Schöner Bauen mit Maven und Nexus

Johannes Caspary | johannes.caspary@cms.hu-berlin.de

Skripte zur Automatisierung des Kompilierens und Testens von Quellcode beschleunigen die Entwicklung neuer Software. Neben dem weit verbreiteten Apache Ant existieren für die Java-Entwicklung zahlreiche Alternativen, von denen Apache Maven besonders hervorgehoben wird. Ihre Stärken können diese Buildwerkzeuge in Verbindung mit Repository-Managern wie Sonatype Nexus entfalten.

Software-Entwicklung im Allgemeinen

Viele Informatiker erinnern sich wahrscheinlich noch aus dem Studium oder der Ausbildung an den Softwareentwicklungsprozess und seine Phasen. Besonders anschaulich – wenn auch etwas veraltet – ist hierbei das Wasserfallmodell mit den folgenden Phasen:

- *Anforderungserhebung:* Klärung der Frage, was benötigt wird
- *Entwurf:* Formalisierung (Modellierung) der Anforderungen
- *Implementierung:* Umsetzung des Entwurfs
- *Test:* Klärung der Frage, ob die Software die richtigen Dinge tut (und das richtig)
- *Verteilung:* Phase der Produktivschaltung der Software

Die meisten Entwicklungsmodelle basieren im Wesentlichen auf diesen Abläufen. Allerdings gibt es bei den agilen Entwicklungsmethoden die Forderung, Test und Implementierung in der Reihenfolge umzudrehen (siehe zur testgetriebenen Entwicklung u.a. [1]). Nach Ablauf aller Phasen entsteht ein sogenanntes *Artefakt* – die fertige Software.

Ebenso herrscht Konsens, dass man die genannten Schritte wiederholen muss. Es ist unrealistisch anzunehmen, dass qualitativ hochwertige Software „im Ganzen" in einem Durchlauf entwickelt werden kann. Man spricht bei der Wiederholung der Phasen von einer iterativen Entwicklung. Dabei wird die Software sukzessive vervollständigt. Die Ergebnisse der Iterationen bezeichnet man als *Inkremente*. Zum Abschluss jeder Phase entstehen so weitere Dokumente wie Lastenhefte, Modelle, Quellcode und Testfälle.

Der oben beschriebene Softwareentwicklungsprozess bezieht sich vor allem auf größere Projekte, aber auch mittlere und kleine profitieren von einem strukturierten Vorgehen. Im universitären Bereich werden deshalb die Prinzipien der Softwareentwicklung berücksichtigt.

Automatisierungsmöglichkeiten in der Entwicklung

Die Anforderungserhebung ist der Ausgangspunkt für den gesamten Entwicklungsprozess. Eine ausführliche und zielorientierte Analyse ist zur Erstellung von Software dringend erforderlich. Zwar existieren diverse Werkzeuge zur professionellen Anforderungserhebung und zum Anforderungsmanagement, aber in der Regel hat man es mit einfachen Textdokumenten zu tun – was oft auch völlig ausreicht.

In der Stufe des Entwurfs wird das noch relativ ungenaue Bild der geforderten Software formalisiert. Dabei bedient man sich der Visualisierung in Form von graphischen Modellen – seien sie nun statisch, ablauf- oder datenflussorientiert. Auch wenn die Modelle einen hohen Detaillierungsgrad und Nähe zur Implementierung besitzen können, findet etwa die graphische Programmierung noch relativ selten Anwendung. Im Wesentlichen geht es hier noch darum, „Bilder zu malen", die eine Brücke zwischen den Anforderungen und der späteren konkreten Umsetzung bauen.

Erst in der Stufe der Implementierung können wir auf Grundlage des Quellcodes mit der Automatisierung des Entwicklungsprozesses beginnen. Abhängig vom Ausmaß der Automatisierbarkeit des Prozesses unterstützen Werkzeuge die Phasen Implementierung, Testen und Verteilung. An dieser Stelle kommen sogenannte *Buildwerkzeuge* zum Einsatz. Ein *Build* umfasst den Ablauf gewisser Routinen im Erstellungsprozess, wie Kompilierung und das Ausführen von Tests, und endet mit der Erstellung eines Artefakts. Im Kontext von Java handelt es sich üblicherweise um JAR- oder WAR-Dateien.[1]

Im Folgenden wird als Beispiel eines Buildwerkzeugs Apache Maven erläutert. Um Maven im Vergleich zu anderen Werkzeugen abzugrenzen, wird auch auf die derzeit verbreiteten Alternativen kurz eingegangen.

