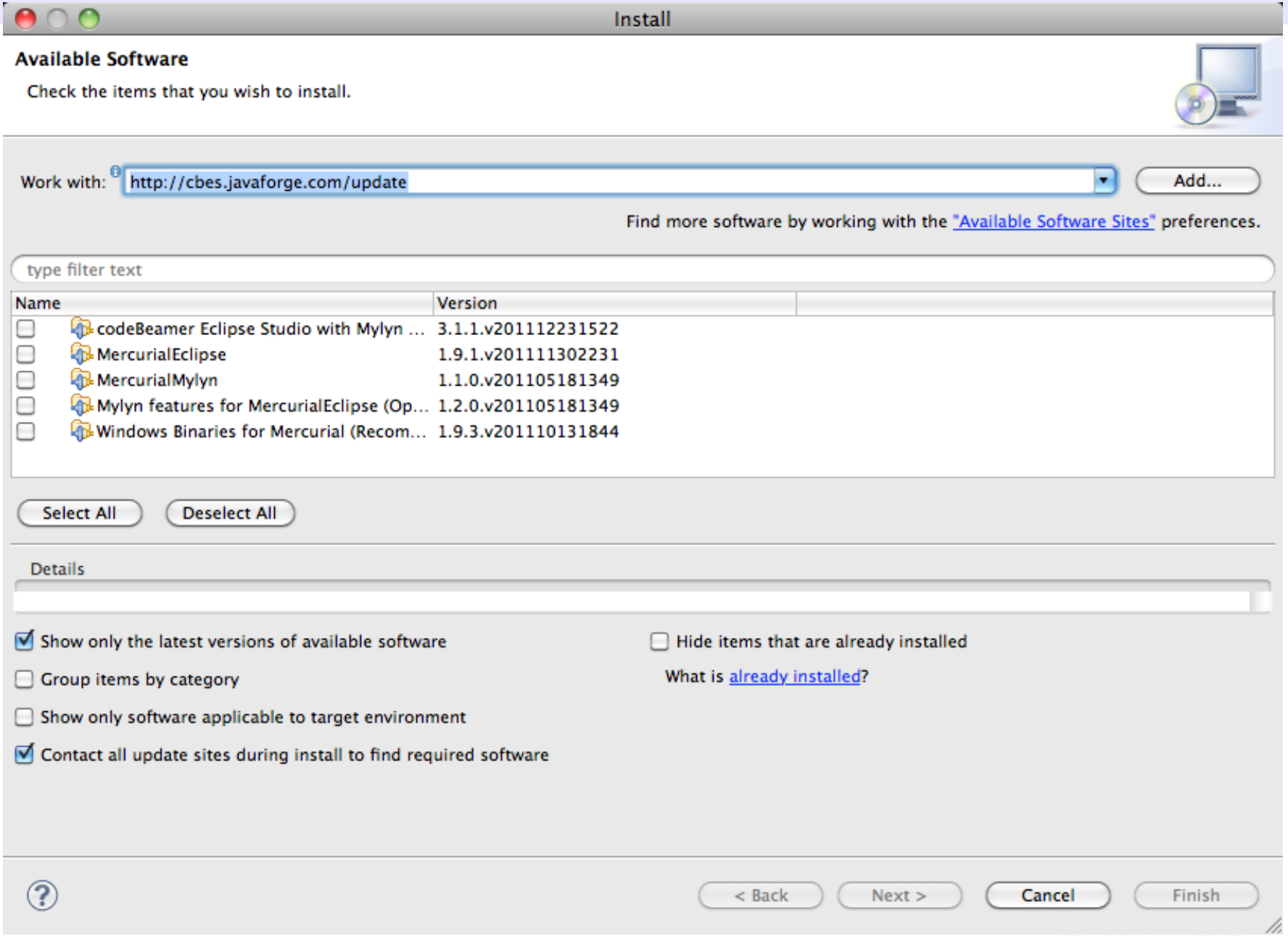
Subversion: Beispiel für Versionsgeschichte



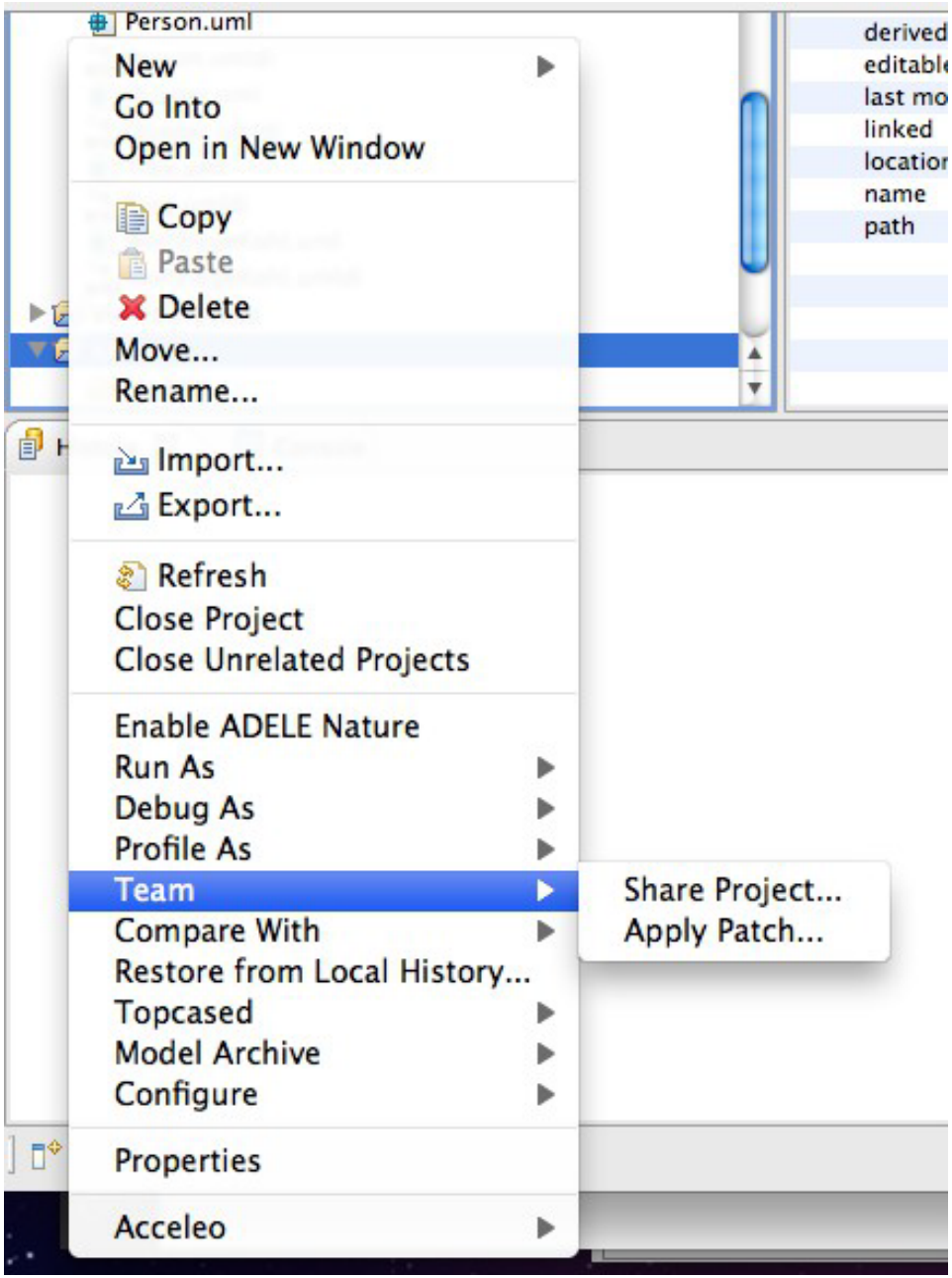
Mercurial

- *Open-Source*-Projekt seit 2005
- aktuelle Version: 5.3 (vom 3.2.2020)
- verteiltes Versionsmanagementsystem (wie *Git*)
  - *Repository* liegt als *Clone* auf dem lokalen Rechner
  - *Repository* (.hg) verwaltet die gesamte Historie
  - Arbeitsverzeichnis enthält aktuelle Verzeichnisse/Dateien + .hg
  - `Clone` überträgt *Repositorys* komplett
  - Änderungen an Dateien werden zu *Changeset* zusammengefasst
  - *Changeset* wird bei `Commit` zu neuer Revision
  - Anpassen unterschiedlicher *Repositorys* mittels `Push`/`Pull` und `Merge`
- leicht erlernbar, leichtgewichtig, gut skalierbar, anpassbar
- von *Facebook* genutzt

