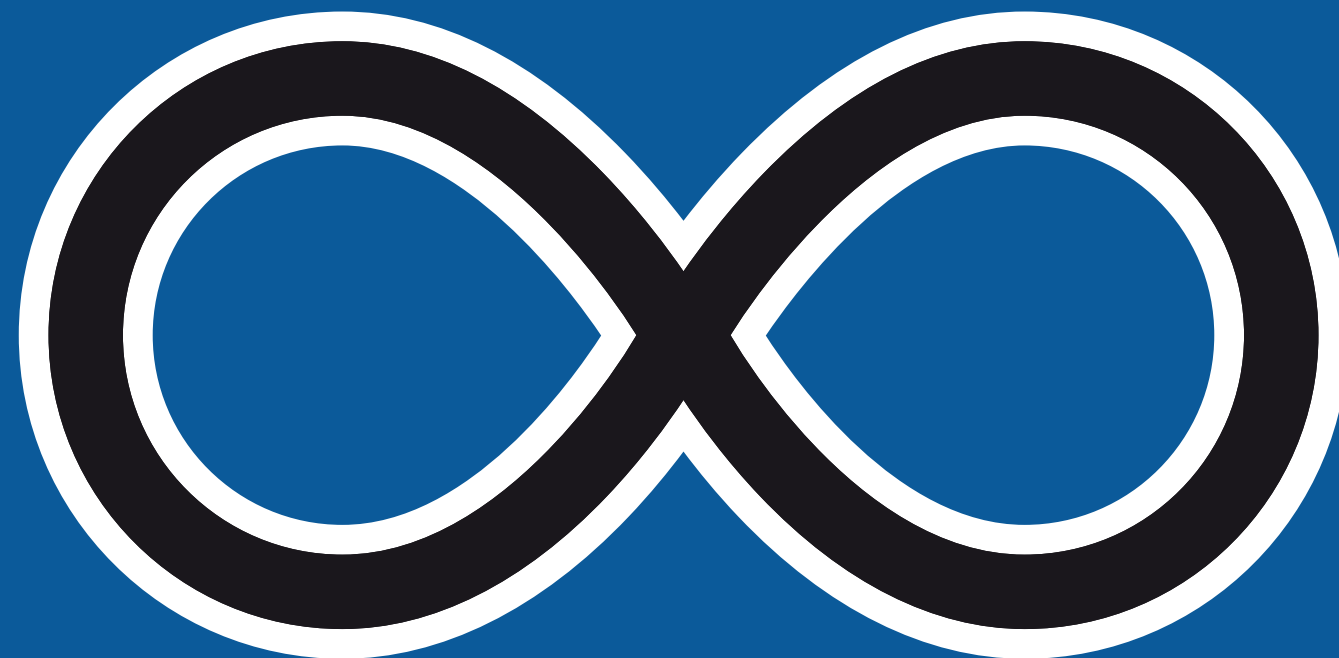


SOFTWAREENTWICKLUNG

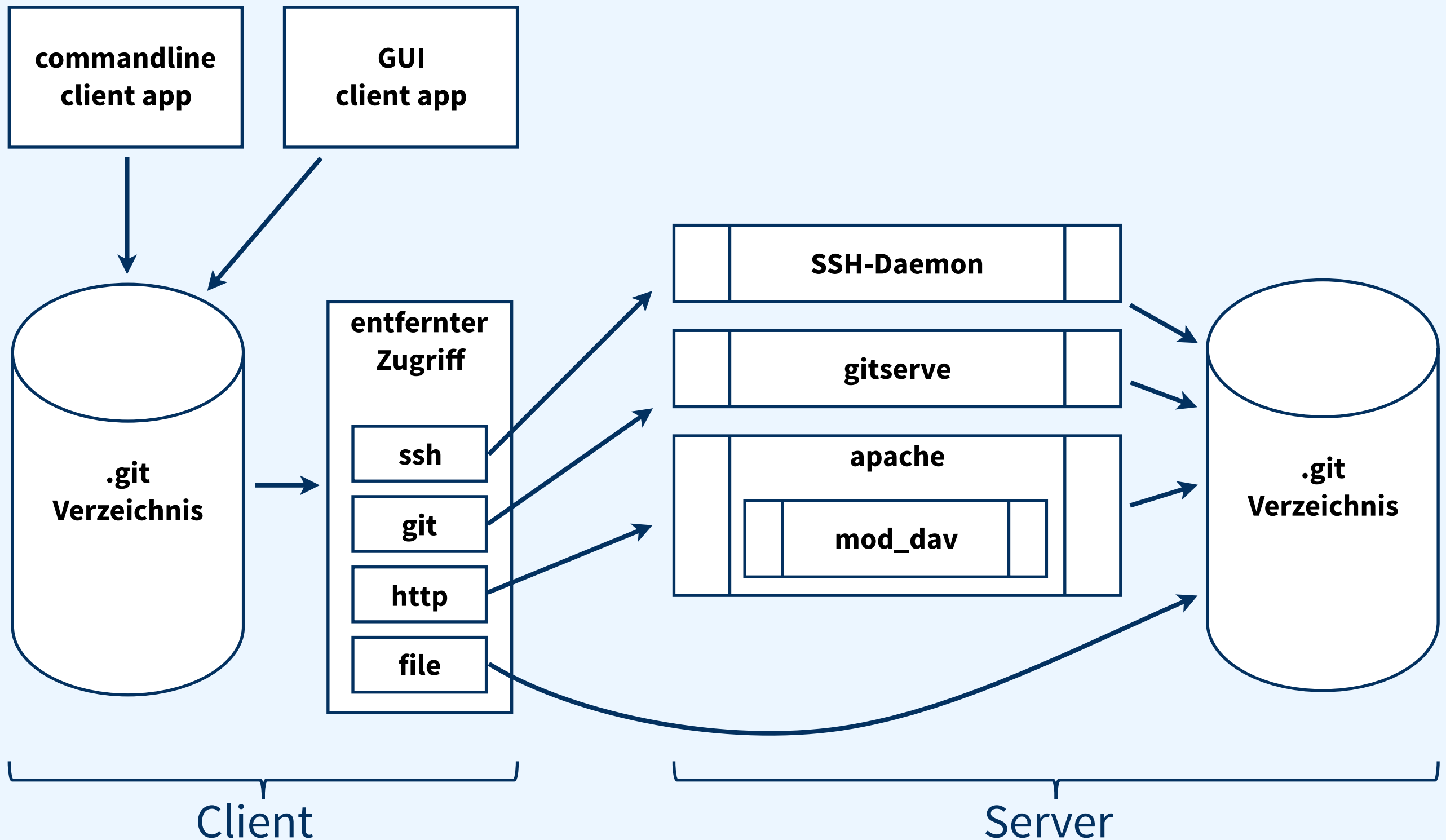
IM TEAM MIT OPEN-SOURCE-WERKZEUGEN

05 - Build Management



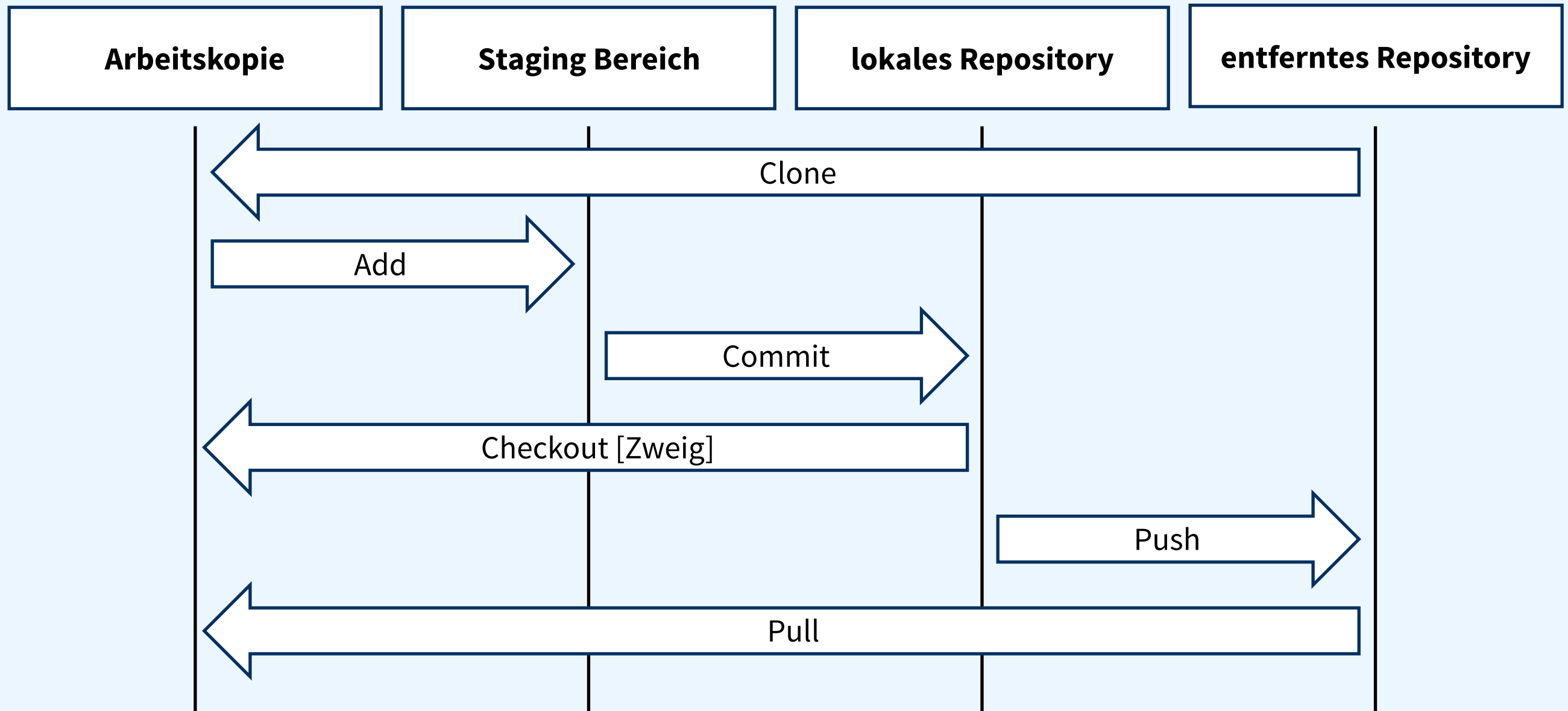
WIEDERHOLUNG

Architektur

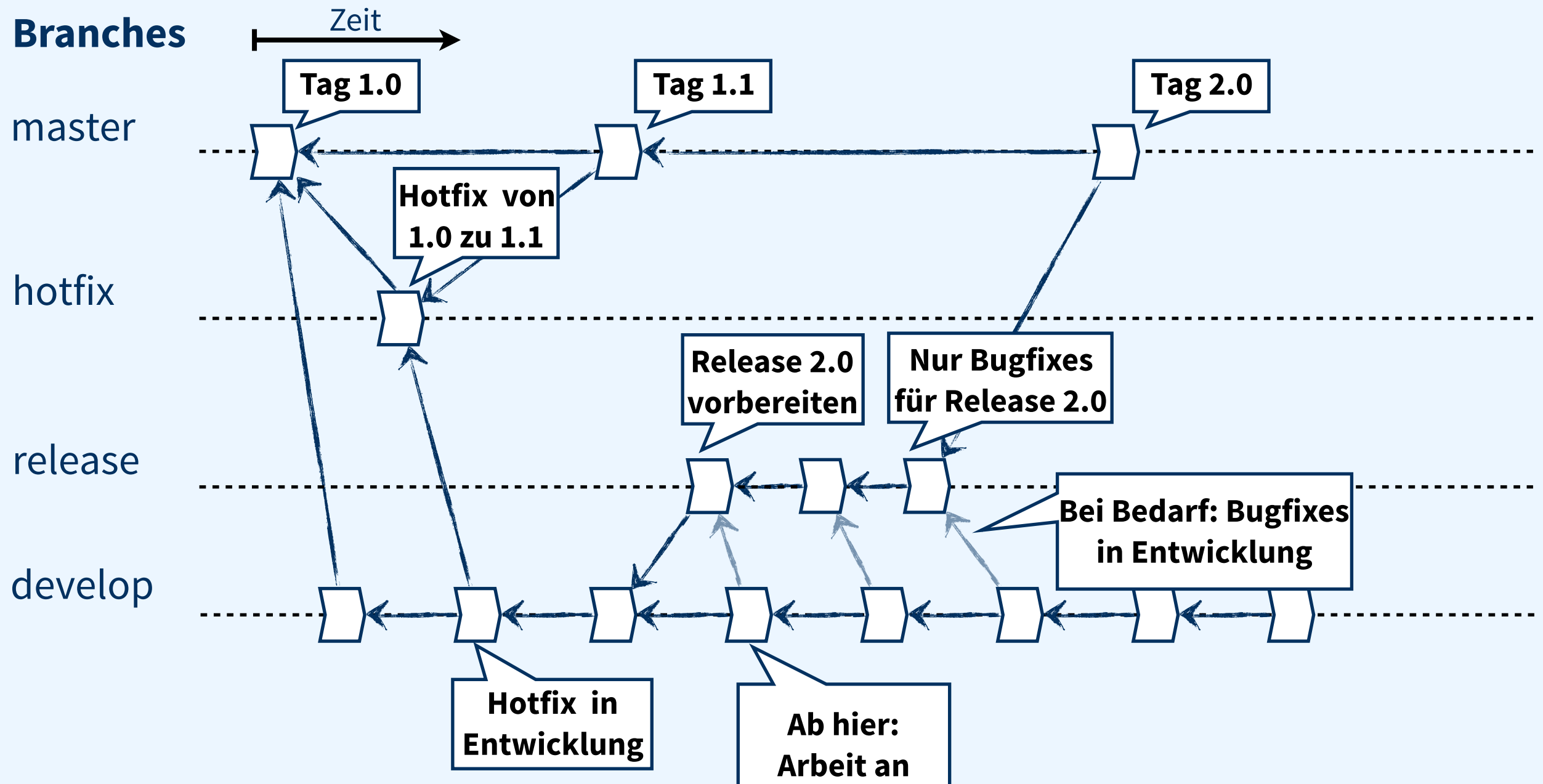


