# Softwareinfrastruktur für SIMPL

René Rehn

**Zusammenfassung:** In der heutigen Softwareentwicklung werden zahlreiche Programme und Entwicklungstools eingesetzt die die verschiedenen Prozesse innerhalb der Softwareentwicklung erheblich vereinfachen. Allerdings ist es dabei wichtig das Entwicklungsteams eine einheitliche Entwicklungsumgebung und Infrastruktur nutzen um Probleme zu vermeiden. Diese Ausarbeitung stellt einige Entwicklungstools die im Studienprojekt SIMPL genutzt werden können vor und erläutert auch einige der Vorteile durch die Nutzung dieser Programme. Es wird zu Beginn kurz auf Studienprojekte im Allgemeinen eingegangen um die unterschiede im Vergleich zu einem Softwarepraktikum zu verdeutlichen. Im weiteren Teil wird auf die Infrastruktur mit dem Schwerpunkt der Entwicklungstools eingegangen.

## 1. Große Softwareprojekte

In diesem Teil der Ausarbeitung möchte ich auf Große Softwareprojekte und das Studienprojekt im Besonderen eingehen. Dazu habe ich mich mit anderen Gruppen die bereits ein Studienprojekt durchgeführt haben unterhalten um auf verschiedene Probleme und deren Lösungen eingehen zu können. Weiterhin habe ich persönliche Überlegungen mit eingebracht und mich mit einem Fachbuch über Software Engineering [1] beschäftigt.

### 1.1 Vergleich Studienprojekt und Softwarepraktikum

Ein Studienprojekt unterscheidet sich stark vom Softwarepraktikum im Vordiplom. Während im Softwarepraktikum nur in 3er Gruppen gearbeitet wird, liegt die Gruppengröße bei einem Studienprojekt bei 6 bis 12 Leuten. Auch die Komplexität und die Dauer eines Studienprojekts sind weit höher als im Softwarepraktikum. Während im Softwarepraktikum meist ein kleineres oder mittleres Softwareprogramm im Laufe eines Semesters erstellt wird, arbeitet die Projektgruppe im Studienprojekt 2 Semester lang an einem weit größeren Programm. Diese Unterschiede führen zu einigen Änderungen die es bei der Durchführung des Projektes zu beachten gibt.

Während im Softwarepraktikum nur wenig Organisationsaufwand notwendig ist, ist eine echte Organisation im Studienprojekt sehr wichtig. Bei einer Gruppengröße von bis zu 12 Leuten ist es wichtig, dass die Leute gemeinsam an dem Projekt arbeiten. Im Softwarepraktikum war es dagegen möglich das sich die Mitglieder der Gruppe nur untereinander Absprachen wer welchen Teil der Spezifikation erstellt oder wer welches Modul bearbeitet und dies anschließend selbstständig erledigen und dies in einem nächsten Treffen gemeinsam zusammengeführt wurde.

In einem Studienprojekt ist eine derartige Herangehensweise nicht möglich, da die Komplexität der zu erstellenden Software es nicht zulässt. Es wäre ein sehr großer Aufwand die einzeln erstellten Artefakte zu einem Endprodukt oder Dokument zu kombinieren. Daher ist es notwendig, dass die Mitglieder des Projektteams gemeinsam an den verschiedenen Artefakten arbeiten und sich die Individuelle Arbeit auf Kleinigkeiten oder zuvor Besprochenes beschränkt.

Weiterhin ist eine echte Rollenverteilung im Studienprojekt notwendig. Während es im Softwarepraktikum meist nur eine Person gibt die eine Rolle vergleichbar mit der eines Projektleiters übernimmt, ist es im Studienprojekt notwendig das die verschiedenen Rollen die es neben dem Projektleiter gibt auch durchgeführt werden.

Bei einem Projekt mit einer solchen Komplexität ist es notwendig das es einen Architekten gibt der sich sowohl um die Struktur der Software als auch um die Erstellung und Programmierung der Schnittstellen sowie um die Integration kümmert. Auch die Qualitätssicherung ist nichts, was die Mitglieder des Teams nebenbei erledigen können, auch hier ist eine Person notwendig die dies erledigt und sich darauf konzentriert. Bei einem Projekt der Größe des Softwarepraktikums ist es möglich die Qualitätssicherung wie den Test ans Ende des Projektes zu verschieben, beim Studienprojekt ist auch dies eine der Aufgaben die kontinuierlich während des Projektverlaufs notwendig ist.

Auch muss es im Studienprojekt einen echten und festen Projektleiter geben. Sowohl das Fehlen eines echten/guten Projektleiters als auch ein oftes Wechseln des Projektleiters wird zu Problemen führen, da diese Person entscheidend für das Vorankommen des Projektes ist. Auch der Kundenkontakt, das Organisieren von Treffen und die Aufgabenverteilung müssen von ihm Erledigt werden.

Weiterhin tritt es im Softwarepraktikum auf, dass die Mitglieder der Projektgruppe alle Aufgaben erledigen. Im Studienprojekt dagegen ist eine klare Teilung der Rollen zu empfehlen. Da die zu erstellende Software weit größer ist als im Softwarepraktikum ist es notwendig, dass zum Beispiel ein Architekt nur mit seiner Aufgabe beschäftigt ist und sich auf diese konzentrieren kann. Ein fehlen einer klaren Struktur oder von fehlerhaften oder nicht vorhandenen Schnittstellen am Ende des Projektes führt zu großen Problemen, deren Lösung viel Zeit benötigt.