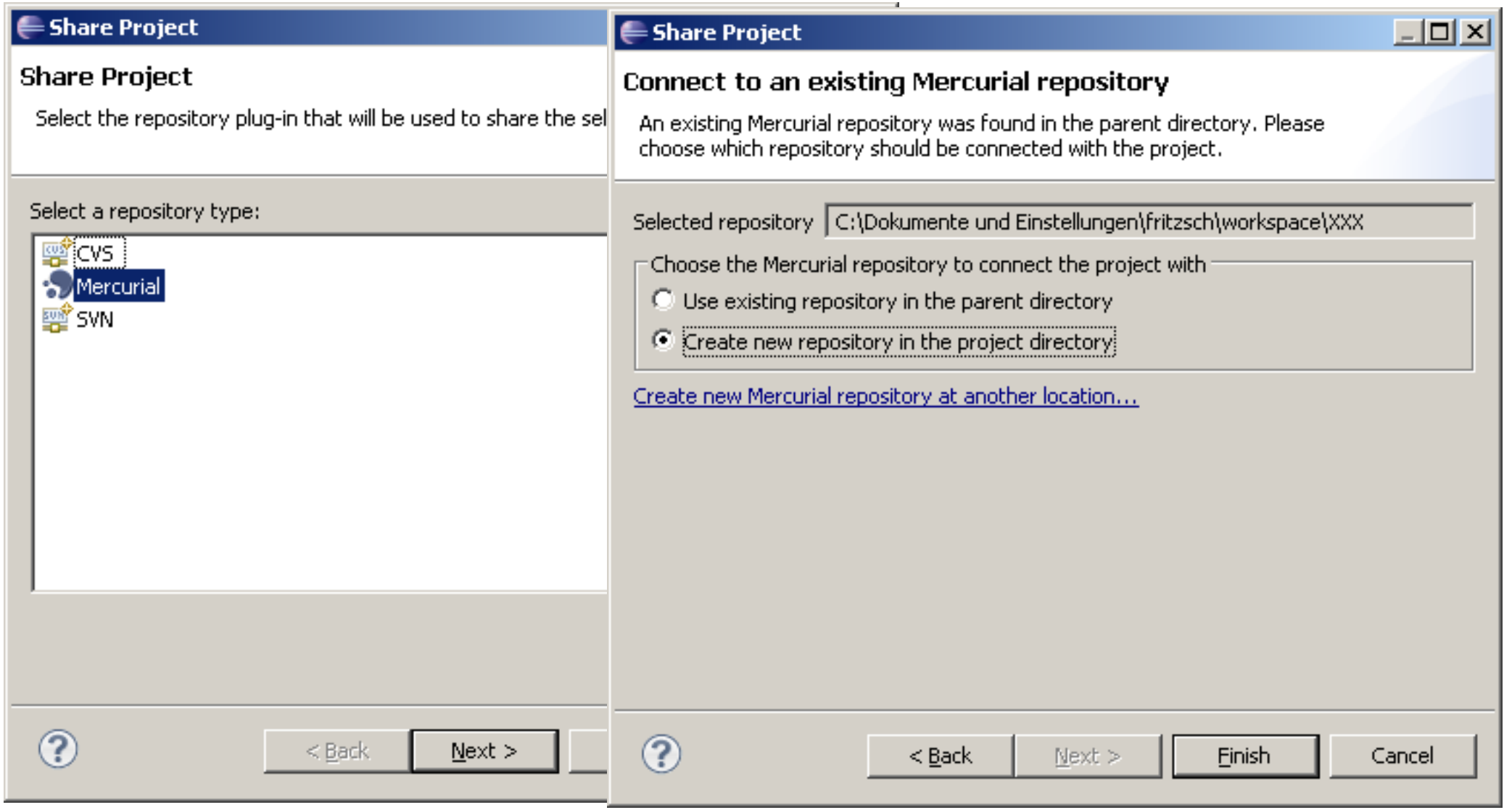
EclipseMercurial: Installation



Projekt unter *Mercurial*-Kontrolle

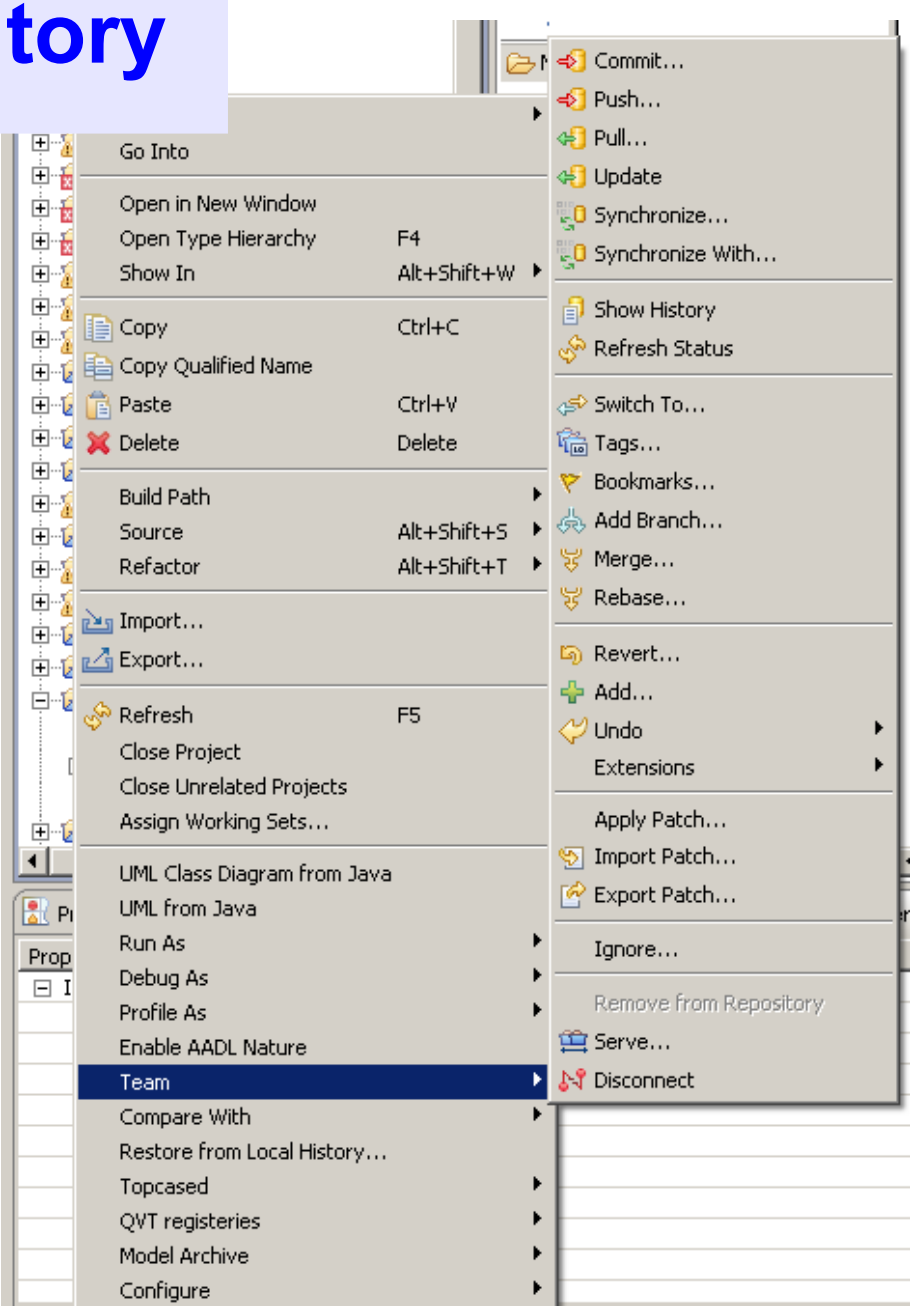


Neues *Mercurial*-Repository

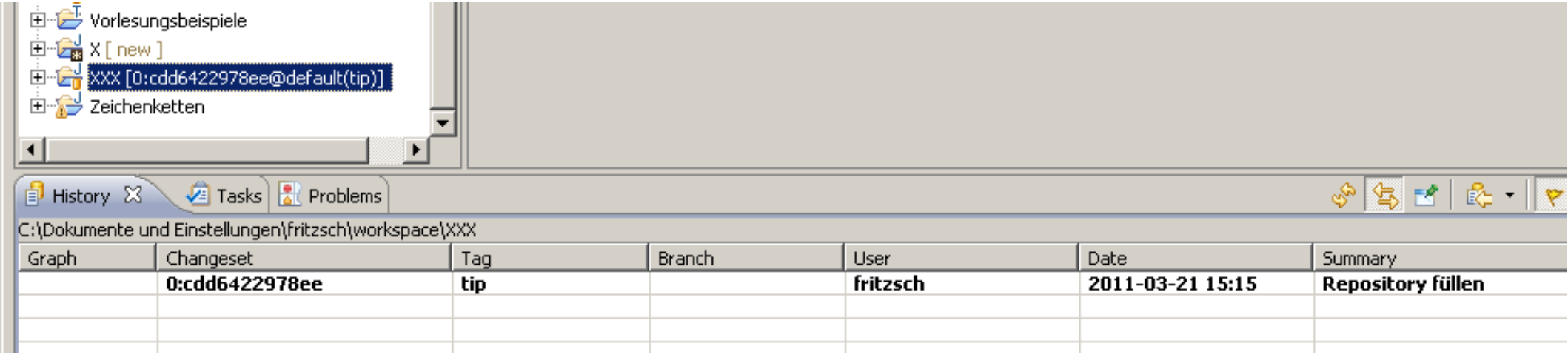


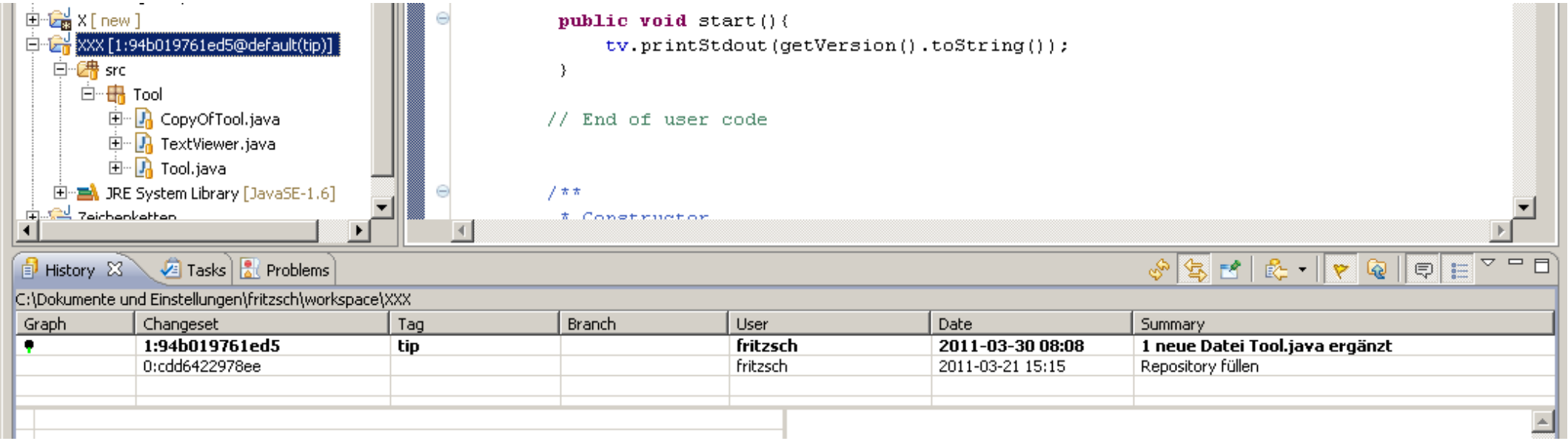
Neues *Mercurial*-Repository

.hg (hier unsichtbar)
im Verzeichnis X angelegt

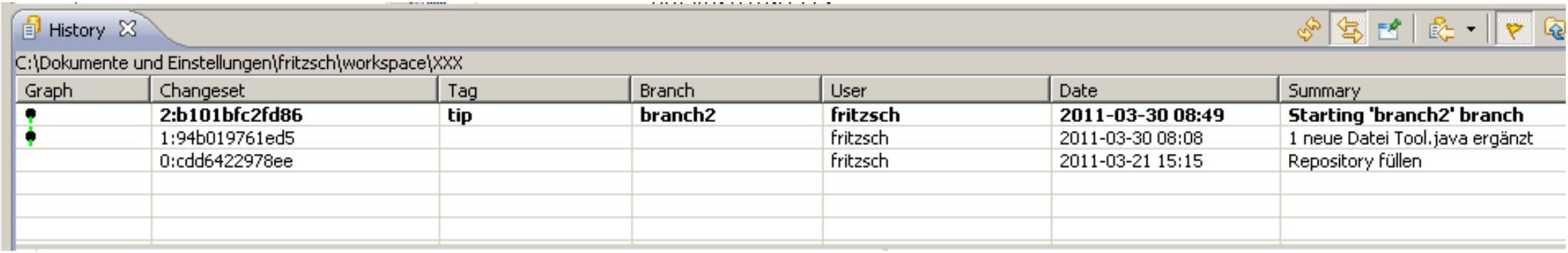


Änderungen und Historie





Branching und Bewertung

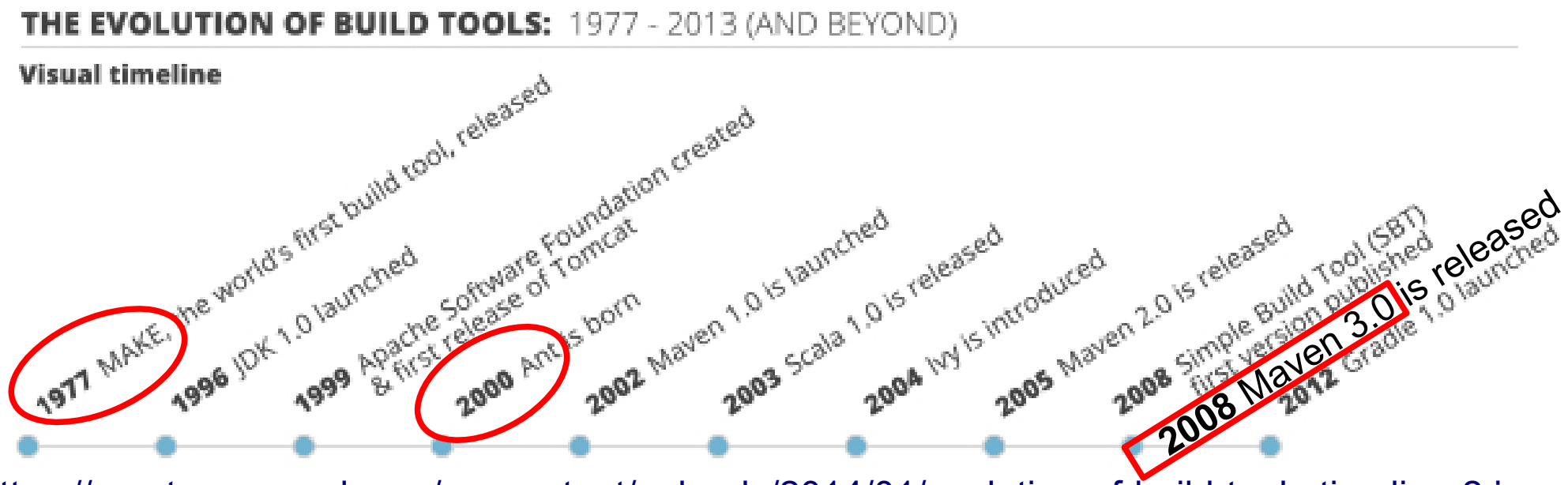


- *Mercurial* zieht Benutzerfreundlichkeit und Verfügbarkeit für alle wichtigen Plattformen von *Subversion* auf den mächtigeren verteilten Ansatz hoch.
- *Git* stärker verbreitet, aber zumindest in *Windows*-Welt *Mercurial* als relevanter Konkurrent zu sehen
- teilweise besser anpassbar und schneller als *Git* [6]
  - Zugriff auf OS, um Menge der potenziell veränderten Dateien einzuschränken (Zeitstempel und Checksummen kombiniert)

OS-Unterstützung, die *Facebook* hier nutzt

normaler Ansatz für verteilte Versionskontrolle

Build-Werkzeuge: Geschichte und Auswahl

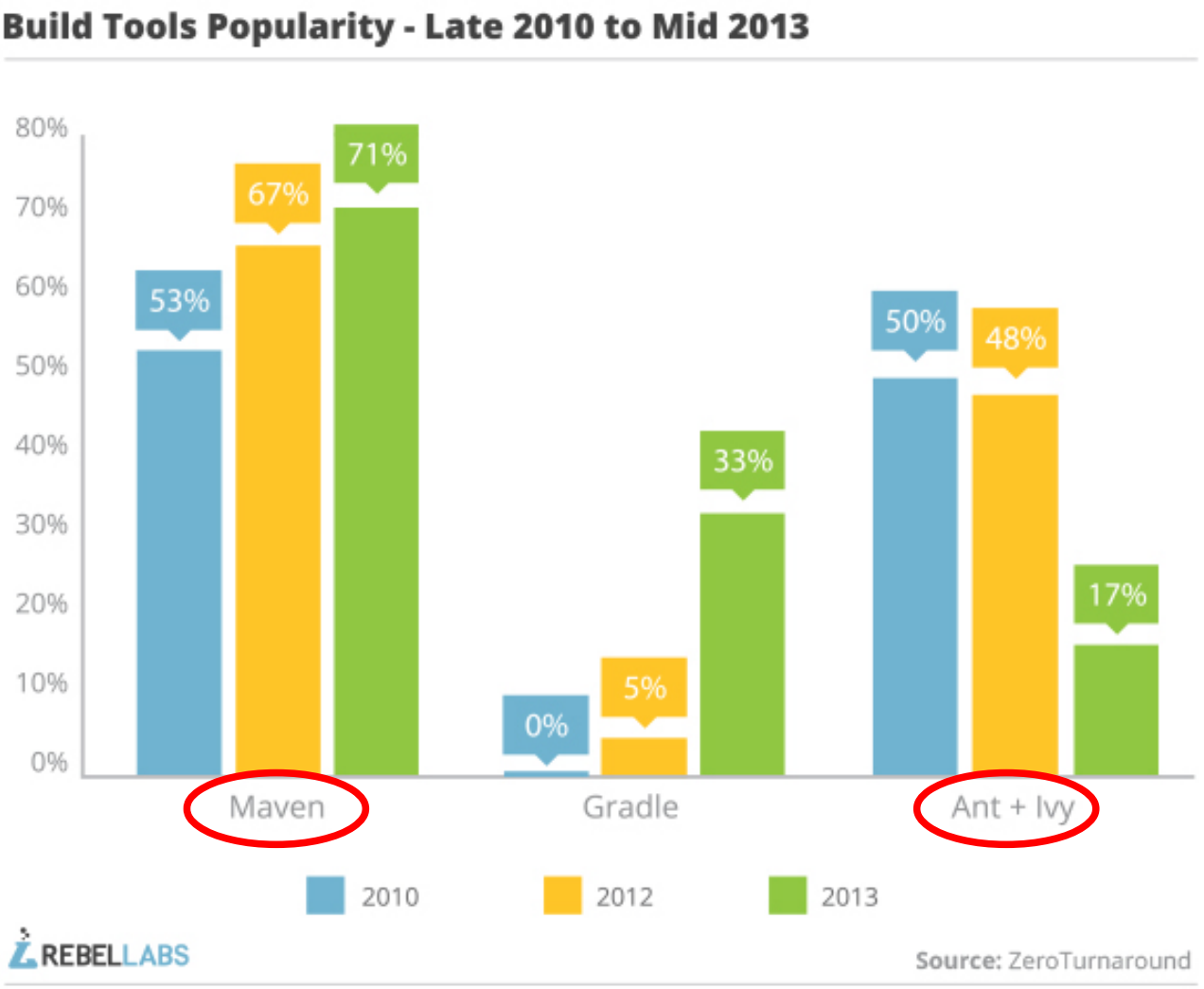


https://zeroturnaround.com/wp-content/uploads/2014/01/evolution-of-build-tools-timeline-2.jpg

- *Make* zwar alt, aber ständig weiterentwickelt
- *Ant* etwas überholt, aber als *Ant* + *Ivy* möglich
- *Maven* moderner als *Ant*, wurde weiterentwickelt
- *Gradle* scheint die Zukunft zu sein.

Build-Werkzeuge: Popularität

- bezieht sich auf *Java*-Projekte
- Wechsel von *Ant* zu *Maven* und weiter zu *Gradle*



https://zeroturnaround.com/wp-content/uploads/2014/01/build-tools-popularity-2010-to-2013.jpg

make

- Ursprung: *GNU Make* (*gmake*)
  - 1977 von *Richard Stallman* und *Roland McGrath* entwickelt
  - aktuelle Version 4.3 (vom 19. Januar 2020)
  - Standard in der *Linux*/*UNIX*-Welt, Kompilierung des *Linux*-Kernels
- ganze Familie von Werkzeugen mit Portierungen/Neuentwicklungen
  - *nmake*, Teil von *Visual Studio* unter *Windows*
  - *dmake*, Standard unter *Sun Solaris Studio*, Kompilierung von *Apache OpenOffice*
- automatische (Abhängigkeiten + Zeitstempel) Feststellung, welche Teile des Programms neu gebaut werden müssen
- deklarativer Ansatz, Suche nach einem *Makefile* im aktuellen Verzeichnis, welches Regeln enthält
- Nachteil: plattformabhängig, besser: *Ant*, *Maven, cmake*

Beispiel-*Makefile*

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
   Tab  cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
main.o : main.c defs.h
   Tab  cc -c main.c
kbd.o : kbd.c defs.h command.h
   Tab  cc -c kbd.c
command.o : command.c defs.h command.h
   Tab  cc -c command.c
display.o : display.c defs.h buffer.h
   Tab  cc -c display.c
insert.o : insert.c defs.h buffer.h
   Tab  cc -c insert.c
search.o : search.c defs.h buffer.h
   Tab  cc -c search.c
files.o : files.c defs.h buffer.h command.h
   Tab  cc -c files.c
utils.o : utils.c defs.h
   Tab  cc -c utils.c
clean :
   Tab  rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

allgemeines Schema
Ziel(e): Abhängigkeit(en)
Tab Kommando(s)

Linken
aller erstellten
Objektdateien

Kompiliere Objektdatei
aus C-Quelldateien und
C-Headerdateien

keine Abhängigkeiten,
nur Löschkommando

Quelle: [3]

Anwendung des Beispiel-*Makefiles*

- `make`
  - erstellt die ausführbare Datei `edit` basierend auf 8 Objektdateien, die wiederum von 8 C-Quelldateien und 3 C-Headerdateien abhängen
- `make clean`
  - löscht die ausführbare Datei `edit` und die 8 Objektdateien
- `make command.o`
  - kompiliert die Objektdateien `command.o` aus der C-Quelldatei `command.c` und zwei Headerdateien
  - völlig analoge Syntax und Semantik für die 7 anderen Objektdateien
- generelle Semantik: Führe Kommando(s) aus,
  falls min. 1 Abhängigkeit neuer als Ziel
  oder das Ziel nicht existiert.