Kommunikation

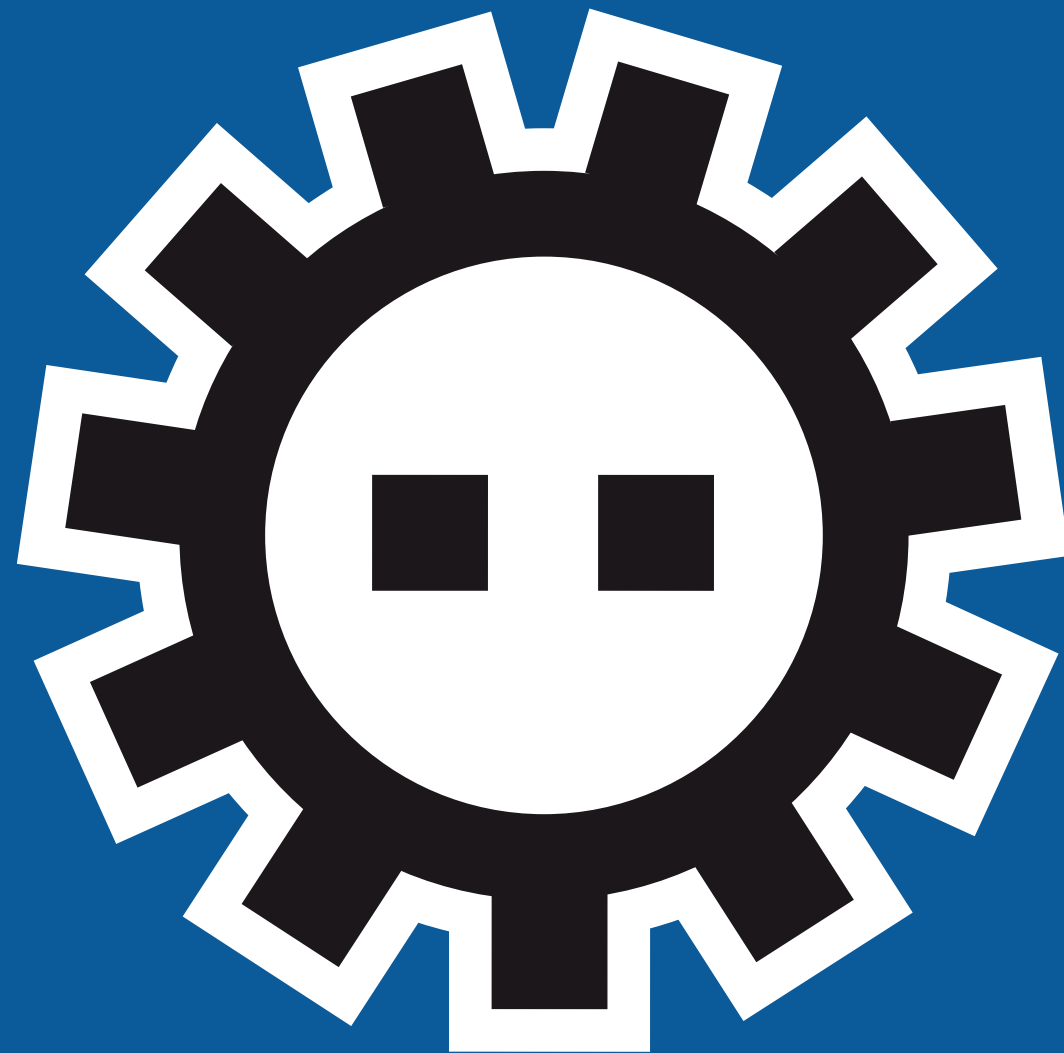
Übersicht über die Auswirkung von Git-Operationen:



Release Branches



Release-Branches erlauben eine neue Veröffentlichung zu stabilisieren (Alpha-Version, Beta-Version, Release Candidates, ...)