Das Buildwerkzeug Maven

Maven [2] ist ein Projekt der Apache Software Foundation [3] und dient zur Umsetzung der Buildautomatisierung für Java-Projekte. Das aktuelle Release zur Zeit der Erstellung dieses Artikels ist Maven 3.0.3, aber auch das Release 2.2.1 ist noch weit verbreitet. Eine sehr gute Anleitung zur Software auf Deutsch findet sich unter [4].

Zur Ausführung von Maven gibt es die Kommandozeilenanwendung *mvn*, die für die Verwendung in Skripten hilfreich ist. Allerdings lässt sich Maven auch über Plugins in Entwicklungsumgebungen wie Eclipse [5] oder IntelliJ IDEA [6] integrieren.

Das wichtigste Dokument für ein Maven-Projekt ist die Projektbeschreibungsdatei *pom.xml*. Dort werden zentral die Eigenschaften des Projekts konfiguriert. Maven interpretiert bei seiner Ausführung lediglich diese Konfiguration und braucht keine zusätzlichen Skriptanweisungen. Damit vertritt Maven einen deklarativen Ansatz, der sich von anderen verbreiteten Buildwerkzeugen wie Ant [7] oder make unterscheidet.[2]

Ein wesentliches Prinzip von Maven ist die sogenannte „Convention Over Configuration“ [8]. Gemeint ist damit, dass man aus einer Konvention heraus für die Projektstruktur plausible Standardwerte annimmt. So werden beispielsweise Java-Klassen, die im Namen das Wort „Test“ enthalten, automatisch als Unit-Test[3] interpretiert [9]. Dennoch ist eine Anpassung der Variablen, wie der Verweise auf die Quelltextordner oder das Outputverzeichnis, explizit in der Projektbeschreibungsdatei möglich. Bei der Entwicklung kann man sich nun entscheiden, ob man die vorgegebene Struktur übernimmt oder sie dem eigenen Projekt anpasst.

Im Zusammenhang mit Maven begegnet man häufig dem Begriff *Plugin* [10]. Ein Maven-Plugin ist ein Zusatzprogramm, das den funktionalen Kern von Maven erweitert. Tatsächlich stellt der Maven-Kern nur ein leichtgewichtiges Framework zur Ausführung von Plugins dar, die fast sämtliche den Buildprozess betreffenden Aufgaben übernehmen [11].

Standardphasen in Maven

Der Buildprozess in Maven wird, ähnlich wie beim Wasserfallmodell, in Phasen eingeteilt. Im Standardlebenszyklus eines Builds, der hier verkürzt wiedergegeben ist, sind die Phasen Validierung *(validate)*, Kompilieren *(compile)*, Test *(test)*, Paketierung *(package)*, Integrationstest *(integration-test)*, Verifikation *(verify)*, Installation *(install)* und Verteilung *(deploy)* definiert [12].

Die Phasen werden sequenziell abgearbeitet. So bewirkt das Kommando *mvn test*, dass neben der Phase *test* auch *validate* und *compile* ausgeführt werden, nicht aber *package*. Durch die strikte Vorgabe des Ablaufs des Buildprozesses kann Maven, im Vergleich zu Ant, in Standardanwendungen auf die explizite Vorgabe von Zielen und deren Abhängigkeiten verzichten.

Wie die Namen der Phasen erkennen lassen, unterstützt Maven nicht nur im Rahmen der Implementierung das Kompilieren, sondern auch die Schritte des Tests (*test* für Unit-Tests und *integration-test* für Tests auf Komponenten) und der Verteilung (*install* zur Speicherung auf dem lokalen und *deploy* zur Verteilung auf einen entfernten Rechner).