Ein weiterer großer Unterschied zwischen Softwarepraktikum und Studienprojekt ist die Zeit und die Zeit die der Projektgruppe zur Verfügung steht. Das Softwarepraktikum hatte eine Laufzeit von einem Semester mit fest definierten Meilensteinen. Das Studienprojekt dagegen hat eine Laufzeit von 2 Semestern und die Erstellung der verschiedenen Meilensteine muss von der Gruppe

durchgeführt werden. Auch dies ist eine Aufgabe die nicht leicht zu erledigen ist. Oft müssen die Meilensteine abgeändert werden, daher ist es von vornherein eine gute Planung notwendig, da ein verschieben von Meilensteinen meist eine komplette Änderungen des Projektplans zur Folge. Auch sollte man sich nicht an starre Vorgaben bei der Trennung der Einzelnen Phasen des Projektes halten. Auch wenn bei modernen Vorgehen zur Erstellung einer Software die Verteilung bei 60% / 15% / 25% (Analyse, Spezifikation, Entwurf / Codierung / Test, nach [1]) liegt. Die meisten Mitglieder eines Projektteams im Studienprojekt noch keine Erfahrung in einem Projekt dieser Größe. Daher ist es zu empfehlen zunächst genau das Projekt und die zu Erstellende Software zu analysieren und sich ein Vorgehen zu überlegen. Und anschließend anhand dieses Vorgehens die Meilensteine zu erstellen.

Weiterhin ist es wichtig, dass alle Mitglieder des Projektteams ihre individuellen Aufgaben erfüllen und am Projekt mitwirken. Im Softwarepraktikum ist es oft der Fall, dass 2 Personen das Projekt allein entwickeln und der dritte sich nur an Spezifikation und Handbuch beteiligt. Im Studienprojekt dagegen ist es notwendig das jede Person ihre Aufgaben übernimmt und besonders Projektleiter seine Aufgabe wahrnimmt und gut durchführt.

## 1.2 Probleme die im Studienprojekt auftreten könne und Lösungen für diese

Im Vorheringen Teil wurde das Studienprojekt mit dem Softwarepraktikum verglichen und dabei wurden bereits wichtige Punkte im Bezug auf Organisation und Rollenverteilung erläutert. In diesem Teil werden einige Probleme aufgelistet, die auch bei einer guten Organisation auftreten können (oder werden) und wie man diese verhindern (bzw. möglichst schnell lösen) kann.

Es kam zu Verzögerungen bei der Implementierung, weil zwei Mitglieder des Teams nur für die Qualitätssicherung zuständig waren. Dadurch waren diese beiden Personen nicht komplett ausgelastet und man hätte sie (zumindest teilweise) bei der Implementierung einsetzen können um diese schneller abzuschließen. Eine direkte Lösung für dieses Problem gibt es nicht, hier kommt es eher darauf an, dass das Team und der Projektleiter flexibel sind und auch bereit sind andere Tätigkeiten als ihre Rolle zu erledigen, wenn sie genügend Zeit haben und sie das nicht in ihren anderen Tätigkeiten behindert.

Ein weiteres Problem waren Unklarheiten über die genaue Lösung. Es kam innerhalb der Gruppe zu Komplikationen da es verschiedene Ideen gab und der Projektleiter es nicht geschafft hatte sich entsprechend durchzusetzen. Dadurch gab es erst relativ spät einen echten Ideenfindungsprozess und man einigte sich erst am Ende des ersten Semesters des Studienprojekts auf die genaue Lösung.

Um dieses Problem zu vermeiden ist auf der einen Seite ein guter Projektleiter notwendig, als auch eine gute Kommunikation innerhalb der Gruppe. Es müssen regelmäßige Treffen organisiert werden und die verschiedenen Mitglieder des Teams müssen, zusammenarbeiten und bereit sein Kompromisse einzugehen.

Zu wenig Kommunikation mit dem Kunden war ein weiteres Problem, so führte dies in einem Studienprojekt zur Ablehnung nach der ersten Iteration. Es ist daher wichtig das regelmäßige Treffen mit dem Kunden abgehalten werden. Der Sinn dieser Treffen sollte es sein bei Unklarheiten in Bezug auf bestimmte Features diese noch einmal genau zu besprechen oder Probleme die im Laufe des Projektes auftreten zu diskutieren. Es sollte keinesfalls versucht werden selbstständig eine Lösung zu finden ohne diese vorher mit dem Kunden besprochen zu haben. Auch wenn sich Features nicht umsetzen oder nur teilweise umsetzen lassen, muss gemeinsam mit dem Kunden besprochen werden und wie man dieses Problem lösen kann.