MOTIVATION

Konversation (Beispiel)

- Kunde:** Konfigurationsmanagement ist ein großes Wort. Wir haben doch eine Versionsverwaltung, in der alle Quelltexte abgelegt sind!
- Berater:** Was ist mit Fremdbibliotheken?
- Kunde:** Die legen wir auch in der Versionsverwaltung ab!
- Berater:** Was ist mit verschiedenen Versionen der Bibliotheken?
- Kunde:** In jedem Projekt sind nur die Bibliotheksversionen abgelegt, die es konkret benötigt.
- Berater:** Sie haben also (a) Build-Ergebnisse in der Versionsverwaltung und (b) diese auch noch redundant. Was machen Sie, wenn die Fremdbibliothek aufgrund eines Sicherheitsproblems aktualisiert werden muss?
- Kunde:** Dann aktualisieren wir eben diese Bibliothek überall. Aber wer braucht überhaupt ein Build von einem Release, das letztes Jahr herausgegeben wurde?
- Berater:** Naja, wenn ein Fehler im Release erkannt wird, muss man diesen auf einem Entwicklersystem auch mal nachstellen können.

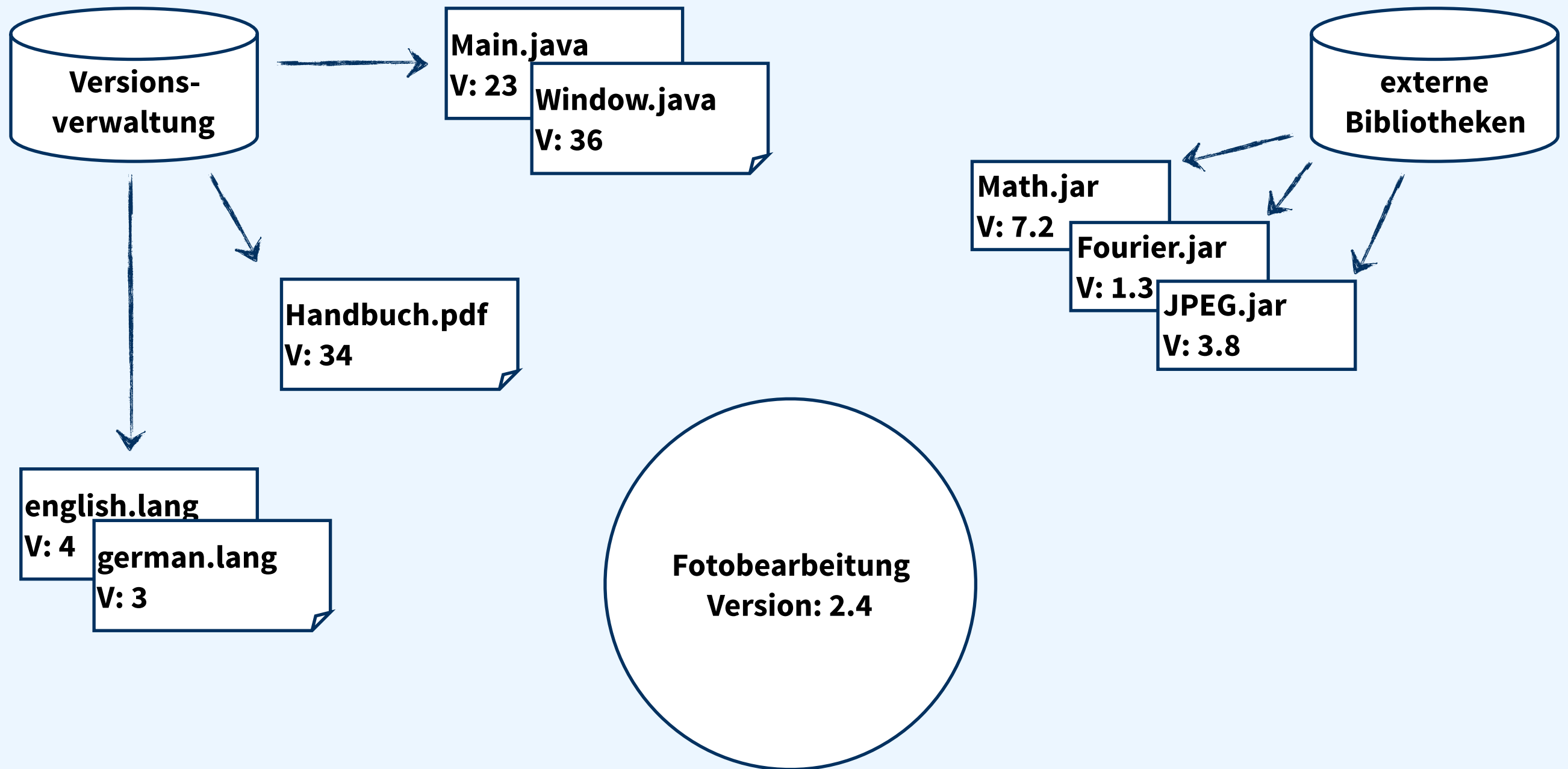
Konversation (Beispiel forts.)

- Berater:** Was ist mit abhängigen Bibliotheken? Wenn Sie die Bibliothek eines Frameworks aktualisieren, müssen Sie vielleicht zusätzliche Bibliotheken hinzunehmen. Im schlimmsten Fall merken Sie das erst zur Laufzeit!
- Kunde:** Ja, das ist ein Problem.
- Berater:** Und was ist mit verschiedenen Konfigurationen der Bibliotheken auf verschiedenen Systemen (Produktiv/Test/Entwickler)?
- Kunde:** Wir haben das mit Checklisten gelöst. Zu jedem Release gibt es genaue Anweisungen, wie das Release reproduziert werden kann, welche Fremdbibliotheken in welcher Version benötigt werden, etc.
- Berater:** Und das funktioniert?
- Kunde:** Meistens...