Auflösen von Projektabhängigkeiten – Dependency Management

Die Definition von Abhängigkeiten eines Projekts zu anderen Artefakten (insbesondere JAR-Dateien), erfolgt über die Projektbeschreibungsdatei pom.xml. Um die Wiederverwendbarkeit zu erhöhen, können sie durch Vererbung oder Import an andere Projekte weitergegeben werden.

Die Auflistung dient nicht nur der reinen Beschreibung der Software. Maven nutzt die Informationen vor allem für seinen Mechanismus zur Auflösung von Abhängigkeiten (*Dependency-Management*). Ausgehend von den gefundenen Abhängigkeiten sucht Maven nach Artefakten in Verzeichnissen mit einer definierten Struktur, den sogenannten *Repositories*. Zuerst wird das lokale Repository auf dem Rechner durchsucht. Ist dort das gesuchte Artefakt nicht in der korrekten Version vorhanden, sucht Maven in einem entfernten Repository.

Standardmäßig lädt Maven fehlende Artefakte von seinem zentralen Repository [13] herunter. Dadurch erhält man zwar immer den aktuellen Versionsstand, besitzt aber wenig Einfluss auf die ausgewählten Artefakte. Durch Anpassung der Maven-Einstellungen kann aber auch auf alternative Repositories verwiesen werden.

Alternativen zu Maven

Abgesehen von allen Vorzügen, die Maven zweifelsohne bietet, gibt es auch Kritikpunkte. Zum einen gibt Maven durch das Prinzip „Convention Over Configuration“ Projekten eine recht strikte Form vor. Sicher sind strukturelle und ablauftechnische Anpassungen auch mit Maven möglich, sie erfordern allerdings unter Umständen einen

1 JAR = Java Archiv, Dateiformat zur Paketierung von Java-Bibliotheken; WAR = Web Archiv, Dateiformat zur Speicherung von servletbasierten Java-Webanwendungen. Die Archive liegen jeweils im ZIP-Format vor.

2 Dennoch können Ant-Skripte auch in Maven integriert werden.

3 Unit-Tests testen die Teilfunktionalität eines Programms (z. B. Methoden).

relativ hohen Aufwand.[4] Zum anderen gilt die Methode, das Projekt über XML-Dateien zu beschreiben, als zu statisch und unhandlich.[5]

Neben Maven existieren weitere Build-Managementwerkzeuge, die mit unterschiedlichen Strategien die genannten Schwachpunkte angehen. Zur Erhöhung der Flexibilität verwenden andere Werkzeuge objektorientierte Skriptsprachen zur Definition des Buildprozesses. Verbreitet sind vor allem Ruby (z. B. buildr [16] und Raven [17]) und Groovy (z. B. Gradle [18] und Gant[6] [19]).

In Hinblick auf das Dependency-Management ist Apache Ivy [20] ein interessantes Projekt. Dieses Werkzeug wird in Kombination mit Ant genutzt und adressiert das Problem der einheitlichen Verwaltung von JAR-Dateien über Repositories. So kann es sowohl auf Maven-Repositories zugreifen, als auch Repositories mit Ivy-spezifischer Struktur verwalten.

Für sich alleine zählt Ivy nicht zu den Buildwerkzeugen, da es nur eine Teilfunktionalität umsetzt. Allerdings nutzt z. B. Gradle das Dependency-Management von Ivy, und auch Ant kann durch die Verwendung von Ivy-Tasks erweitert werden. Buildr verfügt über ein eigenes Dependency-Management, welches mit Maven-Repositories kompatibel ist.

Beschreibung Repositories

Repositories im Zusammenhang mit Maven sind als Softwareverzeichnisse mit einer definierten Struktur und beschreibenden Metadaten zu verstehen. Das Maven-Verzeichnislayout kann auch durch andere Buildwerkzeuge gelesen werden. In das Repository werden die durch den Buildprozess erstellten Artefakte gespeichert, die ebenfalls hinterlegte Projektbeschreibungsdatei pom.xml enthält die Metadaten.

Bei der Lokalinstallation speichert Maven seine durch das Dependency-Management aufgelösten Abhängigkeiten in ein Repository auf dem Rechner. Nicht dort befindliche Dateien werden anschließend in dem öffentlich verfügbaren, zentralen Repository durchsucht. Es gibt aber Argumente, die dafür sprechen, die vorgegebene Struktur anzupassen und auch eigene Repositories im Netzwerk zu hinterlegen.