Das fehlen einer klaren Prozessdefinition führte ebenfalls zu Problemen. Es war dadurch im Laufe des Studienprojekts oft unklar in welcher konkreten Phase man sich gerade befindet. Damit dies nicht auftritt, braucht man eine klare Phaseneinteilung und es sollte versucht werden, diese auch einzuhalten. Das heißt zum Beispiel das man erst mit der Spezifikation beginnt wenn die Analyse abgeschlossen ist, oder das man erst mit der Implementierung beginnt wenn der Entwurf abgeschlossen ist, da es bei Überschneidungen früher oder später zu Problemen und Unklarheiten kommt deren Lösung sehr lange dauern kann, bzw. den neu Beginn einer Phase oder Iteration bedeuten kann.

Natürlich kann es auch bei einer guten Phaseneinteilung immer Änderungen geben, aber es sollte versucht werden im Voraus so zu planen, dass es zu keinen großen Änderungen kommt und eventuell auch Zeitpuffer einzureihen. So kann man zum Beispiel zwischen Beginn und Ende einer Phase einen Puffer von einer Woche planen. Sollte es nun in der ersten Phase zu Problemen kommen, kann dieser Puffer dafür genutzt werden das Problem zu lösen. Wenn keine Probleme auftreten, kann hier direkt mit der nächsten Phase begonnen werden.

Das Durchführen der Usability Tests sorgte ebenfalls für Schwierigkeiten. Die Tests wurden durchgeführt, allerdings stellte man am Ende der Tests fest, dass es an Zeit für eine komplette Auswertung fehlte. Daher sollte man abwägen und gut planen wie viel Zeit man für einen Test, seine Auswertung und die anschließende Verwendung der Ergebnisse benötigt. Es führt zu gar nichts, wenn man einen sehr komplexen Test durchführt aber am Ende nur ein Viertel der Ergebnisse wirklich nutzen kann. In diesem Fall wäre es besser gewesen einen weniger komplexen Test durchzuführen und anschließend alle Ergebnisse zu nutzen.

# 2. Softwareinfrastruktur SIMPL im Überblick

Die Softwareinfrastruktur von SIMPL besteht auf der einen Seite aus der Datenbank DB2 Version 9 und der Workflow-Umgebung (Eclipse BPEL Designer und Apache ODE) und auf der anderen Seite aus den Entwicklungstools Subversion, Maven und Hudson.
Die DB2 dient als Datenbank für die relationalen Daten, als auch die XML, bzw. BPEL Daten mit denen im Studienprojekt SIMPL gearbeitet wird. Der Eclipse BPEL Designer und die Apache ODE sind notwendig um mit BPEL zu arbeiten und Prozesse zu erstellen, zu testen und auszuführen.

Die Entwicklungstools Subversion, Maven und Hudson können zur Erleichterung der Zusammenarbeit im Entwicklungsteam und zur Erleichterung der Builds eingesetzt werden und werden im Hauptteil der Ausarbeitung vorgestellt.

# 3. Die Datenbank - IBM DB2

**Die URL der IBM DB2 ist: http://www-01.ibm.com/software/data/db2/9/**

DB2 ist ein relationales Datenbank Management System der Firma IBM.

Seit Version 9 gilt die DB2 allerdings als der erste Hybriddatenserver. Der Grund dafür ist das ab dieser Version sowohl die Verwaltung von relationalen als auch XML-Daten möglich ist.

Bei der Verwaltung der XML-Dokumente ist die Besonderheit das sie direkt in ihrem hierarchischen XML-Format in den Spalten einer Tabelle abgelegt werden und nicht wie bei anderen Datenbanken als sogenanntes „Large Object“ abgespeichert werden. Es ist außerdem möglich die Integrität der XML-Dokumente anhand eines DB2 XML-Schemas sicherzustellen, damit keine invaliden Änderungen an den XML-Dokumenten vorgenommen werden oder aber ungültige XML-Dokumente gespeichert werden.

DB2 eignet sich durch die Unterstützung von pureXML seit Version 9, auch ideal als Datenserver für SOA, da diese meist auf XML- und Webservice Standards aufbauen.

# 4. Die Workflow-Umgebung - Apache ODE und Eclipse BPEL Designer

Um mit BPEL Prozessen zu arbeiten nutzen kann der Eclipse BPEL Designer in Verbindung mit Apache ODE genutzt werden. Mit dem Eclipse BPEL Designer können BPEL Prozesse in einer grafischen Oberfläche erstellt werden. Wenn die Prozesse erstellt wurden ist ihre Ausführung mit einem Apache ODE Server möglich (Apache ODE muss zuvor eingebunden werden).

## 4.1 Apache ODE

**Die URL von Apache ODE ist: http://ode.apache.org**

Apache ODE ist eine BPEL-fähige Workflow-Engine. Mit ihrer Hilfe können Geschäftsprozesse, die im WS-BPEL Standard geschrieben sind in serviceorientierten Architekturen (SOA) ausgeführt werden.

*Features [7]*

Apache ODE unterstützt sowohl den WS-BPEL 2.0 Standard von OASIS, als auch das ältere BPEL4WS 1.1.

Es werden 2 Kommunikations-Schichten bereitgestellt, eine aufbauend auf Axis2 und eine andere die auf JBI (Java Business Integration) aufbaut.

Weiterhin gibt es die Möglichkeit Prozess Variablen auf einer beliebigen Datenbank abzubilden.

Die Apache ODE erlaubt das Hot-deployment von Prozessen, das bedeutet das die Prozesse im laufenden Betrieb installiert und aktualisiert werden können ohne das dabei der Server angehalten oder neu gestartert werden muss.