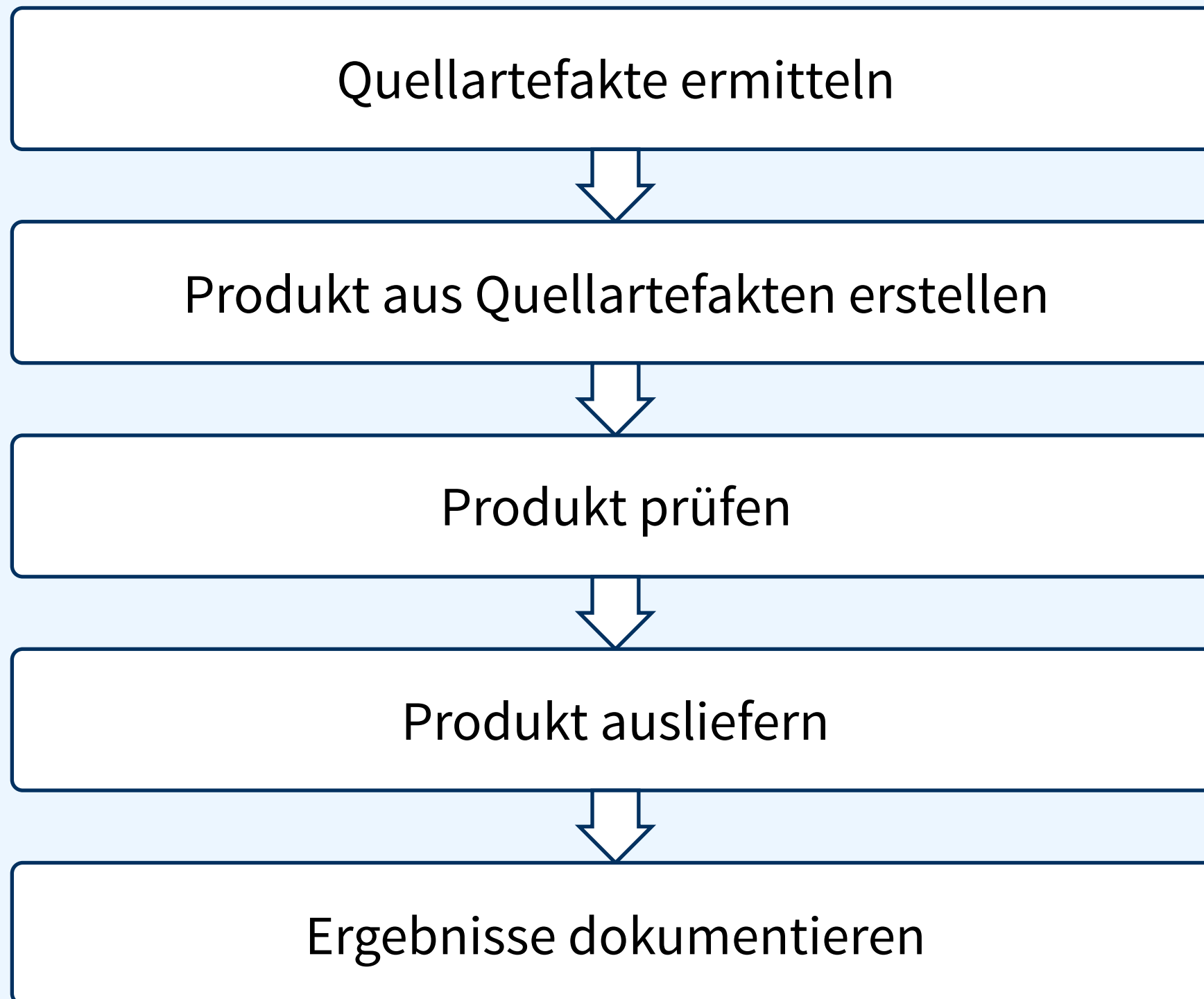
Bauen von Anwendungen

- Wie wird ein Software-Produkt gebaut?
- Zielartefakt:
 - ▶ z.B. Fotoverwaltung in der Version 2.4
- Quellartefakte:
 - ▶ Quellcode (z.B. Java Dateien)
 - ▶ Komponenten von Drittanbietern (z.B. JPEG-Bibliothek)
 - ▶ Ressourcen (z.B. eigene Beispielbilder)
 - ▶ Dokumentation (z.B. Benutzerhandbuch)
 - ▶ Lokalisierung (z.B. englische und deutsche Programmtexte)
- Alle Quellartefakte sind versioniert

Bauen von Anwendungen



Build Prozess

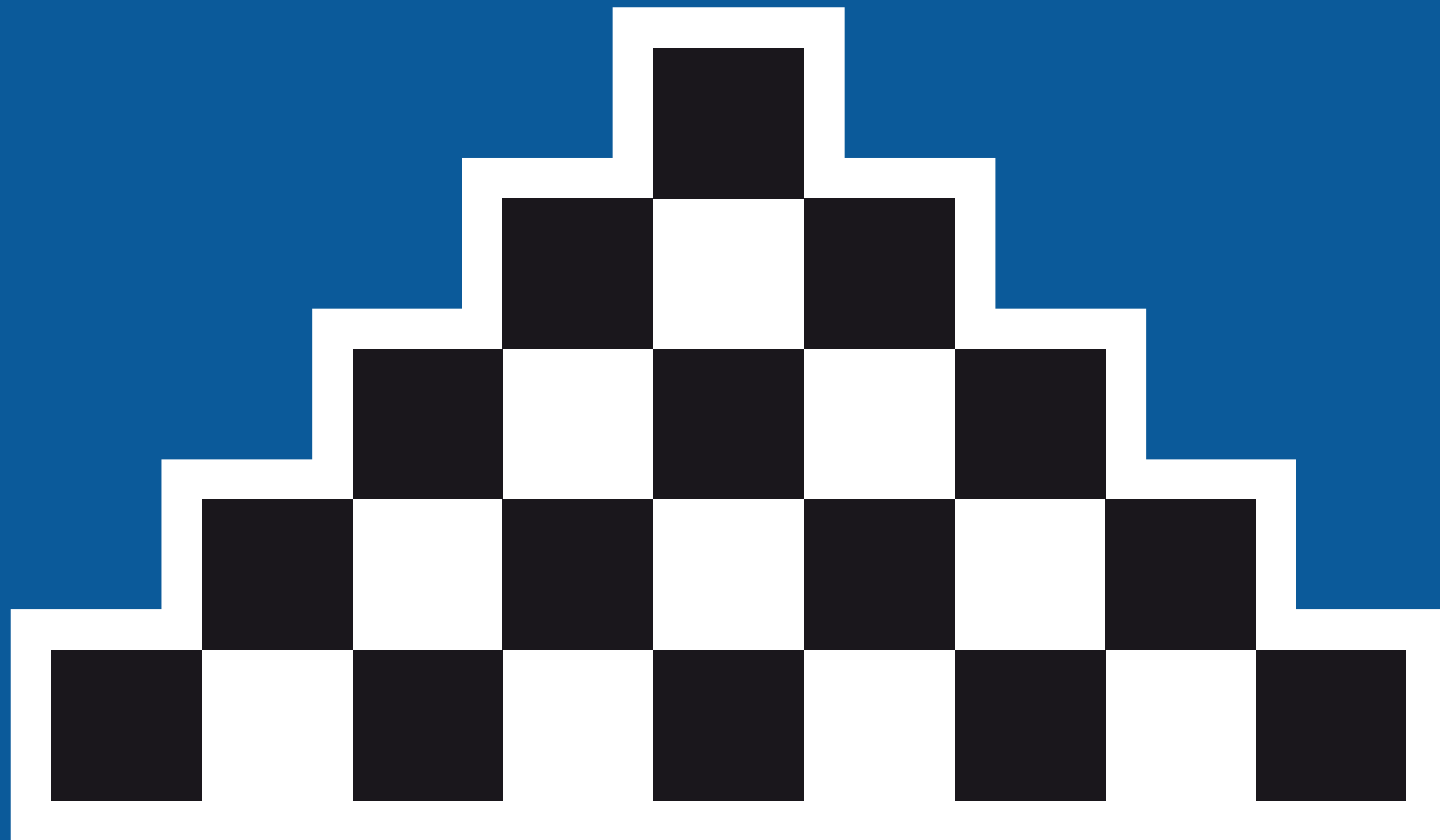


Automation

- Erstellung des Produkts Fotoverwaltung → wiederholte Ausführung von Aufgaben
 - ▶ z.B.: kompilieren und Programmpaket erstellen
 - Manuelle Ausführung hat viele Nachteile:
 - ▶ unangenehme Aufgabe
 - ▶ fehlerträchtig
 - ▶ nur eingeschränkt reproduzierbar
 - ▶ schwierig bei Team-Arbeit
- ➔ Software-Qualität leidet!

Automation: Ziele

- Der Erstellungsprozess sollte unabhängig von der IDE sein
- Der Erstellungsprozess sollte in die IDE integriert sein
- Dokumentationserstellung sollte Teil der Automation sein:
 - ▶ Projektdokumentation
 - ▶ Dokumentation der Compiler-Fehler und -Warnungen
 - ▶ Test-Reports
- Der Erstellungsprozess sollte reproduzierbar auf allen Entwicklungsmaschinen gleich sein



GRUNDLAGEN

Automation: Aufgaben

Aufgabe	Beschreibung
Validation	Überprüfung, ob Quelldokumente korrekt sind, z.B. für XML-Dokumente
Source Generation	Erzeugung von Quelldateien aus Metadaten, z.B. aus Annotationen
Compilation	Kompilierung der Java-Quelldateien
Test Execution	Ausführung der Unit-Tests
Report Generation	Erzeugung von Reports, z.B. die Ergebnisse der Test oder die Testabdeckung
Documentation Generation	Erzeugung von Dokumentation, z.B. von JavaDocs oder der Benutzerdokumentation
Packaging	Erstellung eines Programmpakets, z.B. eines WAR für eine Web-Anwendung



WERKZEUGE

Automation: Werkzeuge

- Automation existiert für viele Programmiersprachen und Plattformen
- Umfang und Konzept variieren aber erheblich
 - ▶ Unix make (für C/C++ Projekte)
 - ▶ Apache Ant für Java
 - ▶ NAnt für .net
 - ▶ Apache Maven
 - ▶ Rake für Ruby
 - ▶ Gradle für Java
- Shellskripte und Batchdateien sind auch verbreitet
 - ▶ sind aber meist nicht plattformunabhängig

Unix make

- Mit *make* werden Abhängigkeiten zwischen Dateien beschrieben:
 - ▶ z.B.: Objektdatei ist von der Quelldatei abhängig
 - » wenn Quelldatei sich ändert, muss die Objektdatei neu erstellt werden
- Wenn Quelldatei jünger als Zieldatei wird eine Operation ausgeführt
- Wenn Abhängigkeitsketten entstehen, führt *make* diese korrekt aus

Makefile

```
calculator: multiplication.o division.o
    ld -o calculator multiplication.o division.o

multiplication.o: multiplication.c
    gcc -c multiplication.c

division.o: division.c
    gcc -c division.c

clean:
    rm *.o calculator
```

Unix make

- Hat sich außerhalb der Entwicklung mit C nicht durchgesetzt

Nachteile:

- Ist nicht plattformunabhängig
 - ▶ *make*-Programm ist zwar für alle Plattformen erhältlich
 - ▶ die Operationen (z.B. »rm *.o«) sind aber plattformabhängig
- Ist dateibasiert
 - ▶ was ist wenn Quelle oder Ziel keine Dateien sind?
 - » z.B. Netzwerkressourcen, Komponentenarchiv, Status einer Aktion, Bibliothek in einer konkreten Version