Gründe für eigene Repositories

Software-Entwicklung, vor allem bei mittleren und größeren Projekten, ist Teamwork. Projekte und Bibliotheken müssen untereinander immer abgestimmt sein und auf verteilten Rechnern die gleiche Datenbasis nutzen. Das lokale Speichern von Dateien reicht nicht aus.

Was die Verwaltung von Änderungen auf Dateisystemebene betrifft, existieren leistungsfähige Versionskontrollsysteme wie SVN [21], Git [22] oder Mercurial [23]. Im Rahmen des AGNES-Teams [24] wird der SVN-Dienst des Computer- und Medienservice (CMS) verwendet [25]. Um jedoch in vollem Umfang das im vorigen Abschnitt besprochene Dependency-Management zu nutzen, muss man in Kombination mit Maven auf Repositories zurückgreifen.

Es sind zwei Problemstellungen zu lösen: 1. Wie veröffentliche ich eigene Artefakte? Und 2. Wie bekomme ich Artefakte, die nicht in öffentlich verfügbaren Repositories verwaltet werden?

Zu Punkt 1 gäbe es grundsätzlich die Möglichkeit, eigene Artefakte in das zentrale Maven-Repository hochzuladen. Dies hätte allerdings die Konsequenz, dass die ganze Welt darauf Zugriff hat. Wenn das nicht gewünscht ist, kann die Einrichtung eines organisationseigenen Repositories mit beschränktem Zugriff die Lösung sein.

Zu Punkt 2 gibt es bei einigen Artefakten zudem lizenzrechtliche Einschränkungen, die der Veröffentlichung über die zentralen Repositories entgegenstehen. In diesen Fällen kann über ein selbstverwaltetes Repository der Zugriff auf diese proprietären Pakete ermöglicht werden, ohne die rechtlichen Vorgaben zu verletzen.

Die Verwendung eines eigenen Repositories hat für eine Organisation viele Vorteile. Man besitzt dadurch nicht nur Kontrolle über den Inhalt (z. B. durch Aussperren ungewollter Artefakte), sondern steuert auch die Verfügbarkeit über das eigene Netzwerk. Wäre das zentrale Repository nicht zu erreichen, könnten ggf. für diesen Zeitraum keine Maven-Projekte gebaut werden.

Ein ganz wesentlicher Punkt ist der Sicherheitsaspekt. Öffentliche Repositories besitzen immer das Gefahrenpotenzial, dass die bezogenen Artefakte, auf welchen Wegen auch immer, schadhafte Software enthalten. Eine restriktive Verwaltung des Repository-Managers hilft hierbei.

Ein weiterer Vorteil liegt in der Möglichkeit, das selbst verwaltete Repository als Proxy zu anderen öffentlich verfügbaren Repositories zu nutzen. Dadurch werden die für die eigenen Projekte benötigten Artefakte zwischengespeichert, was sich in weniger Anfragen und Downloads bei den zentralen Repositories niederschlägt. Bei kleineren Projekten und wenigen Nutzern ist der Effekt jedoch eher gering.

Sonatype Nexus und andere Repository-Manager für Maven

Insbesondere aufgrund der genannten Vorteile hinsichtlich der kontrollierten Verfügbarmachung eigener sowie fremder, lizenzrechtlich eingeschränkter Artefakte, wurde in der Abteilung „DV in der Verwaltung“ des CMS die Einrichtung eines internen Repositories beschlossen. Da die manuelle Pflege eines solchen Repositories aufwändig und fehlerträchtig ist, wurde hierfür ein Repository-Manager ausgewählt. Bei dieser Installation handelt es sich um einen Testeinsatz im kleinen Umfeld, daher wurde ein frei verfügbares Programm genutzt.

In seiner Funktionalität sollte der Repository-Manager als Proxy für entfernte Repositories, als Speicherort für unternehmensinterne JARs und als Verwaltungstool für die Zugriffsrechte

4 Das Programmieren eigener Plugins und deren Einbindung in den Buildlebenszyklus ist eine Möglichkeit, siehe auch unter [14].