Quelle: [3]

Verbesserung mittels Variablen

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
	cc -o edit $(objects)
main.o : main.c defs.h
	cc -c main.c
kbd.o : kbd.c defs.h command.h
	cc -c kbd.c
command.o : command.c defs.h command.h
	cc -c command.c
display.o : display.c defs.h buffer.h
	cc -c display.c
insert.o : insert.c defs.h buffer.h
	cc -c insert.c
search.o : search.c defs.h buffer.h
	cc -c search.c
files.o : files.c defs.h buffer.h command.h
	cc -c files.c
utils.o : utils.c defs.h
	cc -c utils.c
clean :
	rm edit $(objects)
```

- Redundanz in Makefile: dreimal identische Liste von 8 Objektdateien => Variable(n) einführen
- weniger Tipparbeit
- besser wart- und erweiterbar

Quelle: [3]

Nutzung impliziter Regeln

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
  Tab    cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
  Tab    rm edit $(objects)
```

`cc -c <x>.c -o <x>.o` als Rezept zur Kompilierung von <x>.c zu <x>.o

zur Abdeckung des Falls, dass eine Datei `clean` im Verzeichnis existiert

Quelle: [3]

Ein kompaktes *Makefile*

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
  Tab   cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h

.PHONY : clean
clean :
  Tab   rm edit $(objects)
```

C-Headerdatei `defs.h` für alle Objektdateien benötigt

C-Headerdatei `command.h` für diese 3 Objektdateien benötigt

C-Headerdatei `buffer.h` für diese 4 Objektdateien benötigt

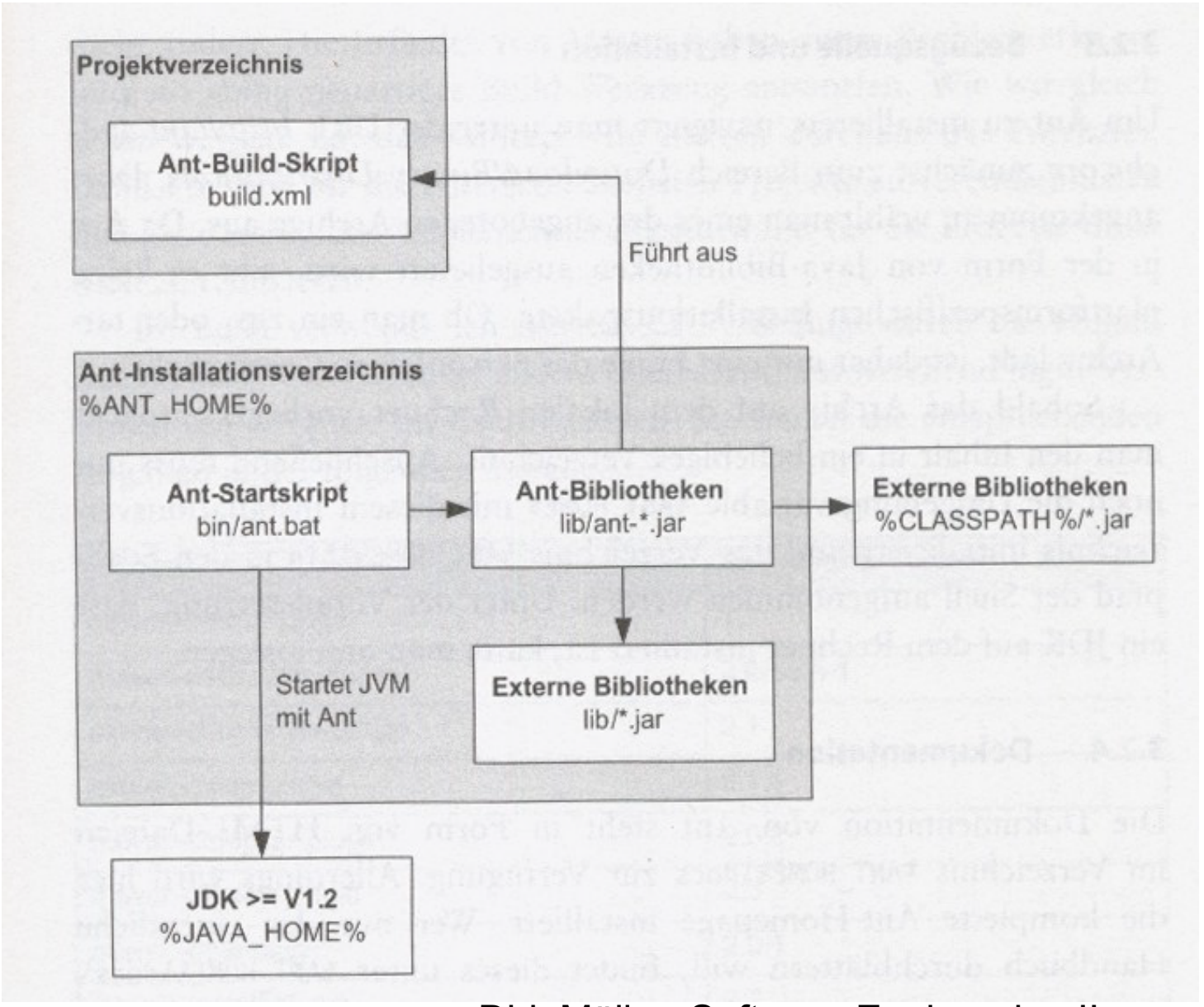
- noch weniger Redundanz, Regeln mit gemeinsamen Abhängigkeiten statt mit gem. Zielen zusammengefasst
- Geschmackssache, evtl. jetzt zu kompakt, Lesbarkeit?

Quelle: [3]

Ant

- *Open-Source*-Projekt seit 2000
- aktuelle Version 1.10.7 (vom 5.9.2019)
- Name ist Apronym: *Another Neat Tool*
  - Ameisen als kleine Tiere, die Großes leisten
- als Nachfolger (bzw. dieses für *Java*-Welt) von *make*
- besonders für (kleinere) *Java*-Projekte geeignet
- benötigt JDK (ab 1.8 empfohlen); JRE nicht ausreichend
- plattformunabhängig konzipiert
  - Dateipfade können angepasst werden
- *Ant*-Skripte als XML-Dateien
- Abhängigkeits-Manager *Ivy* seit 2004
  - Angabe in Extra-Datei `ivy.xml`

Architektur von *Ant*



Quelle: G. Popp

Aufbau eines *Ant*-Skripts

- oberstes Element beschreibt ein Projekt
- dann Schritte in Form von Zielen (*Targets*)
- Name des auszuführenden Zieles wird *Ant* als Parameter übergeben
- jedes Ziel bestehend aus einer oder mehreren Funktionen (*Tasks*), in *Java* geschrieben => erweiterbar
- Funktionen von *Ant* oder externen Bibliotheken geliefert
- Datentypen (*Types*) stellen Variablen dar, mit denen Ziele und Variablen parametrisiert werden
- Datentypen von einfachen Name-Wert-Paaren bis hin zu komplexen Typen

Beispiel eines *Ant*-Skripts

```
<?xml version="1.0"?>
<!-- ======================================================================
     25.10.2005 15:18
     21.08.2006 17:15

     Jamus
     Eine Testumgebung für Java-Applikationen.

     ====================================================================== -->
<project name="jamus" default="build" basedir="..">
        <description>
            Eine Testumgebung für Java-Applikationen.
    </description>

        <!-- - - - - - - - properties - - - - - - - - - -->

        <property name="src" value="src" />
        <property name="docs" value="doc" />
        <property name="qsm_work" value="qsm_work" />
        <property name="templates" value="templates" />
        <property name="scripts" value="scripts" />
        <property name="temp" value="temp" />

        <!-- - - - - - - - task definitions - - - - - - - - - -->

</project>
```

Def. von Eigenschaftswerten (*Properties*)

- Vermeidung fest codierter Werte (vgl. *C*-Programmierung)
- Konstanten in Form von *Properties* definieren
- Name-Wert-Paare, Werte dann nicht mehr änderbar
- *Properties* gelten global im gesamten Skript
- 6 verschiedene Arten, *Properties* zu definieren, z. B.

```
<property file= «dateiname» />
```

  lädt *Properties* aus Datei
- Zugriff auf Werte mittels `${ ... }`
- auch: vordefinierte *Properties*

Vordefinierte *Properties* (Auswahl)

- `${basedir}` Wert des *basedir*-Attributs aus *project*-Element
- `${ant.file}` Pfad des *Build*-Skriptes
- `${ant.version}` Version von *Ant*
- `${ant.project.name}` Name des Projekts im *project*-Element
- `${ant.java.version}` Version der JVM, die *Ant* ausführt
- alle *Java*-System-*Properties*, z. B. `${user.home}`

Vordefinierte *Tasks* (Auswahl)

- `javac` Kompilieren von Quellcode
- `copy` Kopieren von Dateien
- `delete` Löschen von Dateien oder Verzeichnissen
- `mkdir` Erstellen von Verzeichnissen
- `junit` automatisierte (*JUnit*-)Tests
- `move` Umbenennen von Dateien oder Verzeichnissen
- `exec` Ausführen von System-Programmen
- `zip` Zippen, also zum Komprimieren von Dateien
- `cvs` Durchführen von CVS-Operationen
- `mail` Versenden von E-Mails
- `replace` Ersetzen von Text in Dateien

somit nicht mehr plattformunabhängig

Init-Ziel aus dem *Jamus-Build*-Skript

```
<!-- - - - - - - - - - targets - - - - - - - - - - -->

<target name="init">
        <!-- Create the time stamp -->
        <!-- You can use it by typing ${DSTAMP} -->
        <tstamp />
        <!-- Create the bin directory structure used by build -->
        <mkdir dir="${bin}" />
</target>
```

Build-Ziel aus dem *Jamus-Build*-Skript

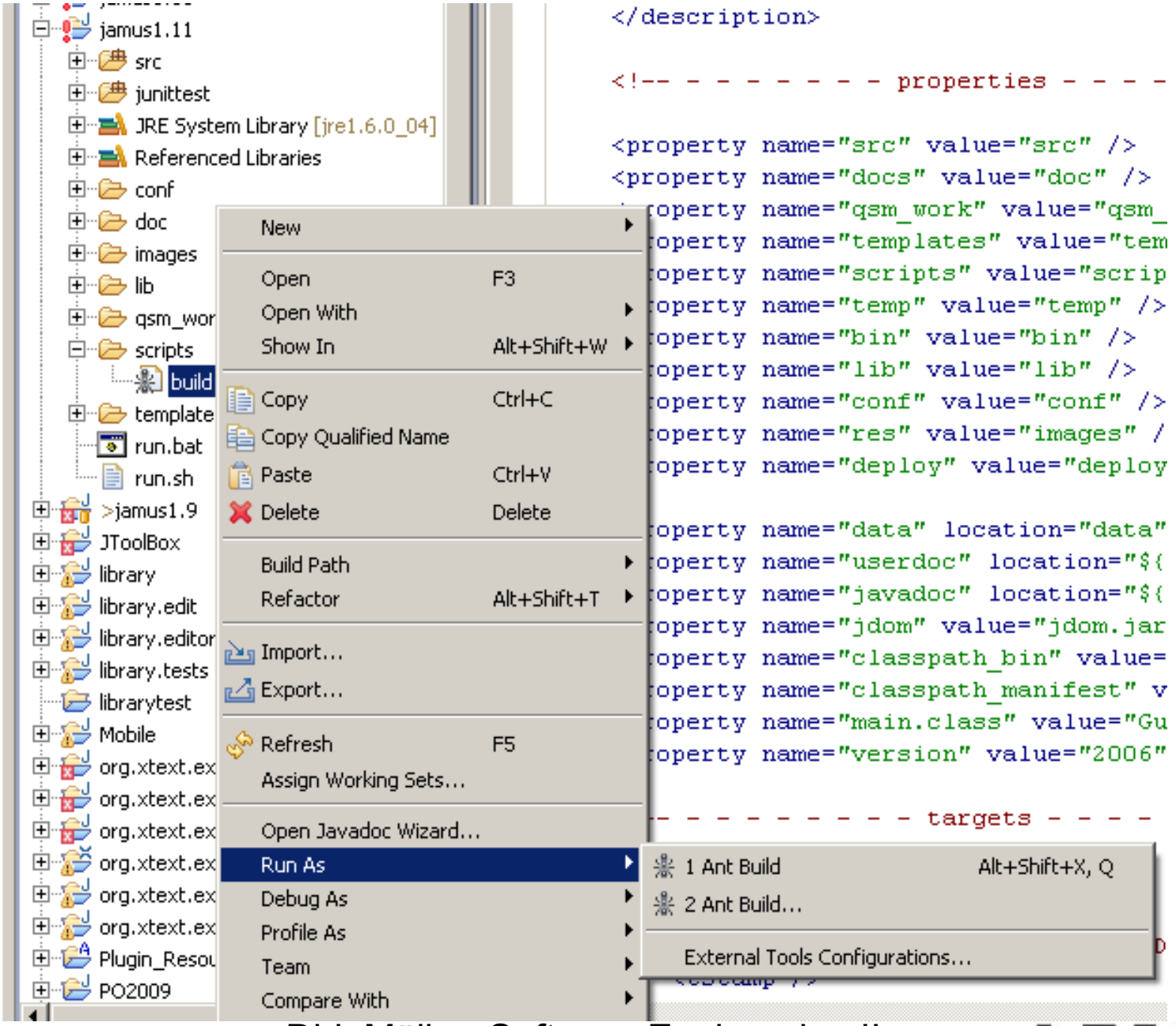
```
        <target name="build" depends="init">
                <!-- compile all sources (incl. java_cup) -->
                <javac srcdir="${src}" classpath="${classpath_bin}" destdir="${bin}"
debug="on" fork="yes" />