Apache Ant

- Idee: Plattformunabhängige Variante von *make*
- Konzepte:
 - ▶ XML als plattformunabhängige Beschreibung des Makefile
 - ▶ für alle plattformspezifischen Operationen stehen Abstraktionen bereit
 - » z.B. `<delete>` statt `rm`, `<copyfile>` statt `cp`, etc...

build.xml

```
<project name="Calculator" default="compile" basedir=". ">
  <target name="compile" description="compile the source " >
    <cc outtype="executable" outfile="calculator" objdir=". ">
      <fileset dir="." includes="*.c"/>
    </cc>
  </target>
  <target name="clean" description="clean up" >
    <delete file=".o"/>
  </target>
</project>
```

Apache Ant

- *Ant* (und auch *make*) sind prozedural geprägt
 - ▶ wenn eine Bedingung erfüllt ist, werden sequentiell Operationen ausgeführt

Nachteile:

- Etablierte Abläufe müssen für jedes Projekt erneut angegeben werden
 - ▶ z.B. erst kompilieren, dann Bibliothek erstellen
- Die Operationen befinden sich auf einer sehr niedrigen Ebene
 - ▶ z.B. lösche Datei, erstelle Verzeichnis, kompiliere Datei
- Auch *Ant* arbeitet nur auf Dateibasis
 - ▶ wie finde und verwende ich z.B. die JPEG-Bibliothek in der Version 3.8?

Was ist Maven?

- Deklaratives Build Management System
 - ▶ *Inhalt* des Projekts wird beschrieben, nicht *Struktur* oder *Abläufe*
- Durch die Verwendung einer definierten Verzeichnisstruktur wird kaum Konfigurationsaufwand benötigt
 - ▶ *convention over configuration*
- Maven bietet vordefinierte Arbeitsabläufe
 - ▶ siehe Tabelle *Automation: Aufgaben*
- Viele Plugins verfügbar, die die Arbeitsabläufe erweitern
 - ▶ *embedded web container*
- Maven verwaltet auch die Abhängigkeiten von Projekten
 - ▶ *transitive dependency management*

Maven installieren

- Maven ist ein Java-Programm und benötigt ein JRE
- Maven ist ein Kommandozeilenprogramm
- Kann von folgender Adresse heruntergeladen werden:
 - <http://maven.apache.org/download.cgi>
 - siehe auch Installationsanweisung auf der Seite
- Maven funktioniert nur innerhalb von Projektverzeichnissen
 - benötigt eine Konfigurationsdatei »*pom.xml*«
 - benötigt eine standardisierte Verzeichnisstruktur

Maven Lifecycle

- Zentrales Konzept von Maven:
 - ▶ die Ausführung von vordefinierten Arbeitsabläufen (Lifecycle)
- Jeder Lifecycle besteht aus sequentiellen Phasen
- In die Phasen können sich Plugins einhängen, die Operationen durchführen
 - ▶ Standardplugins sind vordefiniert und müssen höchstens konfiguriert werden
 - » z.B. Java-Dateien kompilieren, Jar-Bibliothek erstellen
 - ▶ weitere Plugins können bei Bedarf in den Lifecycle eingehängt werden
- Nicht immer muss der komplette Lifecycle durchlaufen werden
 - ▶ z.B.: `mvn compile` → alle Phasen bis *compile* werden durchlaufen

Default Lifecycle (Auswahl)

Default Lifecycle	
Phase	Beschreibung
validate	überprüft, ob die Projektkonfiguration korrekt ist
generate-sources	Erzeugung von Quelldateien aus Metadaten, z.B. aus Annotationen
process-resources	Kopiert und ersetzt Variablen in Konfigurationsdateien
compile	Kompilierung der Quelldateien
test-compile	Kompilierung der Test-Quelldateien
test	Ausführung der Unit-Tests
package	Erstellung eines Programmpakets, z.B. eines WAR für eine Web-Anwendung
install	legt das erzeugte Paket im lokalen Maven-Repository ab
deploy	legt das erzeugte Paket im entfernten Maven-Repository ab

Weitere Lifecycles

Clean Lifecycle

Phase	Beschreibung
pre-clean	bereitet das Aufräumen vor
clean	löscht temporäre und erzeugte Dateien
post-clean	schließt das Aufräumen ab

Site Lifecycle

Phase	Beschreibung
pre-site	bereitet das Erzeugen der Projektdokumentation vor
site	Erzeugt die Projektdokumentation in Form von HTML-Dokumenten
post-site	schließt das Erzeugen der Projektdokumentation ab
site-deploy	kopiert die erzeugte Projektdokumentation auf einen entfernten Server

Lifecycle Beispiel

- Für ein erstes Beispiel soll ein Testprojekt verwendet werden
 - ▶ kann mit `mvn archetype:generate` werden
 - ▶ enthält eine Java-Datei und eine Test-Datei
- »`mvn test`« führt alle Phasen bis zur Testphase aus

```
$ mvn archetype:generate -B -DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.3 -DgroupId=de.betermieux -DartifactId=beispiel

...
[INFO] project created from Old (1.x) Archetype in dir: /Users/stefan/Temp/beispiel
[INFO] -----
[INFO] BUILD SUCCESS

...
$ cd beispiel
$ mvn test

...
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ beispiel ---
[INFO] Compiling 1 source file to /Users/stefan/Temp/beispiel/target/classes
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ beispiel ---
[INFO] Compiling 1 source file to /Users/stefan/Temp/beispiel/target/test-classes

...
```

Lifecycle Beispiel

- Mit »mvn site« lässt sich die Projektdokumentation erstellen
 - ▶ findet sich dann unter `target/site/index.html`

```
$ mvn site
```

```
...
```

```
[INFO] Rendering site with org.apache.maven.skins:maven-default-skin:jar:1.0 skin.
```

```
[INFO] Generating "About" report
```

```
[INFO] Generating "Plugin Management" report
```

```
[INFO] Generating "Distribution Management" report
```

```
[INFO] Generating "Dependency Information" report
```

```
[INFO] Generating "Source Repository" report
```

```
[INFO] Generating "Mailing List" report
```

```
[INFO] Generating "Issue Tracking" report
```

```
[INFO] Generating "Continuous Integration" report
```

```
[INFO] Generating "Project Plugins" report
```

```
[INFO] Generating "Project License" report
```

```
[INFO] Generating "Project Team" report
```

```
[INFO] Generating "Project Summary" report
```

```
[INFO] Generating "Dependencies" report
```

```
...
```

```
$ open target/site/index.html
```

beispiel

Last Published: 2013-12-19 | Version: 1.0-SNAPSHOT

beispiel

Project Documentation

- ▼ Project Information
 - About
 - Plugin Management
 - Distribution Management
 - Dependency Information
 - Source Repository
 - Mailing Lists
 - Issue Tracking
 - Continuous Integration
 - Project Plugins
 - Project License
 - Project Team
 - Project Summary
- Dependencies

Project Dependencies

test

The following is a list of test dependencies for this project. These dependencies are only required to compile and run unit tests for the application:

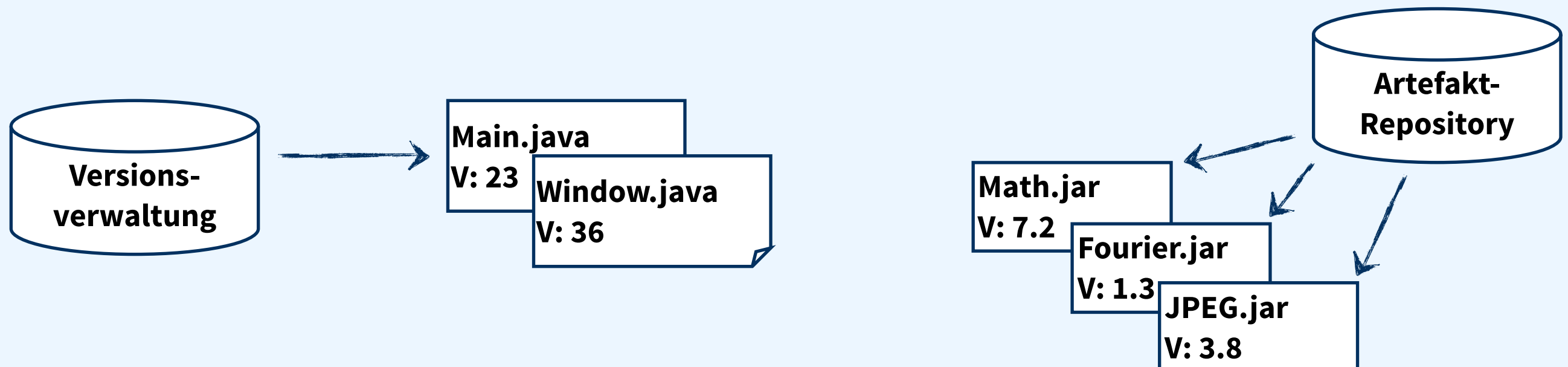
GroupId	ArtifactId	Version	Type	License
junit	junit 	3.8.1	jar	Common Public License Version 1.0 