5 Für Maven 3 existiert das Polyglot Maven-Projekt [15], das ermöglichen will, dass die Projektbeschreibung nicht mehr ausschließlich in XML, sondern auch alternativ in Groovy und ähnlichen Sprachen ermöglicht wird. Bis zur Fertigstellung des Artikels gab es keine Umsetzung dieser Funktionalität für Maven 3.

6 Bei Gant dient Groovy lediglich zum Ersatz der XML-Struktur von Ant.

auf die einzelnen Repositories und Artefakte dienen. Das Programm sollte auf Debian Linux-Servern lauffähig sein und nach Möglichkeit über eine einfach zu bedienende Weboberfläche verfügen.

Auf dem Markt existieren vor allem drei Repository-Manager: *Apache Archiva* [26], *Artifactory* [27] und *Sonatype Nexus* [28] (früher Proximity). Allen ist gemeinsam, dass sie in Java programmiert und als WAR Webarchiv lauffähig sind. Zur Bedienung des Repository-Managers besitzen alle eine Weboberfläche, die einen leistungsfähigen, auf Lucene [29] basierenden Suchmechanismus zur Verfügung stellt.

Der erste verfügbare Maven Repository-Manager war Sonatype Nexus. Sonatype bietet eine Open-Source-Version unter der Bezeichnung *Nexus OSS* an und wird unter der Lizenz GNU Affero GPL Version 3 (AGPLv3) bereitgestellt. Er erfüllt die angegebenen Kriterien und verfügt darüber hinaus über eine LDAP-Integration, eine einfache Adressierungsmöglichkeit der Repositorygruppen über eine einzige URL sowie geplante Tasks (z. B. zum Aufräumen alter Snapshots) [30]. Das Repository wird auf dem Dateisystem im Maven 2-Layout[7] abgelegt.

Ein großer Vorteil von Nexus ist die umfangreiche Dokumentation, die in Form eines kompletten Buches in HTML und PDF vorliegt [32]. Dies erleichtert Installation und Konfiguration erheblich. Als kommerzielles Pendant wird der Nexus Professional vertrieben (2995 US-Dollar für 50 Nutzer). Er verfügt über erweiterte Suchfunktionen, die Möglichkeit der Projektwebseitenintegration und feingranulare Einstellmöglichkeiten in Hinsicht auf Validierung und Verifikation von Artefakten [33].

Seit 2006 wird der Repository-Manager Artifactory von JFrog angeboten. Auch dieser teilt sich in eine frei verfügbare und eine kommerzielle Version auf. In puncto Interoperabilität und Funktionalität übertrifft Artifactory den Nexus OSS, z. B. durch die hierarchische Gliederung von Nutzergruppen und erweiterte Suchoptionen. Die Speicherung der Artefakte und ihrer Metadaten erfolgt standardmäßig über eine Datenbank, alternativ können die Artefakte aber auch direkt im Dateisystem abgelegt werden. Artifactory unterstützt keine Maven 1-Clients und verbraucht wegen umfangreicher Indizierung mitunter mehr Speicherplatz auf dem Datenträger als Nexus.[8]

Der Dritte im Bunde ist Apache Archiva. Dieses Top-Level-Projekt von Apache wurde 2005 gestartet und im November 2007 das erste Produktionsrelease veröffentlicht. Es handelt sich um ein vollständig frei verfügbares Werkzeug unter der Apache License 2.0. Archiva ist etwas funktionsärmer als Nexus OSS und Artifactory, insgesamt aber vergleichbar. Das Repository wird wie bei Nexus im Dateisystem auf dem Server gespeichert. Zur Verwaltung der Daten nutzt Archiva standardmäßig eine Derby DB.

Eine ausführliche Gegenüberstellung der Eigenschaften der genannten drei Repository-Manager findet sich auch unter [36].

Maven und Sonatype Nexus im Einsatz

Im AGNES-Team des CMS wird schon seit längerem für Java-Projekte nach Möglichkeit Maven eingesetzt. Der größte Vorteil liegt im Dependency-Management. Für die ersten Projekte mit relativ standardisierten und breit verfügbaren JAR-Bibliotheken wie *JDOM* für die XML-Verarbeitung oder *log4j* für das Logging war das voreingestellte Maven-Repository völlig ausreichend.