                <!-- run java_cup -->
                <java fork="true" dir="${bin}" classpath="${bin}" classname="java_cup
.Main">
                        <arg line="-grammar ..${file.separator}${src}${file.separator
}java_cup${file.separator}qsm.cup -symbols Sym -parser Grm -nonterms" />
                </java>

                <!-- move generated files to its standard location and overwrite the
existing ones -->
                <move todir="${src}${file.separator}Parse">
                        <fileset file="${bin}${file.separator}Sym.java" />
                        <fileset file="${bin}${file.separator}Grm.java" />
                </move>

                <!-- recompile parser source files (should compile only the two gener
ated files) -->
                <javac srcdir="${src}" classpath="${classpath_bin}" destdir="${bin}"
debug="on" fork="yes" />
        </target>
```

Dateiauswahl mit *Ant*

- Auswahl von Dateien über *Filesets*
- *Fielesets* enthalten *Patternsets* und/oder Selektoren
- *Patternset*-Element selektiert Dateien mittels Einschluss- oder Ausschlussbedingungen

```
<patternset>
   <include name = «suchmuster» />
   <exclude name = «suchmuster» />
</patternset>
```

Start eines *Ant*-Scripts in *Eclipse*



Statusmeldungen des *Ant*-Scripts

Console

<terminated> jamus1.11 build.xml [Ant Build] C:\Programme\Java\jre1.6.0_04\bin\javaw.exe (06.04.2011 06:00:53)