Eine detaillierte Analyse und Validierung von Prozessen ist sowohl von der Kommandozeile aus, als auch während des Einsatzes möglich.

Außerdem verfügt die Apache ODE über ein Management Interface für Prozesse, Instanzen und Nachrichten. Dadurch ist es jeder Zeit möglich herauszufinden, welche Prozesse gerade gestartet sind oder welche Variablen sie benutzen.

## 4.2 Eclipse BPEL Designer

Der Eclipse BPEL Designer ist ein Plugin für die Eclipse Entwicklungsumgebung. Das Ziel des Eclipse BPEL Projektes war es eine Unterstützung in Eclipse anzubieten die es ermöglicht mit WS-BPEL 2.0 zu arbeiten. Mit dem Eclipse BPEL-Designer ist es möglich, BPEL-Prozesse zu erstellen, zu editieren, auszuführen und zu testen. Die Erstellung von BPEL-Prozessen mit dem Designer ist einfach und ohne Probleme möglich. Getestet werden können die erstellten Prozesse auf einem Apache ODE Server.

*Features [6]*

Der BPEL-Designer verfügt über mehrere leicht verständliche Ansichten in denen BPEL Prozesse grafisch angezeigt und bearbeitet werden können.

Es gibt ein EMF (Elcipse Modeling Framework) Model in welchem die WS-BPEL 2.0 Spezifikation enthalten ist. Dadurch werden Fehler in den erstellten

Prozessen direkt erkannt und es können Warnungen ausgegeben werden. Dies ist von Vorteil, da man die Prozesse nicht erst ausführen muss um eine Rückmeldung über Fehler zu erhalten. Stattdessen wird man direkt während der Bearbeitung auf Fehler hingewiesen.

Außerdem enthalten sind ein erweiterbares Framework, welches es möglich macht BPEL-Prozesse direkt von den Entwicklungswerkzeugen aus in einer BPEL-Engine einzusetzen und auszuführen und ein weiteres Framwork, welches es den Nutzern ermöglicht die Ausführung von Prozessen Schritt für Schritt zu verfolgen.

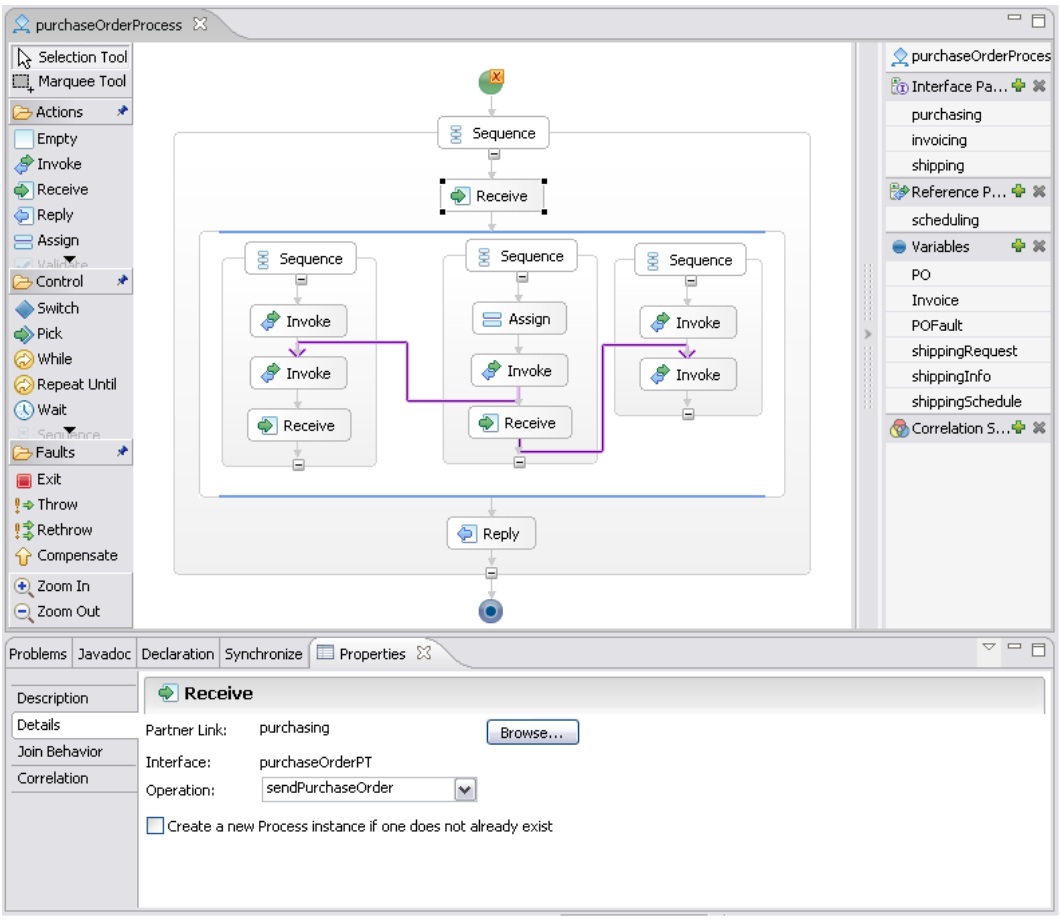


**Abbildung 1.** Eclipse BPEL-Designer Hauptfenster [6]

In Abbildung 1 sieht ist das Hauptfenster des BPEL Designers zu sehen. Hier werden BPEL-Prozesse grafisch dargestellt. Auf der rechten Seite ist eine Liste der Variablen und Partner Links zu sehen. Auf der Linken Seite ist eine Palette zu sehen, mit welcher es möglich ist per Drag-and-Drop neue Bestandteile hinzuzufügen. Das Editieren der einzelnen Bestandteile des Prozesses erfolgt im unteren Teil des Hauptfensters unter Properties.