Im Laufe der Zeit ergaben sich allerdings Probleme durch den Ansatz, die JAR-Dateien ausschließlich aus dem zentralen Repository zu beziehen. Zur besseren Wiederverwendung gliederte man selbst entwickelte Funktionalitäten in eigene Projekte aus und benötigte eine Möglichkeit zur Speicherung. Weiterhin wollte man Artefakte nutzen, die aus lizenzrechtlichen Gründen nicht öffentlich verfügbar sind. Diese mussten ungünstigerweise von Hand in die Projekte eingefügt werden, was den Nutzen des Dependency-Managements für die Entwicklung stark reduzierte.

Um die gewünschte Flexibilität wiederzuerlangen, wurde ein eigenes, zunächst auf das AGNES-Team beschränktes Repository eingesetzt. Die Wahl fiel auf den Sonatype Nexus OSS, vor allem aufgrund der umfangreichen Dokumentation. Nach erfolgreichen lokalen Tests und einem positiven ersten Eindruck wurde diese Anwendung als Repository-Manager weiter untersucht. Abbildung 1 zeigt schematisch den stufenweisen Zugriff auf das Nexus-Repository von einem lokalen Rechner.

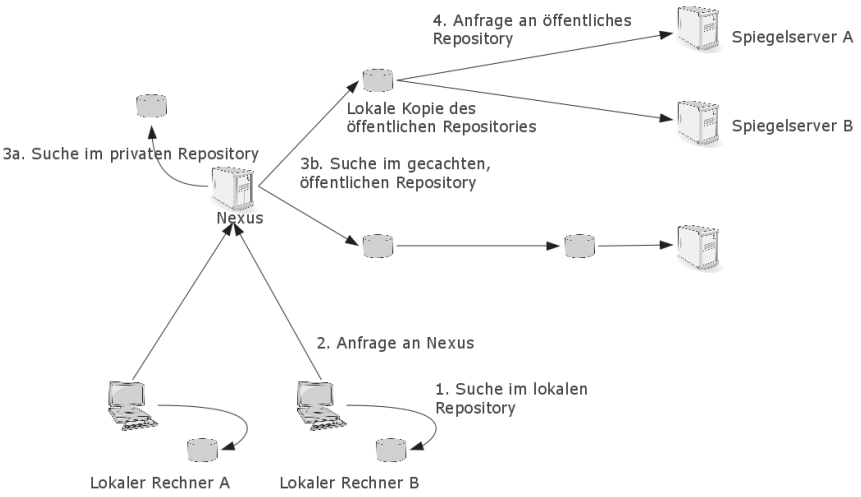

Abb. 1: Infrastruktur und Suche nach Artefakten mit Nexus

7 siehe Erläuterung hierzu unter [31].

8 Ein interessanter Schlagabtausch zwischen den Entwicklern findet sich unter [34] und [35].

Als Server für Nexus wurde von der Wintech-Gruppe des CMS ein virtueller Debian-Server eingerichtet. Die Anwendung selbst ist passwortgeschützt innerhalb des Universitätsnetzes zu erreichen. Die SSL-Verschlüsselung[9] erfolgt über ein durch den PKI-Service [37] der HU bereitgestelltes signiertes Server-Zertifikat.

Inzwischen nutzt auch das Projekt HU-IAM[10] der Abteilung „DV in der Verwaltung" Nexus intensiv. Auf diese Weise wird die Installation von Nexus weiter getestet und verbessert. Inzwischen läuft der Repository-Manager in Version 1.9 sehr stabil und ist über den HU-eigenen Nagios-Dienst [38] in die Systemüberwachung eingebunden.

Aussicht

Nach den guten ersten Erfahrungen soll in Zukunft die Zugriffsstruktur des Repository-Managers überarbeitet werden. Wichtig hierfür sind insbesondere die Definition von verschiedenen Rollen für lesende und in das Repository schreibende Nutzer. Der Einsatz von Nexus Professional ist nicht geplant. Da bisher nur Nexus OSS als Repository-Manager genutzt wurde, ist eine Evaluation der Funktionalität und der Verlässlichkeit von Artifactory und Archiva sinnvoll.