```
Buildfile: C:\Dokumente und Einstellungen\fritzsch\workspace\jamus1.11\scripts\build.xml
init:
build:
     [java] Dateiname: ..\src\java_cup\qsm.cup
     [java] Opening files...
     [java] Parsing specification from standard input...
     [java] ++ Parse is starting
     [java] ++ now in parse()
     [java] ++ now in user_init()
     [java] ++ now in init()
     [java] Parsing specification is over ...
     [java] Checking specification...
     [java] Warning: Terminal "CONST" was declared but never used
     [java] Warning: Terminal "GOTO" was declared but never used
     [java] Building parse tables...
     [java]   Computing non-terminal nullability...
     [java]   Computing first sets...
     [java]   Building state machine...
     [java] ++ Parse is over
     [java]   Filling in tables...
     [java]   Checking for non-reduced productions...
     [java] Writing parser...
     [java] Closing files...
     [java] ------- CUP v0.10j Parser Generation Summary -------
     [java]   0 errors and 2 warnings
     [java]   103 terminals, 155 non-terminals, and 353 productions declared,
     [java]   producing 599 unique parse states.
     [java]   2 terminals declared but not used.
     [java]   0 non-terminals declared but not used.
     [java]   0 productions never reduced.
     [java]   0 conflicts detected (0 expected).
     [java]   Code written to "Grm.java", and "Sym.java".
     [java] ---------------------------------------------------- (v0.10j)
     [move] Moving 2 files to C:\Dokumente und Einstellungen\fritzsch\workspace\jamus1.11\src\Parse
    [javac] Compiling 2 source files to C:\Dokumente und Einstellungen\fritzsch\workspace\jamus1.11\bin
BUILD SUCCESSFUL
Total time: 11 seconds
```

Kein javac, zuvor schon übersetzt !

Beurteilung von *Ant*

- sehr flexibel, alles unter Kontrolle
- plattformunabhängig (bzgl. OS)
- speziell für *Java* => einige hier zulässige Annahmen, die nicht mehr explizit angegeben werden müssen

Vorteile

- nutzt XML => schwer lesbar, langer Code
  - keine Programmiersprache
  - bedingte Ausführung nur mit Kunstgriffen möglich
- möglicher Ausweg: automatische Generierung von *Ant*-Skripten durch IDEs
  - aber: Migration zu anderer IDE erfordert manuelle Analyse
  - aber: automatisch generierte Skripte noch länger und umständlicher, da allgemein genug gehalten
- letztendlich „*angle bracket tax*“ zu zahlen

Nachteile

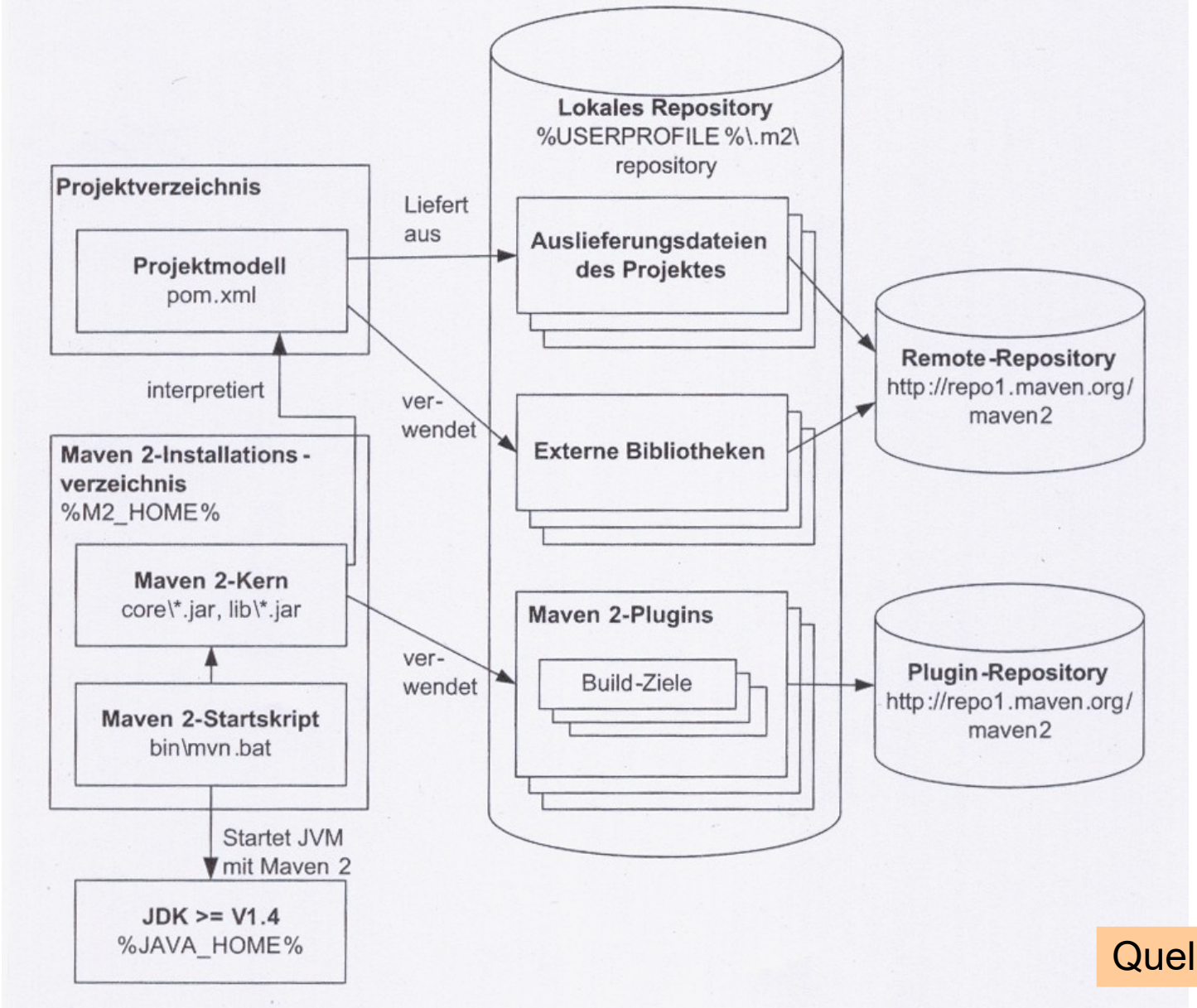
Quelle: [5]

Maven

- *Open-Source*-Projekt seit 2002
- aktuelle Version: *Maven 3.6.3* (vom 25. November 2019)
- besonders für *Java*-Projekte geeignet
  - benötigt JDK (ab 1.7), JRE nicht ausreichend
- modellbasierter, deklarativer Ansatz
  - Metadaten des Projekts werden in einem Modell beschrieben