Artefakte

- Das Ergebnis des Build-Prozesses von Maven wird Artefakt genannt
 - ▶ meist eine Bibliothek oder (Web-)Anwendung
- Artefakte werden über drei »Koordinaten« identifiziert:
 - ▶ groupId → eindeutiger Name des Herstellers des Artefakts
 - ▶ artifactId → eindeutiger Name des Artefakts
 - ▶ version → Versionsnummer des Artefakts
- Jedes Artefakt muss diese Koordinaten definieren
- Artefakte können von weiteren Projekten verwendet werden
 - ▶ indem eine Abhängigkeit zum Artefakt in die `pom.xml` eingefügt wird

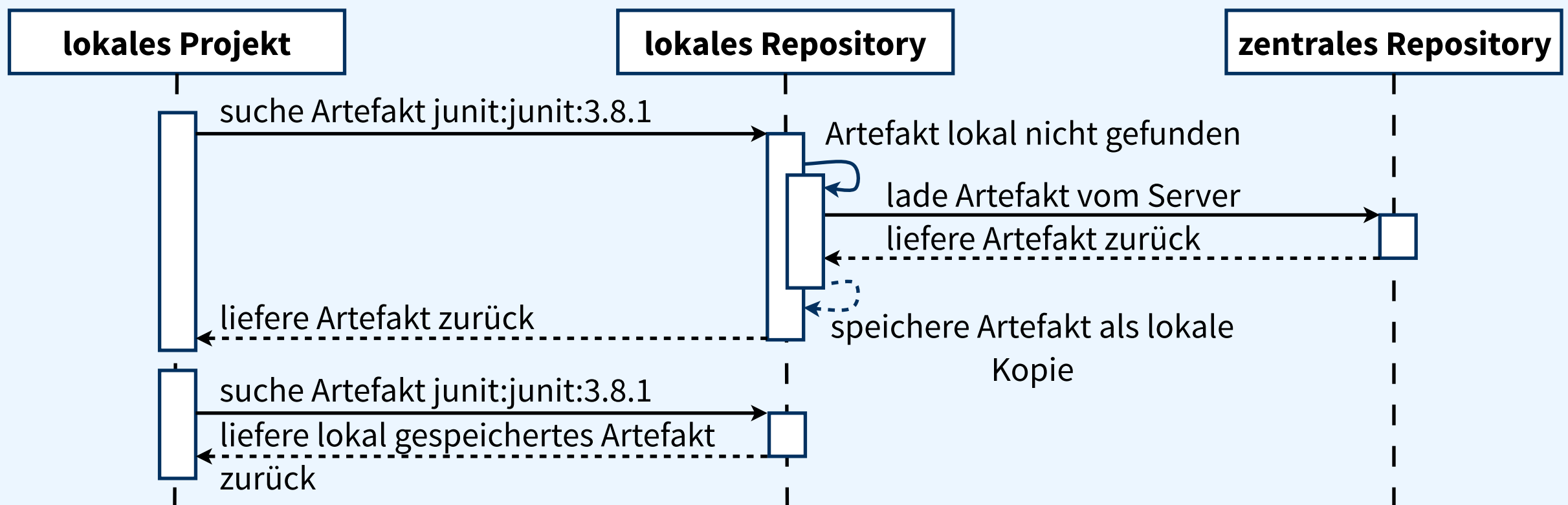
Artefakt-Repository

- Benötigte Artefakte werden von Maven in einem Repository gesucht
- Ein lokales Repository wird von Maven automatisch erstellt
 - findet sich unter *Benutzerverzeichnis/.m2/repository*
- »mvn install« installiert das aktuelle Projekt im Repository
 - andere Projekte können dieses dann als Abhängigkeit verwenden
- Nicht verwechseln mit dem Repository der Versionsverwaltung:



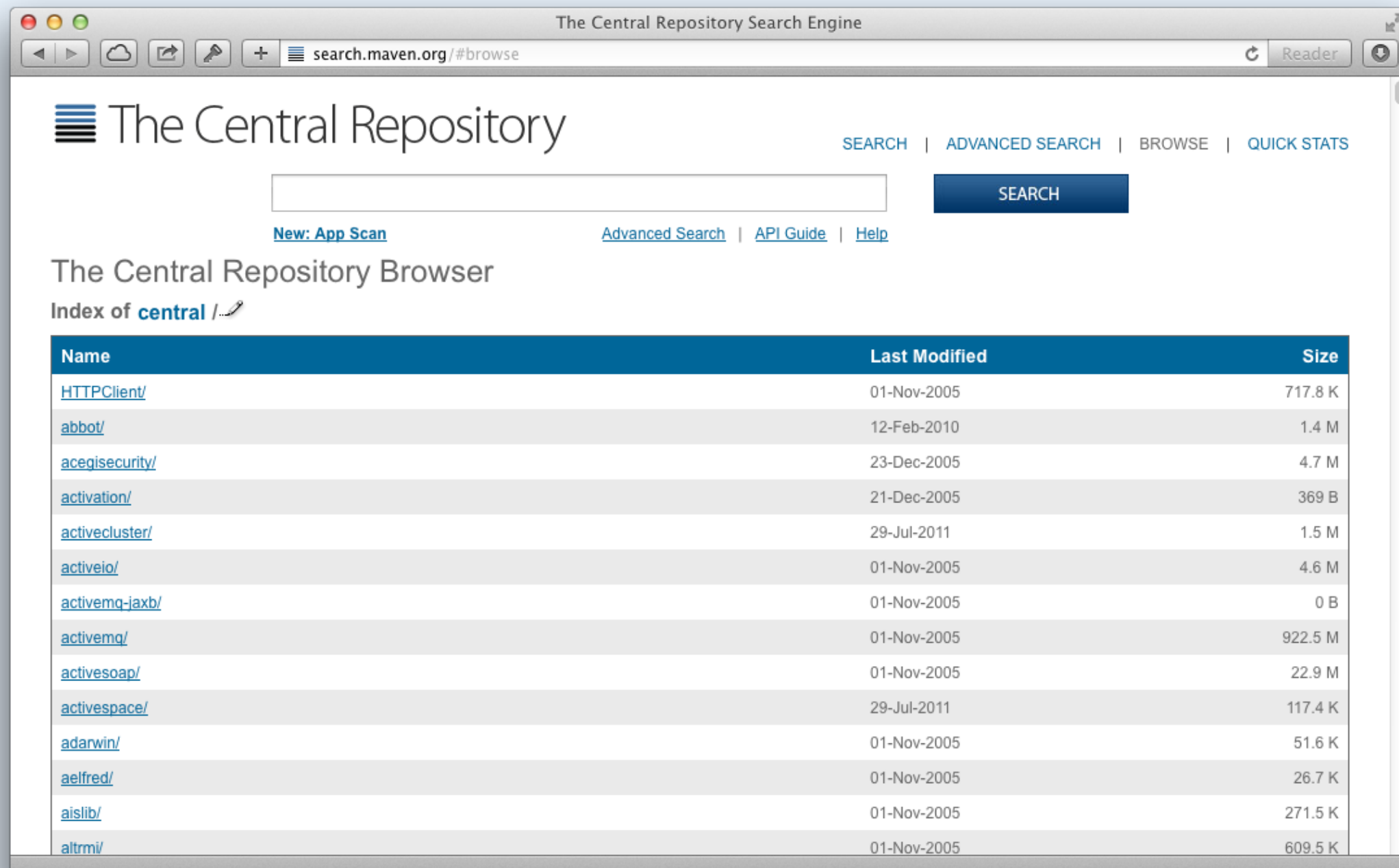
Zentrales Repository

- Wie gelangt Maven an Fremdbibliotheken, die man nicht vorher selber in das lokale Repository installiert hat?
- Maven stellt ein zentrales Repository mit frei verfügbaren Artefakten bereit: `http://repo1.maven.org`
- Das Repository wird automatisch verwendet:



Zentrales Repository

Durchsuchen unter: <http://search.maven.org>



The screenshot shows a web browser window titled "The Central Repository Search Engine" with the address bar displaying "search.maven.org/#browse". The page features a search bar with a "SEARCH" button and navigation links for "SEARCH", "ADVANCED SEARCH", "BROWSE", and "QUICK STATS". Below the search bar, there are links for "New: App Scan", "Advanced Search", "API Guide", and "Help". The main content area is titled "The Central Repository Browser" and shows the "Index of central". A table lists various artifacts with their names, last modified dates, and sizes.