Beim Anlegen eines neuen BPEL Prozess Files wird gleichzeitig auch eine WSDL Datei erstellt. In dieser Datei sind Angaben zu den Schnittstellen,

Zugangsprotokolle und Details zum Deployment und notwendige Informationen zum Zugriff auf dem Service enthalten.

Das Editieren dieser Datei ist ähnlich dem BPEL-Process Files möglich und wird ebenfalls grafisch unterstützt.

Im Großen und Ganzen ist der Eclipse-BPEL-Designer ein sehr angenehmes Plugin, der einen einfachen Umgang und eine einfache Verarbeitung von WS-BPEL 2.0 Prozessen bietet.

# 5. Entwicklungstools

In diesem Teil werden die drei Entwicklungstools Subversion, Maven und Hudson vorgestellt. Wenn diese Entwicklungstools gemeinsam genutzt werden, hat Hudson eine zentrale Rolle. In Hudson werden Builds geplant und anschließend mit Maven durchgeführt. Dabei ist der Quellcode in einem zentralen Repository (Subversion oder CVS) abgelegt

## 5.1Subversion (SVN)

**Die URL von Subversion ist: http://subversion.tigris.org**

### *5.1.1 Was ist Subversion*

Subversion ist eine Open-Source Software zur Versionskontrolle. Das Herzstück von Subversion bildet das sogenannte Repository. Das Repository ist ähnlich einem normalen Filesystem. Nutzer können auf das Repository zugreifen und können die enthaltenen Daten lesen oder Daten im Repository speichern. Das besondere an einem Repository ist, dass es nicht nur die aktuelle Version der Daten speichert, sondern jede Änderung die an diesen Daten gemacht wurde. Das bedeutet, wenn ein Nutzer auf das Repository zugreift, dann hat er nicht nur die Möglichkeit die aktuellen Daten zu lesen, sondern kann auch auf jede ältere Version der Daten zugreifen.

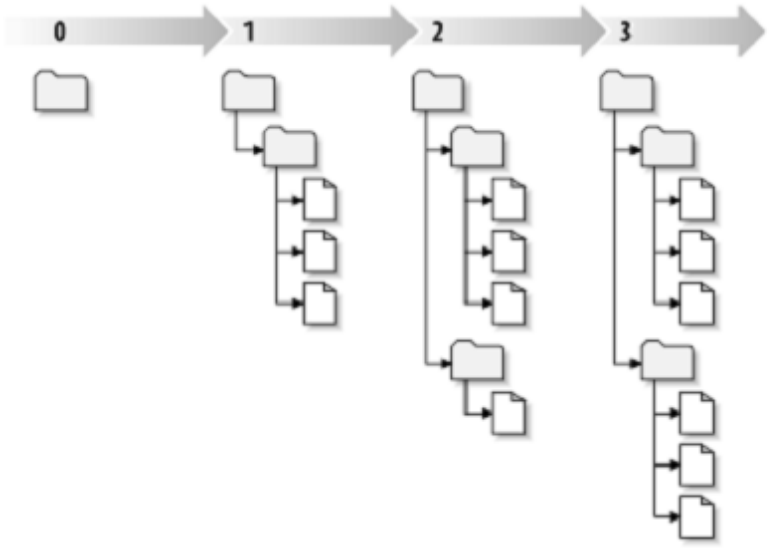


**Abbildung 2.** Repository als Verzeichnisbaum [8, Seite 11]

Man kann sich das Repository als eine Folge von Verzeichnisbäumen vorstellen(siehe Abbildung 2.), beginnend mit dem leeren Repository (Revisionsnummer 0). Wenn nun Daten oder Verzeichnisse hinzugefügt werden, dann "wächst" der Verzeichnisbaum innerhalb des Repositorys und bei jeder Änderung erhöht sich die Revisionsnummer um 1. Das Bedeutet das jede Revisionsnummer einen Verzeichnisbaum, und somit den Zustand des Repositorys nach einer Änderung, repräsentiert.

Wie auch andere Software zur Versionskontrolle ist das Ziel von Subversion es möglich zu machen, dass mehrere Nutzer an denselben Daten arbeiten und sich dabei nicht gegenseitig behindern (Änderungen anderer Nutzer überschreiben). Dies könnte beispielsweise auftreten, wenn zwei Nutzer an einer Datei arbeiten. Wenn nun der erste Nutzer seine Datei speichert und kurze Zeit später der zweite Nutzer seine Version dieser Datei speichert, ist die Version von Nutzer zwei die aktuell im Repository präsente Version. Die Änderungen von Nutzer eins sind nicht verloren, allerdings sind sie auch nicht in der aktuellen Version vorhanden. Es ist klar, dass dies vermieden werden muss.