  - Projektmodell (POM), Datei `pom.xml` im Projektverzeichnis
- Nachladen noch benötigter Bibliotheken nach Bedarf
  - *Maven*-Repositorys (keine Versionierung)
- legt automatisch den *Build*-Prozess fest, trifft Annahmen über dessen Ablauf, nur auf oberster Stufe anstoßen (hoher Automatisierungsgrad)
- unterstützt auch Tests und Dokumentation

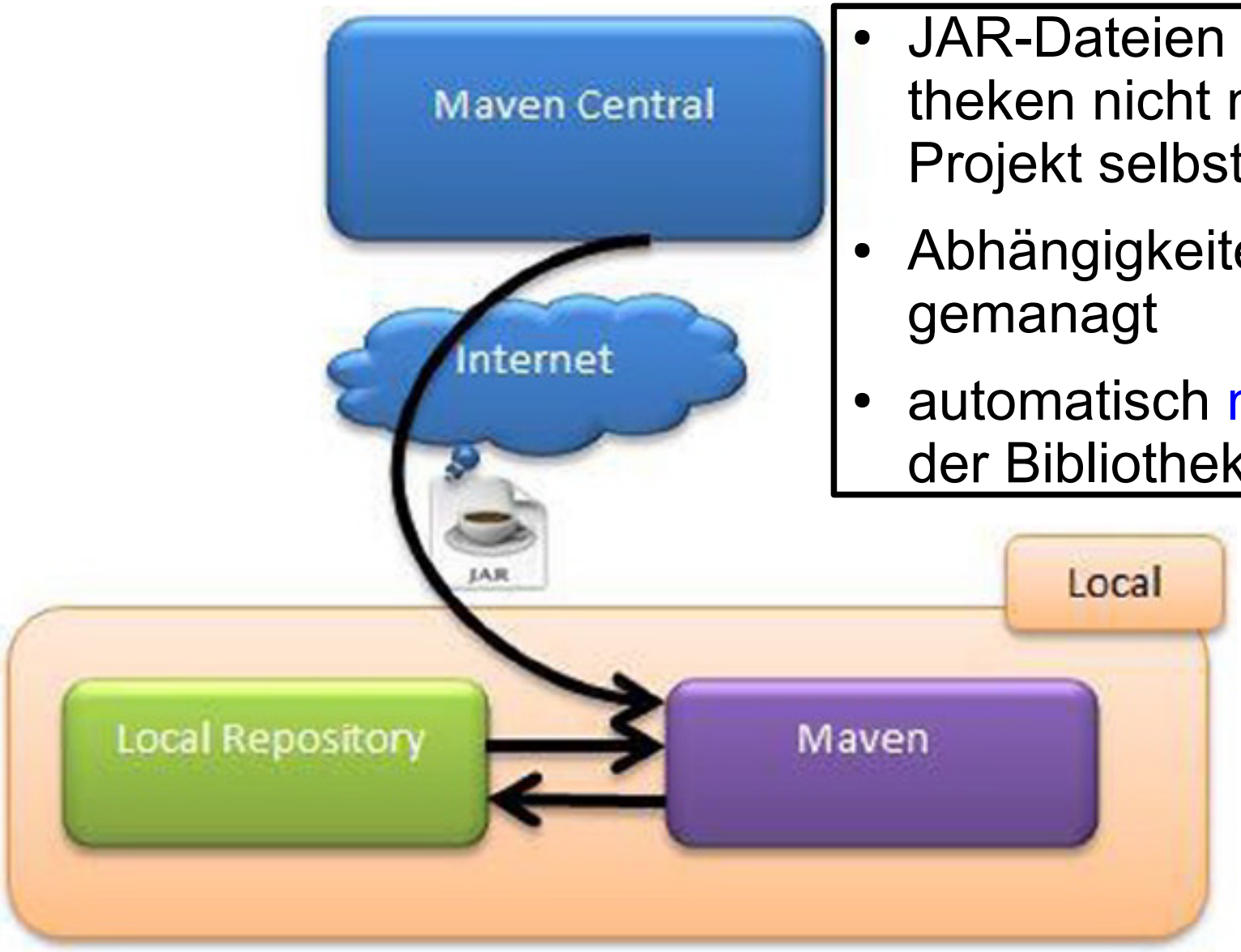
„Konvention vor Konfiguration"

Architektur von *Maven*



Quelle: G. Popp

Abhängigkeitsmanagement

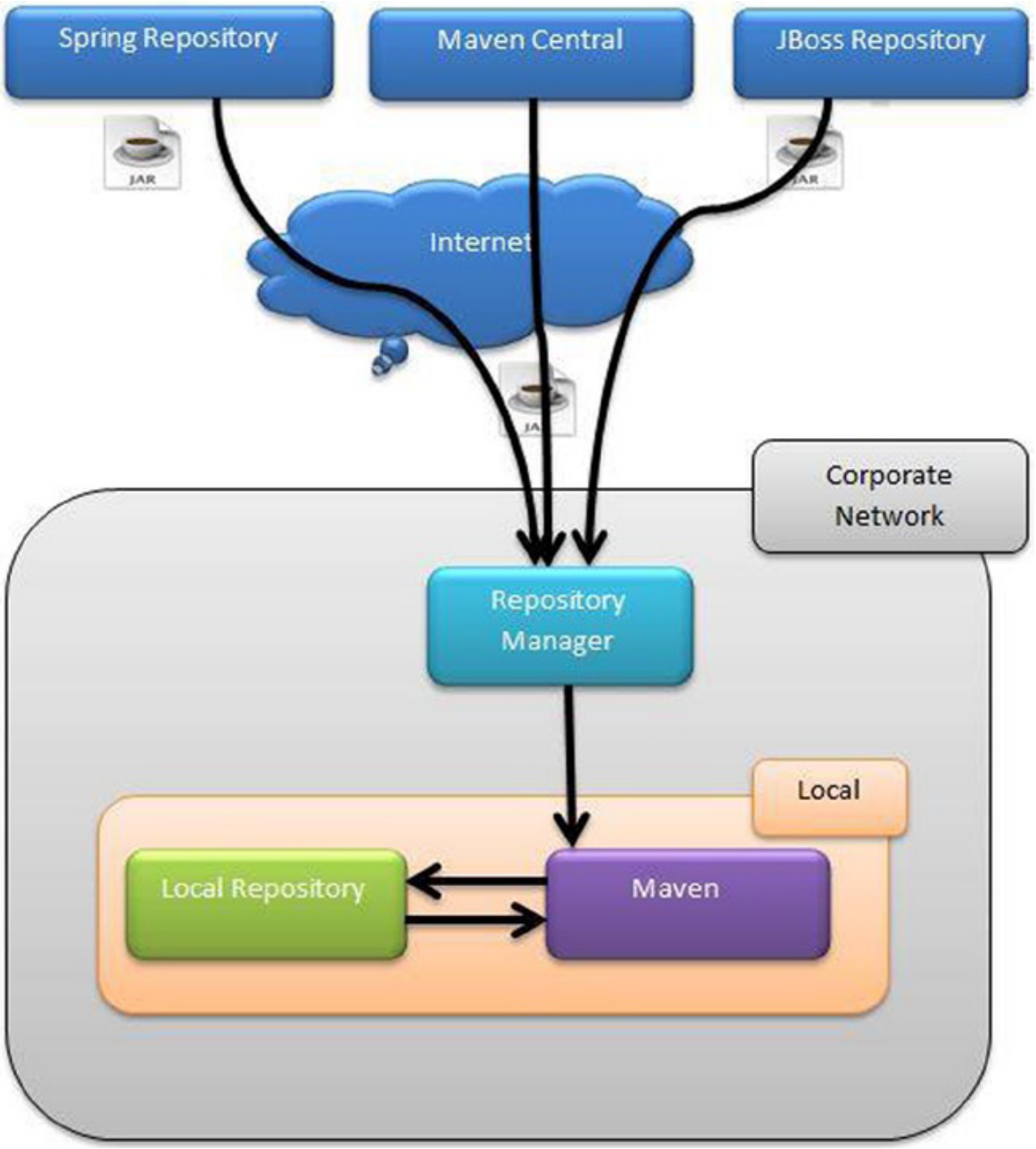


- JAR-Dateien der Bibliotheken nicht mehr in Projekt selbst nötig
- Abhängigkeiten transitiv gemanagt
- automatisch neue Version der Bibliothek unterstützt

Quelle: [2], S. 16

Repository-Manager

- Zwischenschaltung eines Proxys
- wichtig im Unternehmen
  - Sicherheit und geistiges Eigentum
  - Unternehmenspolitiken besser zu realisieren, z. B. nur offiziell freigegebene Software zu nutzen
  - geringere Abhängigkeit vom potenziellen Flaschenhals *Maven Central*



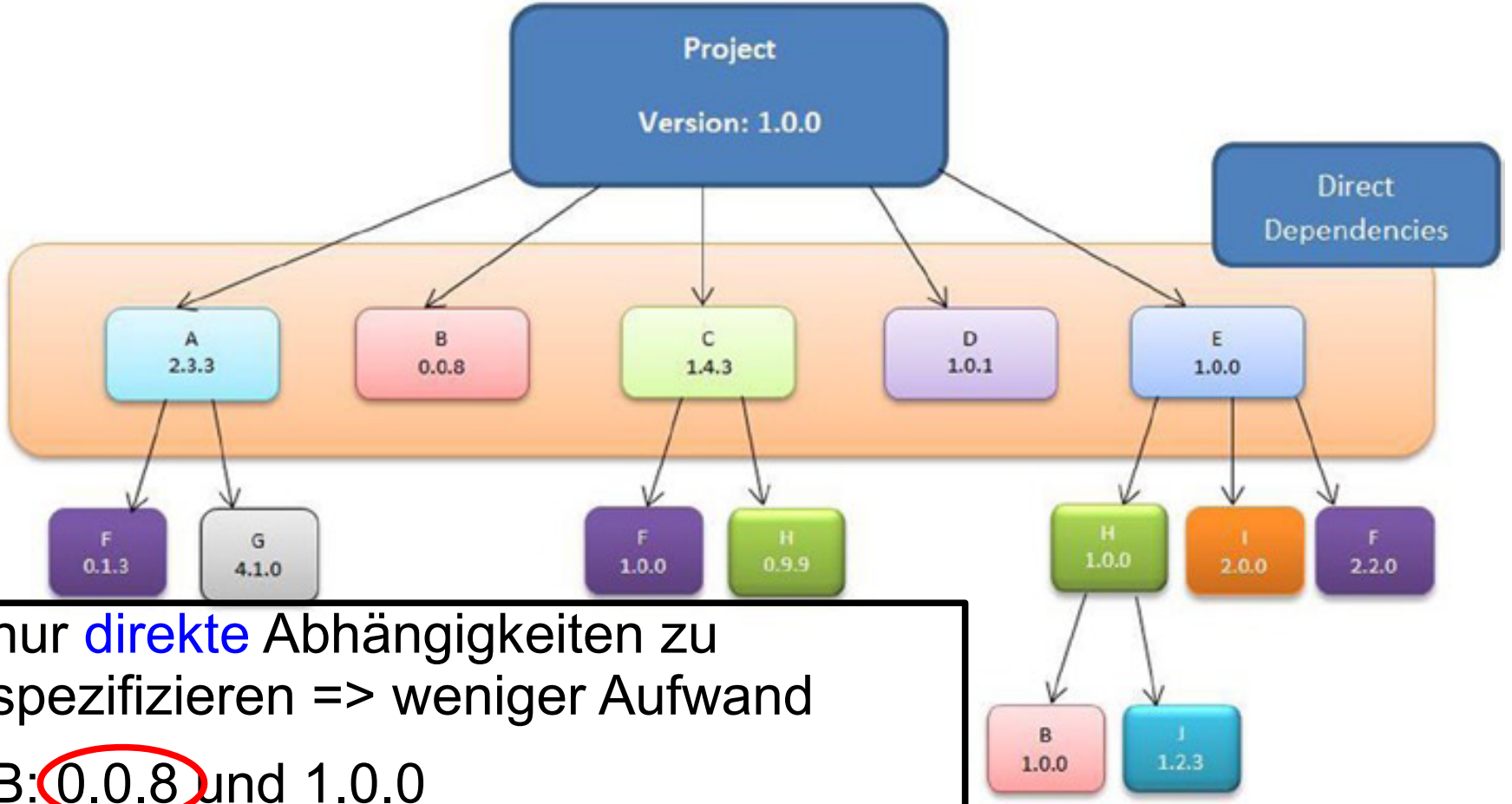
Quelle: [2], S. 17

GAV-Koordinaten bei Abhängigkeiten

- *Maven*-Abhängigkeiten typischerweise Archive
  - JAR, EAR, WAR oder ZIP
- eindeutige Beschreibung durch *Group*, *Artifact*, *Version*
- `groupId`: Name der verantwortlichen Organisation, z. B. `org.hibernate`, `log4j`, `org.springframework.boot`
- `artifactId`: eindeutiger Name des vom Projekt erzeugten Artefakts,
  z. B. `hibernate-tools`, `log4j`, `spring-core`
- `version`: Versions-Nummer des Projekts, z. B. `1.0.0`, `2.3.1-SNAPSHOT`, `4.3.6.Final`
- `type`: Art des Containers, z. B. JAR, WAR, EAR

Quelle: [2], S. 19

Transitive Abhängigkeiten und Versionsmediation



- nur direkte Abhängigkeiten zu spezifizieren => weniger Aufwand
- B: 0.0.8 und 1.0.0
  - am nächsten am Projektknoten
- F: 0.1.3, 1.0.0 und 2.2.0
  - gleiche Distanz, zuerst gefundene Abh.

Quelle: [2], S. 19f.

Maven: Installation für Kommandozeile

1. Sicherstellen, dass aktuelle JDK sauber installiert ist, z. B.

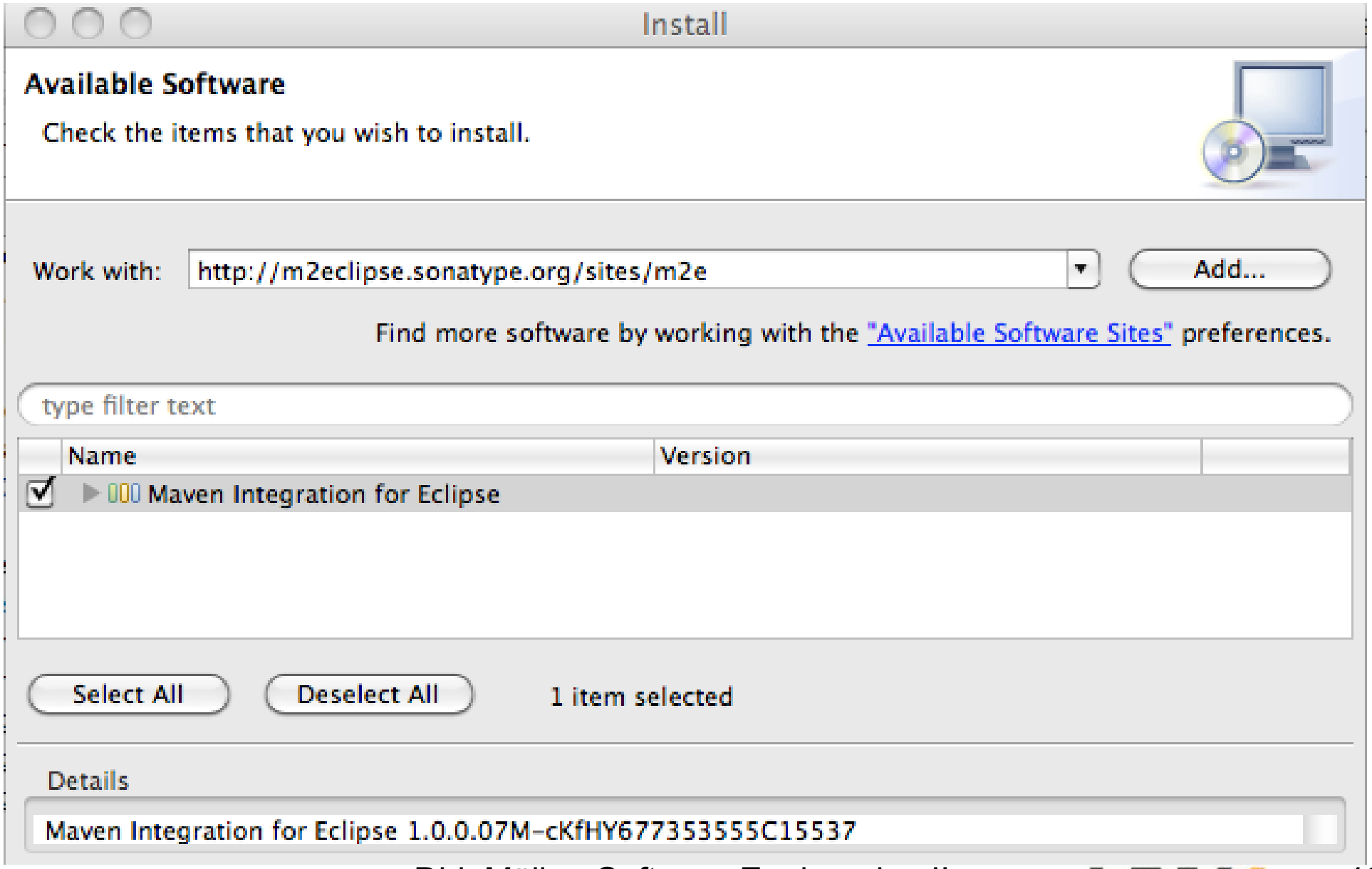
```
C:\Users\dmueller>java -version
java version "1.8.0_74"
Java(TM) SE Runtime Environment (build 1.8.0_74-b02)
Java HotSpot(TM) 64-Bit Server VM (build 25.74-b02, mixed mode)
C:\Users\dmueller>echo %JAVA_HOME%
c:\Program Files\Java\jdk1.8.0_74
```

2. Download des ZIP-Archivs der Version *Maven 3.6.3* (25. November 2019) von https://maven.apache.org/download.cgi
3. Umgebungsvariable (Systemvariable) `M2_HOME` auf Installationsverzeichnis setzen, z. B. auf `C:\apache-maven-3.6.3`
4. Umgebungsvariable (Systemvariable) `Path` um `%M2_HOME%\bin` am Anfang ergänzen

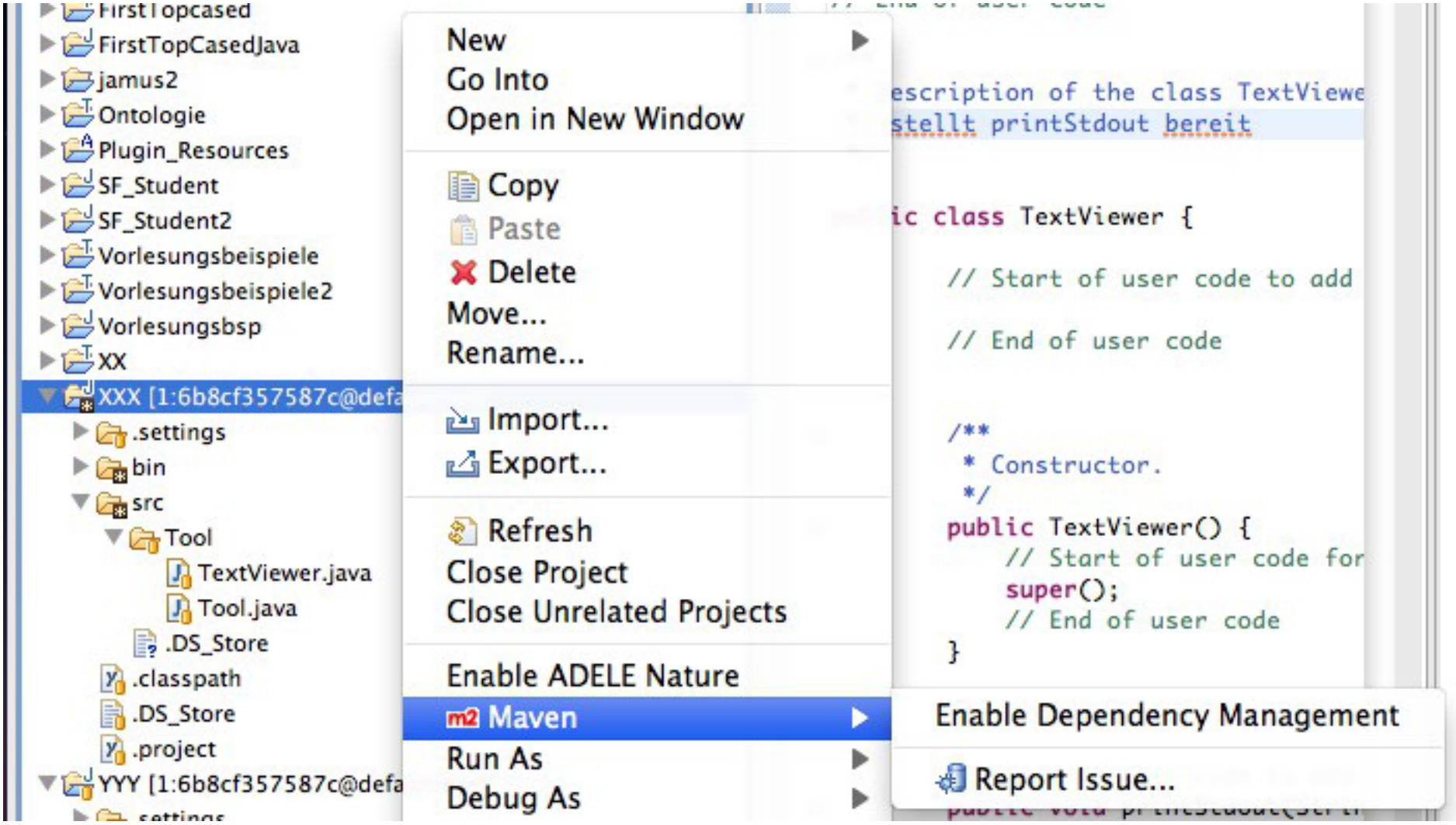
Maven: Erster Test der Installation