Name	Last Modified	Size
HTTPClient/	01-Nov-2005	717.8 K
abbot/	12-Feb-2010	1.4 M
acegisecurity/	23-Dec-2005	4.7 M
activation/	21-Dec-2005	369 B
activecluster/	29-Jul-2011	1.5 M
activeio/	01-Nov-2005	4.6 M
activemq-jaxb/	01-Nov-2005	0 B
activemq/	01-Nov-2005	922.5 M
activesoap/	01-Nov-2005	22.9 M
activespace/	29-Jul-2011	117.4 K
adarwin/	01-Nov-2005	51.6 K
aelfred/	01-Nov-2005	26.7 K
aislib/	01-Nov-2005	271.5 K
altrmi/	01-Nov-2005	609.5 K

Maven Verzeichnisstruktur

projekt/	Hauptverzeichnis des Projekts
+src/	Alle Quelldateien
+main/	Quelldateien für die Anwendung
+java/	Java Quellcode
+-Klasse.java	
+resources/	Konfigurationsdateien, Bilder, Sprachdateien
+-config.xml	
+-bild.jpg	
+test/	Quelldateien für den Test der Anwendung
+java/	Java Quellcode der Testklassen
+-KlasseTest.java	
+site/	Quelldateien für die Projektdokumentation
+target/	Alle generierten Dateien
+classes/	kompilierte Klassen
+site/	Projektdokumentation
+pom.xml	Maven Konfigurationsdatei

fettgedruckte Elemente = Maven-Konvention

pom.xml

- Project Object Model (POM)
 - ▶ zentrale XML-Konfiguration für Maven
- Trotz Konventionen kann in Maven vieles konfiguriert werden
 - ▶ Projektkoordinaten (groupId/artifactId/version)
 - ▶ Projektart (packaging)
 - ▶ zu erzeugende Dokumentationen
 - ▶ Abhängigkeiten
 - ▶ weitere Automationsplugins
- Komplette Übersicht unter:
<http://maven.apache.org/pom.html>
- minimales Beispiel →

pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>de.betermieux</groupId>
  <artifactId>beispiel</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

</project>
```

pom.xml Projektarten

- Mit dem *packaging*-Element wird das Ergebnis des Build-Prozesses bestimmt
- Je nach Zuweisung wird in der Lifecycle-Phase *package* eine spezifische Aktion ausgeführt:
 - ▶ »jar« → aus den kompilierten Klassen ein JAR-Paket erstellen
 - ▶ »war« → eine Web-Archiv erstellen
 - ▶ »pom« → kein Artefakt erstellen, nur die pom-Datei veröffentlichen
 - ▶ »ear« → Ein Java-Enterprise-Archiv erstellen
- Für die jeweiligen Pakete müssen entsprechend der Konvention Quelldateien im `src`-Ordner abgelegt sein

pom.xml

```
<project>
  ...
  <packaging>jar</packaging>
  ...
</project>
```

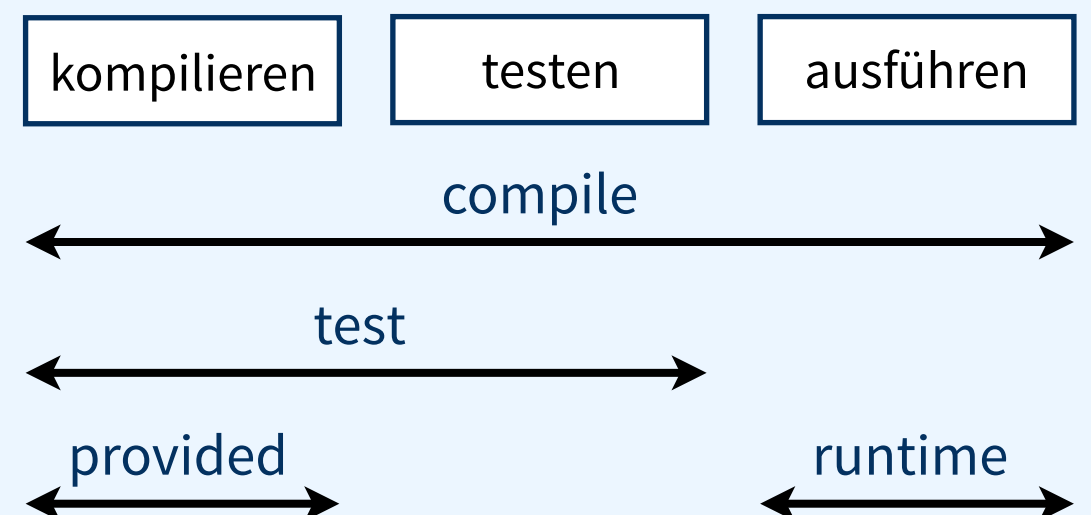
pom.xml Abhängigkeiten

- Abhängigkeiten des Projekts können im dependencies-Element definiert werden
- Zusätzlich zu den Koordinaten des Artefakts sollte noch ein Scope angegeben werden:
 - ▶ *compile* → Standardwert, Abhängigkeit wird zum Kompilieren und Ausführen benötigt
 - ▶ *test* → Abhängigkeit wird nur zum Testen benötigt, wird nicht ausgeliefert
 - ▶ *provided* → nur zum Kompilieren
 - ▶ *runtime* → nur für die Ausführung

pom.xml

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Gültigkeitsbereiche:



Transitive Abhängigkeiten

- Wenn Abhängigkeiten ebenfalls Abhängigkeiten besitzen, werden diese ebenfalls zum Projekt hinzugefügt
 - ▶ transitive Hülle
- Mit »`mvn dependency:tree`« kann der Abhängigkeitsbaum in der Konsole dargestellt werden
 - ▶ *junit* hat keine weiteren Abhängigkeiten:

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building beispiel 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ beispiel ---
[INFO] de.betermieux:beispiel:jar:1.0-SNAPSHOT
[INFO] \- junit:junit:jar:3.8.1:test
...
```

Transitive Abhängigkeiten

- Fügen wir testweise ein Artefakt mit vielen Abhängigkeiten ein →
- Und rufen erneut »mvn dependency:tree« auf:

pom.xml

```
<project>
...
<dependencies>
...
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.0.0.RELEASE</version>
  </dependency>
</dependencies>
</project>
```

```
$ mvn dependency:tree
```

```
...
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ beispiel ---
[INFO] de.betermieux:beispiel:jar:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:3.8.1:test
[INFO] \- org.springframework:spring-webmvc:jar:4.0.0.RELEASE:compile
[INFO]     +- org.springframework:spring-beans:jar:4.0.0.RELEASE:compile
[INFO]     +- org.springframework:spring-context:jar:4.0.0.RELEASE:compile
[INFO]         \- org.springframework:spring-aop:jar:4.0.0.RELEASE:compile
[INFO]     +- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
[INFO]         \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] \- org.springframework:spring-expression:jar:4.0.0.RELEASE:compile
...
```

Projektdokumentation

- Die Erzeugung der Projektdokumentation wird mit »mvn site« angestoßen
 - eigener Lifecycle, unabhängig vom Bauen des Artefakts
- Die erzeugte Webseite verwendet:
 - Projektinformationen aus pom.xml
 - Projektdokumentationen aus dem src/site/-Verzeichnis
 - Projektreports von Plugins, aus den Quelltexten generiert

Vorteile:

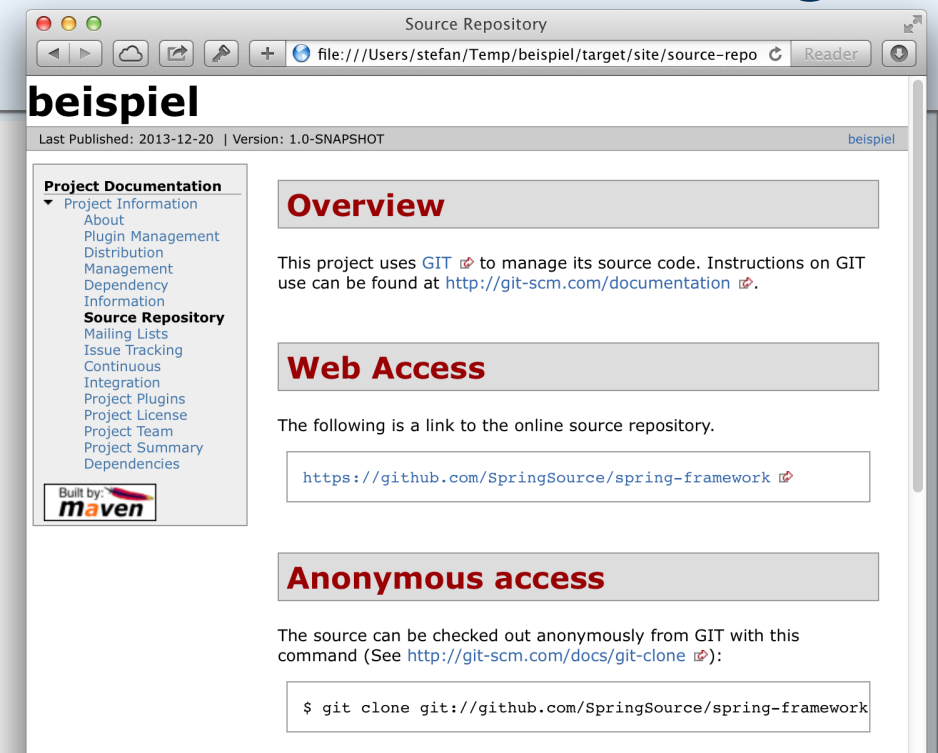
- Artefakt-Quellcode und -Dokumentation liegen im gleichen Verzeichnis
- HTML-Webseite als Ergebnis ist plattformneutral und kann auf externen Rechnern veröffentlicht werden