Um Probleme dieser Art zu verhindern, gibt es zwei mögliche Vorgehen, das Lock-Modify-Unlock Prinzip und das von Subversion umgesetzte Copy-Modify-Merge Prinzip [8, Seite 2 – 6].

**Lock-Modify-Unlock:** Dieses Prinzip ist recht einfach, es bedeutet, dass eine Datei nur von einem Nutzer bearbeitet werden kann. Das Vorgehen hierbei ist, dass ein Nutzer eine Datei "locked" (sperrt) und diese damit nicht mehr von anderen Nutzern bearbeitet werden kann, bis dieser Nutzer die Datei wieder freigibt. Erst wenn dieser Nutzer mit seiner Änderung fertig ist und die Datei freigibt, kann sie von anderen Nutzern verwendet und bearbeitet werden.

Das größte Problem bei diesem Prinzip ist, dass es eine große Einschränkung darstellt. So kann es zu administrativen Problemen kommen, wenn ein Nutzer

eine Datei locked und nach Abschluss seiner Bearbeitung die Datei nicht wieder freigibt. In diesem Fall muss der Administrator des Repositorys benachrichtigt werden und die Datei freigeben. Ein weiteres Problem ist zum Beispiel wenn zwei oder mehrere Nutzer unterschiedliche Teile einer Datei bearbeiten. Theoretisch könnten beide Nutzer gleichzeitig an der Datei arbeiten ohne sich gegenseitig zu behindern und die Änderungen könnten danach einfach in einer neuen Version der Datei zusammengefügt werden. Da die Datei allerdings beim ersten Nutzer gesperrt wird, ist es für andere Nutzer nicht mehr möglich diese Datei zu bearbeiten. Eine Lösung dieser Probleme bietet das Copy-Modify-Merge Prinzip.

**Copy-Modify-Merge:** Bei diesem Prinzip wird bei Zugriff auf dem Rechner jedes Nutzers eine persönliche "Arbeitskopie" der Datei oder des Projektes erstellt. Die Nutzer können nun parallel an den Daten arbeiten und ihre Arbeitskopien bearbeiten. Wenn die Nutzer mit ihrer individuellen Bearbeitung fertig sind, dann werden ihre Änderungen mit der aktuellen Version im Repository zusammengeführt. Das Versionskontrollsystem unterstützt diesen Vorgang, allerdings ist stets ein menschlicher Nutzer notwendig damit es korrekt funktioniert.

Ein Beispiel dazu stellt folgendes Szenario dar:

Zwei Nutzer arbeiten simultan an einer Datei. Nutzer eins speichert seine Änderungen zuerst im Repository ab. Wenn nun der andere Nutzer seine Daten speichern möchte, wird ihn Subversion darüber informieren, dass seine Version der Datei veraltet ist. Mit Subversion ist es nun möglich die Änderungen der Version im Repository mit den persönlichen Änderungen des Nutzers zusammenzuführen. Ein Konflikt tritt hier auf, wenn sich die Änderungen der beiden Nutzer überschneiden. Die Version des Nutzers wird in diesem Fall damit gekennzeichnet das ein Konflikt aufgetreten ist. Es ist für ihn nun möglich beide Änderungen unabhängig voneinander zu sehen und diese manuell zusammenzuführen.

Diese Konflikte können allerdings durch eine Absprache der Nutzer untereinander vermieden werden, in dem im Voraus geklärt wird, welcher Nutzer für welche Änderungen, bzw. für welchen Teil einer Datei verantwortlich ist. Selbst wenn Konflikte auftreten sind sie in der Regel schneller zu lösen, als wenn die Bearbeitung der Datei auf Grundlage des Lock-Modify-Unlock-Prinzipes durchgeführt wurden wäre.

Subversion nutzt standardmäßig das Copy-Modify-Merge Prinzip, allerdings ist es auch möglich Dateien zu locken. Dies ist nützlich wenn mit Dateien gearbeitet wird, deren Änderungen nicht zusammengeführt werden können, wie zum Beispiel Grafikdateien.

**Das Tag- und Branchkonzept:** In Subversion gibt es die Möglichkeit eine separate Entwicklungslinie anzulegen, die unabhängig der Hauptentwicklung (Trunk) des Projektes bearbeitet werden kann. Diese Entwicklungslinie ist als Branch bekannt. Branches werden verwendet genutzt um neue Features zu testen oder einen anderen Teil des Projektes zu entwickeln, ohne dabei die Hauptentwicklung durch Bugs oder Errors zu behindern. Wenn diese neuen Teile

fertig und stabil sind, besteht die Möglichkeit sie mit dem Trunk zusammenzuführen.

Es gibt weiterhin die Möglichkeit eine bestimmte Revision zu markieren (tagging). Damit hat man die Möglichkeit immer auf eine bestimmte Version des Programms zurückzugreifen, zum Beispiel eine stabile Version die allerdings noch nicht alle Features enthält.