```
C:\Users\dmueller>mvn -v

Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)

Maven home: C:\apache-maven-3.6.3\bin\..

Java version: 10.0.1, vendor: Oracle Corporation, runtime: C:\Program
Files\Java\jdk-10.0.1

Default locale: de_DE, platform encoding: Cp1252

OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Einbindung in *Eclipse (1/2)*

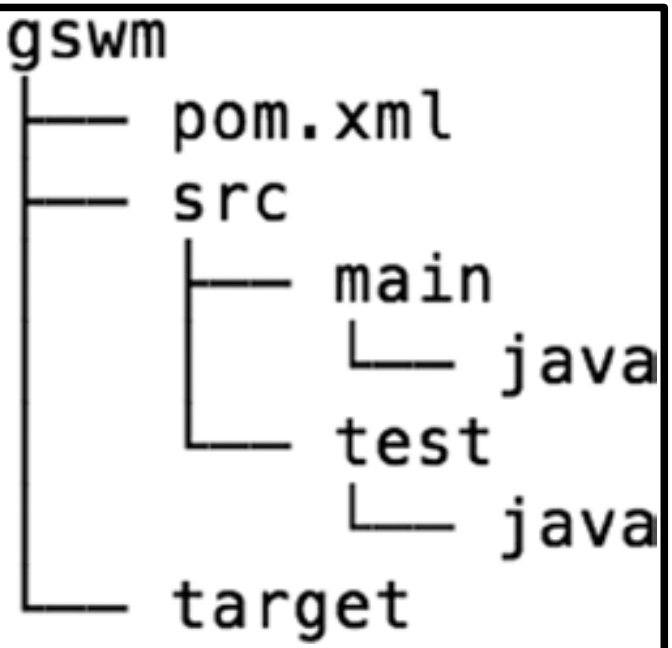


Einbindung in *Eclipse* (2/2)



„Hello World“ mit *Maven*

- `pom.xml` mit dem Projektmodell
- *Maven*-Build-Prozess besteht aus mehreren standardisierten Phasen
  - z. B. `compile`, `test`, `package`
  - *Build*-Ziele sind zugeordnet
  - meist nur ein Teil abgearbeitet
- einfacher Aufruf, z. B. `mvn compile`
  - *Maven* führt alle übergeordneten Phasen des Lebenszyklus aus
  - Laden des Plug-ins `maven-compiler-plugin` und Ausführen des zugeordneten *Build*-Zieles `compile`
  - Plug-ins werden im *Maven*-Repository verwaltet und von dort in den *Build*-Prozess eingebunden