Projektinformation

- In der `pom.xml` können Metadaten zum Projekt gepflegt werden, z.B.:
 - ▶ `<description/>` Beschreibung des Projekts
 - ▶ `<developers/>` beteiligte Entwickler
 - ▶ `<scm/>` Details zur verwendeten Versionsverwaltung

`pom.xml`

```
<project>
...
<description>Spring Web MVC</description>
<developers>
  <developer>
    <name>Juergen Hoeller</name>
    <email>jhoeller@gopivotal.com</email>
  </developer>
</developers>
<scm>
  <connection>scm:git:git://github.com/SpringSource/spring-framework</connection>
  <url>https://github.com/SpringSource/spring-framework</url>
</scm>
...
</project>
```

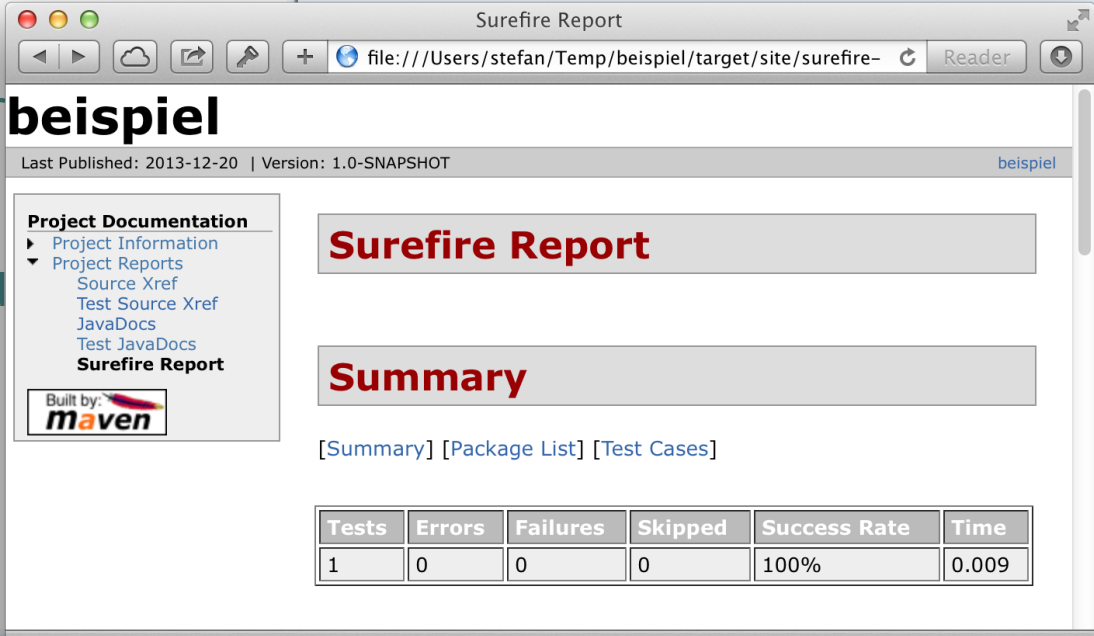


Projektreports

- Reports können bei Bedarf von Plugins beim Bauen des Artefakts generiert und in die Projektdokumentation integriert werden
- Gängige report-fähige Plugins sind z.B.
 - ▶ Surefire-Plugin → Ergebnisse des Test-Durchlaufs dokumentieren
 - ▶ Javadoc-Plugin → Dokumentation der Java-Klassen erstellen

pom.xml

```
<project>
...
<reporting>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-report-plugin</artifactId>
    </plugin>
    <plugin>
      <artifactId>maven-javadoc-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
...
</project>
```



Surefire Report


file:///Users/stefan/Temp/beispiel/target/site/surefire- Reader

beispiel

Last Published: 2013-12-20 | Version: 1.0-SNAPSHOT

Project Documentation

- Project Information
- Project Reports
- Source Xref
- Test Source Xref
- JavaDocs
- Test JavaDocs
- Surefire Report**

Built by: 

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
1	0	0	0	100%	0.009

Projektdokumentation

- Außer den generierten HTML-Dateien (Projektinformationen und Projektreports) können auch Freitexte in die Dokumentation aufgenommen werden
- Maven bietet drei Möglichkeiten, aus Text-Dateien HTML zu erzeugen:
 - ▶ APT → Mavens eigene Markup-Sprache, vergleichbar mit Wikitext und Markdown
 - ▶ XDOC → XML-Dialekt zur semantischen Auszeichnung von Text
 - ▶ FML → XML-Dialekt zur Erzeugung von FAQs
- Die Erstellung der Freitext-Dokumentation werden wir hier nicht weiter betrachten

Maven Plugin für Eclipse

- Eclipse bietet in der JavaEE Version ein vollständiges Maven-Plugin:
 - ▶ <http://www.eclipse.org/m2e/>
 - ▶ benötigt keine lokale Maven-Installation
 - ▶ kann in anderen Eclipse-Varianten auch nachinstalliert werden
- Funktionen:
- Eigener Editor für die `pom.xml`
- Abhängigkeiten werden in Eclipse direkt angezeigt
- Maven kann direkt in Eclipse ausgeführt werden
 - ▶ »Run Configurations...«

Demo Eclipse

The screenshot displays the Eclipse IDE interface with the following components:

- Project Explorer (Left):** Shows the project structure for 'beispiel', including 'src/main/java' (containing 'App.java'), 'src/test/java' (containing 'AppTest.java'), 'JRE System Library [J2SE-1.5]', and 'Maven Dependencies' (listing various JARs like junit-3.8.1.jar, spring-webmvc-4.0.0.RELEASE.jar, etc.).
- Navigator (Top Left):** Provides a quick access bar with icons for Java EE, Debug, SVN Repository Exploring, Team Synchronizing, Resource, and Git Repository Exploring.
- Editor (Center):** Displays the 'beispiel/pom.xml' file. The 'Dependency Hierarchy [test]' view is active, showing a tree of dependencies. The 'Resolved Dependencies' view is also visible on the right side of the editor.
- Bottom Panel:** Contains tabs for 'Overview', 'Dependencies', 'Dependency Hierarchy' (selected), 'Effective POM', and 'pom.xml'.

Dependency Hierarchy [test] (Left Panel):

- junit : 3.8.1 [test]
- spring-webmvc : 4.0.0.RELEASE [compile]
 - spring-beans : 4.0.0.RELEASE [compile]
 - spring-core : 4.0.0.RELEASE [compile]
 - spring-context : 4.0.0.RELEASE [compile]
 - spring-aop : 4.0.0.RELEASE [compile]
 - aopalliance : 1.0 [compile]
 - spring-beans : 4.0.0.RELEASE [compile]
 - spring-core : 4.0.0.RELEASE [compile]
 - spring-beans : 4.0.0.RELEASE [compile]
 - spring-core : 4.0.0.RELEASE [compile]
 - spring-expression : 4.0.0.RELEASE [compile]
 - spring-core : 4.0.0.RELEASE [compile]
 - commons-logging : 1.1.1 [compile]
 - spring-expression : 4.0.0.RELEASE [compile]
 - spring-core : 4.0.0.RELEASE [compile]
 - spring-web : 4.0.0.RELEASE [compile]
 - aopalliance : 1.0 [compile]
 - spring-aop : 4.0.0.RELEASE [compile]
 - spring-beans : 4.0.0.RELEASE [compile]
 - spring-context : 4.0.0.RELEASE [compile]
 - spring-core : 4.0.0.RELEASE [compile]

Resolved Dependencies (Right Panel):

- aopalliance : 1.0 [compile]
- commons-logging : 1.1.1 [compile]
- junit : 3.8.1 [test]
- spring-aop : 4.0.0.RELEASE [compile]
- spring-beans : 4.0.0.RELEASE [compile]
- spring-context : 4.0.0.RELEASE [compile]
- spring-core : 4.0.0.RELEASE [compile]
- spring-expression : 4.0.0.RELEASE [compile]
- spring-web : 4.0.0.RELEASE [compile]
- spring-webmvc : 4.0.0.RELEASE [compile]



ZUSAMMENFASSUNG

Ausblick

- Maven dient als Fundament, um die nächsten Schritte umzusetzen:
- Software-Tests mit Junit
 - ▶ innerhalb des Maven-Lifecycles, in der Phase *test*
- Continuous Integration
 - ▶ zentraler Server führt den Maven-Lifecycle aus
- Repository Manager
 - ▶ eigener Maven-Repository-Server
- Messung der Software-Qualität
 - ▶ Reporting-Plugins messen und dokumentieren Software-Qualität

DANKE