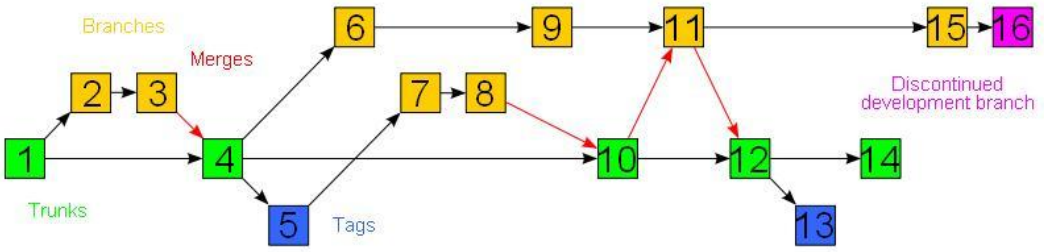


**Abbildung 3.** Verlauf eines Projektes in Subversion [2]

Das Anlegen von Branches und Tags geschieht über das Konzept der "billigen Kopie". Das bedeutet, dass man anstatt eine komplette, zweite Kopie, des Projektverzeichnisses durch Duplizierung der Daten erstellt, nur eine Verknüpfung anlegt. Diese kann im weiteren Verlauf des Projektes genauso weiterverwendet werden wie das Original. In Abbildung 3. sieht man einen typischen Projektverlauf mit mehreren Branches für alternative Entwicklungslinien, Tags und auch das Zusammenführen (Mergen) der Branches mit den Trunks und Tags. Dabei ist Revision 16, die aktuellste Version ("top revision"). Diese wird typischerweise als head bezeichnet.

### *5.1.2 Features von SVN im Vergleich zu CVS*

Subversion wurde als eine bessere Version von CVS entwickelt, daher wurden die meisten Features von CVS in Subversion übernommen.

Einer der Hauptunterschiede zwischen Subversion und CVS liegt in der Kennzeichnung der Inhalte des Repositorys. Während in CVS die Dateien unabhängig von einander versioniert werden, wird in Subversion immer das gesamte Projektarchiv versioniert. Das bedeutet, dass bei jeder Änderung dem gesamten Projektarchiv eine neue Revisionsnummer zugeordnet wird. So ist es einfacher eine exakte Version zu beschreiben. Statt zum Beispiel die Version vom "17.09.2008 16:50:32" (CVS) ist es einfach die "Revision 1348". Die Revisionsnummer einer einzelnen Datei entspricht dabei der Revisionsnummer des Projektarchives, als sie das letzte Mal geändert wurde.

In Subversion ist es möglich Verzeichnisse zu versionieren. In CVS wird nur die History der individuellen Dateien gespeichert, in Subversion dagegen werden auch die Änderungen an den Verzeichnisbäumen gespeichert. Das heißt das sowohl Dateien als auch Verzeichnisse versioniert werden.

CVS hat einige Beschränkungen in der Versionierung von Daten. So sind Operationen wie Kopieren und Umbenennen von Dateien in CVS nicht unterstützt. Dadurch kommt es zum Beispiel zu Problemen wenn man eine Datei mit einer neuen Datei überschreibt die denselben Namen hat, denn die neue Datei erbt dann automatisch die History der alten, überschriebenen Datei. Dies ist besonders dann ein Problem wenn beide Dateien außer der Bezeichnung nichts gemeinsam haben. In Subversion dagegen ist es ohne Probleme möglich Dateien, sowie Verzeichnisse hinzuzufügen, zu löschen, zu kopieren und umzubenennen. Außerdem beginnt jede neu hinzugefügte Datei (oder eine Überschreibung einer alten Datei) eine neue, eigene History.

Die Übertragung der Änderungen der Daten in Subversion geschieht atomar. Das bedeutet, eine Änderung, auch wenn sie mehrere Dateien umfasst, wird entweder komplett oder gar nicht im Repository gespeichert. Dies hat den Vorteil, dass es nicht durch Verbindungsabbrüche oder gleichzeitige Zugriffe zu inkonsistenten Zuständen des Repository kommen kann.

Subversion bietet im Gegensatz zu CVS einen verbesserten Umgang mit Binärdaten (z.B. Grafiken) an. Sie werden auf dieselbe Weise wie normale Textdateien behandelt und es wird nur die Differenz zwischen zwei geänderten Versionen abgespeichert. In CVS dagegen mussten verschiedene Versionen von Dateien dieses Typs immer komplett gespeichert werden.

Eine weitere Neuerung von Subversion gegenüber CVS ist es, dass beim Erstellen einer lokalen Kopie (Arbeitskopie), beim Aktualisieren der lokalen Daten und beim Übertragen der Änderungen in einem anderen Verzeichnis eine Kopie dieser Datei gespeichert wird.

Diese Kopie enthält zwei wichtige Informationen über die Arbeitskopie, einmal zu welcher gegebenen Revision die Arbeitskopie erstellt wurde und den Zeitpunkt zu dem die Arbeitskopie zu letzte aktualisiert wurde. Mit diesen beiden Informationen ist es für Subversion möglich zu bestimmen in welchem Zustand sich die Arbeitskopie befindet:

- **Unverändert und aktuell**

Die Daten innerhalb der Arbeitskopie sind unverändert und auch die Daten im Repository wurden nicht verändert. Eine Übertragung der Daten oder ein Update der Daten hat keine Änderung zur Folge.