Quelle: [2], S. 23ff.

pom.xml

```
<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">

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.6</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
</properties>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.gswmbook</groupId>
    <artifactId>gswm</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>Getting Started with Maven</name>
    <url>http://apress.com</url>
    <developers>
        <developer>
            <id>balaji</id>
            <name>Balaji Varanasi</name>
            <email>balaji@inflinx.com</email>
            <properties>
                <active>true</active>
            </properties>
        </developer>
    </developers>
</project>
```

passend zur *Maven*-Version, 4.0.0

Zeichensatzcodierung
=> plattformunabhängig

Version des *Maven*-Compilers

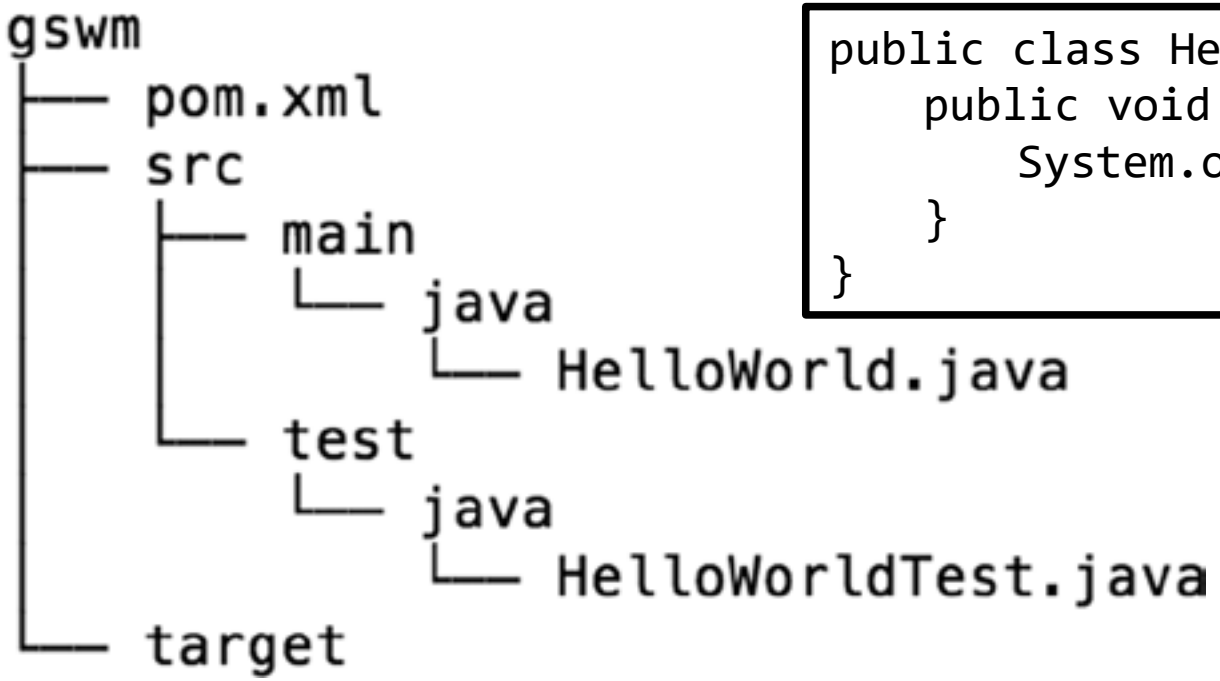
Domänenname des Herstellers
in umgekehrter Reihenfolge

Typ des Artefakts, Alternativen
wären `ear`, `war`

Name und URL, optional kurze
Beschreibung mittels `description`

Liste der beteiligten Entwickler

HelloWorld.java




```
public class HelloWorld {
    public void sayHello() {
        System.out.print("Hello World");
    }
}
```

- minimale *Hello-World*-Klasse ohne Einstiegspunkt
- aber `mvn package` bereits möglich
  - Kompilieren und Erstellen der Auslieferungsdatei
- aber vielleicht fehlerhafte Implementierung...

HelloWorldTest.java

```
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
public class HelloWorldTest {
    private final ByteArrayOutputStream
        outStream = new ByteArrayOutputStream();
    @Before
    public void setUp() {
        System.setOut(new PrintStream(outStream));
    }
    @Test
    public void testSayHello() {
        HelloWorld hw = new HelloWorld();
        hw.sayHello();
        Assert.assertEquals("Hello World", outStream.toString());
    }
    @After
    public void cleanUp() {
        System.setOut(null);
    }
}
```

```
<dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
</dependencies>
```

Ergänzung am Ende der pom.xml

Wirklich der gewünschte Text ausgegeben?

Test von „Hello World“

- erfolgreich
- ohne Ärger mit den Details von *JUnit*
- Tippfehler, z. B. „HelloWorld“ statt „Hello World“ als nächstes Experiment
- deutet Mächtigkeit von *Maven* an

```
C:\Windows\System32\cmd.exe
D:\Eigene Dateien\HTW_Dresden\SWEngII\HelloWorldMaven>mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< com.apress.gswmbook:gswm >----------------------
[INFO] Building Getting Started with Maven 1.0.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ gswm ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory D:\Eigene Dateien\HTW_Dresden\SWEngII
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ gswm ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\Eigene Dateien\HTW_Dresden\SWEngII\HelloWor
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ gs
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory D:\Eigene Dateien\HTW_Dresden\SWEngII
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ gswm --
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ gswm ---
[INFO] Surefire report directory: D:\Eigene Dateien\HTW_Dresden\SWEngII\HelloWor

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running HelloWorldTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ gswm ---
[INFO] Building jar: D:\Eigene Dateien\HTW_Dresden\SWEngII\HelloWorldMaven\targe
[INFO] ------------------------------------------------------------------------
```

Maven: Ergänzungen und Bewertung

- `mvn clean`
  - Löschen aller generierten Artefakte
- `mvn javadoc:javadoc`
  - Generierung der *JavaDoc*-Dokumentation
- `mvn test`
  - Modultests durchführen
- mächtiger Ansatz der *Build*-Verwaltung für die *Java*-Welt
  - Konvention vor Konfiguration
  - ganzer Lebenszyklus incl. Test und Dokumentation abgedeckt
  - Abhängigkeitsverwaltung automatisch, aber umstritten [7]
  - Anpassung für Spezialfälle z.T. schwierig, Kombination mit *Ant*
  - XML-Problem (Lesbarkeit, Umfang) von *Ant* geerbt
    - aber: hierarchische POM-Dateien (Wiederverwendung) möglich

Gradle

- seit 2012
- aktuelle Version 6.0 (8. November 2019)
- vereint gute Ansätze von *Maven* und *Ant*
  - Motto: „Konvention ist gut und ebenso Flexibilität.“
- nutzt eine eigene, ausdrucksstarke domänenspezifische Sprache (DSL)
- anfangs *Ivy* genutzt, später eigener Manager für Abhängigkeitsverwaltung
- von *Google* für *Android OS* genutzt (seit 2013)

Gradle-Beispiel: `build.gradle`

```
apply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'findbugs'
apply plugin: 'pmd'

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    testCompile group: 'org.hamcrest', name: 'hamcrest-all', version: '1.3'
}
```

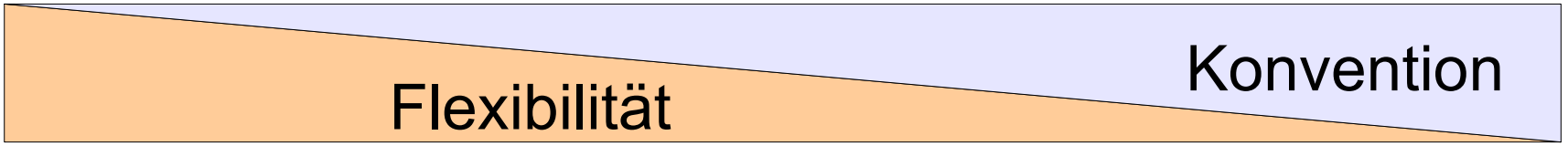
`gradle tasks --all` liefert Liste aller Tasks, die ausgeführt werden können

Zusammenfassung Versionsmanagement

- *Subversion* seit >15 Jahren etabliert
- Verteilter Ansatz hat den zentralen überholt.
  - *Git* und *Mercurial* inzwischen standardmäßig bei den meisten neuen Projekten eingesetzt
  - Vorteile
    (einfaches *Merge*, Rechteverwaltung, Offline-Arbeit, robust)
    überwiegen die
    Nachteile
    (zu viele Zweige möglich, Netzwerk anfangs stärker beansprucht, zu viele *Repositories* möglich)
- Trend zum *Crossover* aus verteiltem und zentralem Ansatz
  - offizielles *Repository* („*golden master*“) vom zentralen Ansatz übernommen
  - andere Vorteile vom verteilten Ansatz übernommen

Zusammenfassung *Build*-Werkzeuge

- Unterscheidung in *C/C++*- und *Java*-Welt sinnvoll
  - *Make* (und seine Derivate) als der Standard in der C/C++-Welt
- *Java*-Welt bietet viele Möglichkeiten



Ant(+Ivy)	*Gradle*	*Maven*
-XML	+DSL +Lebenszyklus	-XML +Lebenszyklus
Konfiguration	Konvention ist gut und ebenso Flexibilität.	Konvention vor Konfiguration

Welches Werkzeug wofür?

	Versionsverwaltung	*Build*-Verwaltung
kleines *Java*-Projekt	*Subversion*	*Ant*
großes *Java*-Projekt	*Mercurial*, *Git*	*Maven, Gradle*
kleines *C/C++*-Projekt	*Subversion*	*GNU Make*
großes *C/C++*-Projekt	*Mercurial, Git*	*GNU Make*

Literatur

[1] Gunther Popp: „Konfigurationsmanagement mit Subversion, Maven und Redmine“, dpunkt-Verlag, 4. Auflage, 2013

[2] Balaji Varanasi, Sudha Belida: „Introducing Maven“, *E-Book* bei Apress, 2014

[3] „GNU make“, Documentation 0.73, 30.9.2014, Download am 8.3.2016, http://www.gnu.org/software/make/manual/make.html

[4] Java Build Tools: „Ant vs Maven vs Gradle | Technology Conversations“, 18.6.2014, Download am 9.3.2016, http://technologyconversations.com/2014/06/18/build-tools/

[5] Jess Johnson: „Java Build Systems: A Sad State of Affairs“, 4.10.2010, Download am 9.3.2016, http://grokcode.com/538/java-build-systems-a-sad-state-of-affairs/

[6] Alex Brewitt: „Facebook makes Mercurial faster than Git“, 9.1.2014, Download am 10.03.2016, http://www.infoq.com/news/2014/01/facebook-scaling-hg

[7] Kent R. Spillner: „Java Build Tools: Ant vs. Maven“, 14.11.2009, Download am 10.03.2016, http://kent.spillner.org/blog/work/2009/11/14/java-build-tools.html

[8] Ben Thompson: „Git didn’t beat SVN, GitHub did.“, 27.05.2016, Download am 03.04.2018, https://blog.gitprime.com/git-didnt-beat-svn-github-did/