9 SSL = Secure Sockets Layer

10 HU-IAM = Identitätsmanagement der Humboldt-Universität zu Berlin

Literaturverzeichnis

[1] Beck, K.: *Test-Driven Development By Example*. Addison-Wesley, 2003

[2] *Apache Maven Project*. http://maven.apache.org/

[3] *The Apache Software Foundation*. http://apache.org/

[4] *Maven 2.2.1 und Maven 3.0*. http://www.torsten-horn.de/techdocs/maven.htm

[5] *Eclipse – The Eclipse Foundation open source community website*. http://www.eclipse.org/

[6] *IntelliJ IDEA – The Most Intelligent Java IDE*. http://www.jetbrains.com/idea

[7] *Apache Ant*. http://ant.apache.org/

[8] *Sonatype – Convention Over Configuration*. http://www.sonatype.com/books/mvnref-book/reference/installation-sect-convention Configuration.html

[9] *Maven Surefire Plugin*. http://maven.apache.org/plugins/maven-surefire-plugin/howto.html

[10] *Maven – Available plugins*. http://maven.apache.org/plugins/index.html

[11] *Sonatype – Universal Reuse through Maven Plugins*. http://www.sonatype.com/books/mvnref-book/reference/installation-sect-universal-reuse.html

[12] *Sonatype – Chapter 4. The Build Lifecycle*. http://www.sonatype.com/books/mvnref-book/reference/lifecycle.html#lifecycle-sect-default

[13] *Index of /maven2/*. http://repo1.maven.org/maven2/

[14] *Sonatype – Chapter 7. Maven Configuration*. http://www.sonatype.com/books/mvnref-book/reference/configuring.html

[15] *Sonatype / polyglot-maven – github*. https://github.com/sonatype/polyglot-maven

[16] *Buildr – Apache Buildr*. http://buildr.apache.org/

[17] *Raven*. http://raven.rubyforge.org/

[18] *Gradle*. http://www.gradle.org/

[19] *Gant*. http://gant.codehaus.org/

[20] *Apache Ivy*. http://ant.apache.org/ivy/

[21] *Apache Subversion*. http://subversion.apache.org/

[22] *Git – Fast Version Control System*. http://git-scm.com/

[23] *Mercurial SCM*. http://mercurial.selenic.com/

[24] *Humboldt-Universität zu Berlin*. Webseite des AGNES-Portals: https://agnes.hu-berlin.de

[25] *SvnBenutzerHowTo*. http://twiki.cms.hu-berlin.de/svn/svnbenutzerhowto.html

[26] *Archiva: The Build Artifact Repository Manager*. http://archiva.apache.org/

[27] *JFrog – Home of Artifactory Binary Repository Manager*. http://www.jfrog.com/

[28] *Sonatype – Nexus*. http://nexus.sonatype.org/

[29] *Apache Lucene*. http://lucene.apache.org/

[30] *Sonatype – Nexus Open Source*. http://www.sonatype.com/books/nexus-book/reference/ch01s02.html

[31] *Repository Layout – Final – Maven – Codehaus*. http://docs.codehaus.org/display/MAVEN/Repository+Layout+-+Final

[32] *Sonatype – Repository Management with Nexus*. http://www.sonatype.com/books/nexus-book/reference/

[33] *Sonatype – 1.3. Nexus Professional*. http://www.sonatype.com/books/nexus-book/reference/ch01s03.html

[34] *Sonatype Blog – Contrasting Nexus and Artifactory*. http://www.sonatype.com/people/2009/01/contrasting-nexus-and-artifactory/

[35] *From the Frog's Mouth*. http://blogs.jfrog.org/2009/01/contrasting-artifactory-and-nexus.html

[36] *Maven Repository Manager Feature Matrix*. http://docs.codehaus.org/display/MAVENUSER/Maven+Repository+Manager+Feature+Matrix

[37] *PKI-Services – Computer- und Medienservice*. Humboldt-Universität zu Berlin, http://www.cms.hu-berlin.de/dl/zertifizierung

[38] *Nagios – The Industry Standard in IT Infrastructure Monitoring*. http://www.nagios.org/