- **Lokal Verändert und aktuell**

Die Daten wurden innerhalb der Arbeitskopie verändert und es gab bisher keine Änderungen im Repository. Eine Übertragung der Änderungen aktualisiert die Daten im Repository, ein Update dagegen hat keine Änderungen an den lokalen Daten zur Folge (auch keine Wiederherstellung des Zustandes vor den Änderungen).

- **Unverändert und veraltet**

Die Daten in der Arbeitskopie sind unverändert, allerdings gab es seit dem letzen Update Änderungen an den Daten im Repository. Eine Übertragung der lokalen Daten hat keine Änderungen zur Folge, ein Update dagegen bringt die Daten der Arbeitskopie auf den neusten Stand.

- **Lokal verändert und veraltet**

Die Daten wurden sowohl in der Arbeitskopie als auch im Repository verändert. Eine Übertragung der Änderungen würde mit einem out-of-date-error abgebrochen werden. Die Daten der Arbeitskopie müssen zuerst aktualisiert werden. Bei diesem Update wird Subversion versuchen die Änderungen im Repository mit den lokalen Daten zusammenzuführen, ist dies nicht automatisch möglich, muss der Nutzer dies manuell durchführen.

Obwohl diese Art der Speicherung den Speicherplatz verdoppelt, hat es doch einige Vorteile.

So kann das anzeigen der lokalen Änderungen ohne Zugriff auf das Subversion Repository geschehen. Dadurch muss Subversion beim Übertragen der Änderungen nur die geänderten Teile einer Datei übertragen. In CVS dagegen musste stets die komplette Datei übertragen werden, da die Änderungen erst im Repository berechnet werden. Außerdem ist es möglich jederzeit die Änderungen an einer Datei gegenüber ihrer Basisversion zu ermitteln ohne dabei auf das Repository zugreifen zu müssen.

In Subversion ist das anlegen von Branches und Tags viel effizienter als in CVS. Während es in CVS eine fest vorgegebene Semantik für diese Operationen gab, wird in Subversion nur eine sogenannte “billige Kopie” angelegt, die entweder Branch- oder Tag-Charakter hat. Das Anlegen dieser Kopie ist einer sehr kurzen konstanten Zeiteinheit möglich.

### *4.2.2 Subversion im Studienprojekt SIMPL*

Im Studienprojekt SIMPL kann Subversion als Versionsverwaltungsprogramm genutzt werden. Ein Versionsverwaltungsprogramm ist sinnvoll, da wir in einer Gruppe von 8 Leuten an einer Software arbeiten und mit Subversion ist es möglich, dass die Gruppenmitglieder unabhängig von einander an Dateien und Modulen arbeiten können.

Durch Das Prinzip des Branching ist es möglich alternative Entwicklungszweige zu erstellen, das macht es möglich, dass die verschiedenen Entwickler an verschiedenen Modulen des Studienprojektes arbeiten können, ohne sich gegenseitig zu behindern. Und diese mit Unterstützung durch Subversion in einer neuen Version zusammenzuführen.

Durch die Möglichkeit auf ältere Versionen der Daten zu zugreifen ist es möglich zu einer stabilen Version des Projektes zurückzukehren falls es zu

schwerwiegenden Fehlern kommt und auf diese Weise zu analysieren welche Änderungen zu dem Problem geführt haben.

Dadurch, dass das Repository extern ist, können die Entwickler an ihren eigenen Arbeitskopien arbeite und ihre Änderungen zu nächst testen bevor sie sie veröffentlichen und den anderen Mitgliedern zur Verfügung stellen.

## 5.3 Maven

**Die URL von Maven ist: http://maven.apache.org/**

### *5.3.1 Was ist Maven*

Maven ist ein Projetk-Management-Tool der Apache Software Foundation. Es enthält ein umfassendes Projektmodell und stellt eine Reihe von Normen bereit. Es verfügt über einen definierten Projekt-Lebenszyklus, ein Äbhängigkeits-Management-System sowie eine Logik welche die Ausführung von Plugin-Goals in den definierten Phasen eines Lebenszyklus ermöglicht.

Ein Plugin-Goal ist eine Aufgabe innerhalb der Maven Terminologe. Diese Goals sind typischerweise durch die verschiedenen installierten Plugins verfügbar. Es ist allerdings nicht notwendig stets alle Goals auszuführen, da es einige Standard Goals gibt, die von anderen aufgerufen werden.

Diese Art der Ausführung kommt daher, dass Maven auf einer so genannten Plugin-Architektur basiert. Das bedeutet das man verschiedenen Plugins (z.B. compile, test, deploy, package) auf das Projekt anwenden. Die Installation dieser Plugins geschieht mit dem einem einfachen Aufruf der Form:

**maven – DartifactId=ARTIFACTID -DgroupId=GROUPID -Dversion=VERSION plugin:download**

Dabei müssen ARTIFACTID, GROUPID und VERSION einfach mit den Details des jeweilig zu installierenden Plugins ersetzt werden.

Ein weiterer Vorteil von Maven ist es, dass es eine gemeinsame Build-Schnittstelle für Projekte darstellt. Früher war es oft sehr aufwendig ein Projekt aus Subversion auszuchecken und aus den Quellen zu erstellen. Meist nahm dies einige Zeit in Anspruch. Mit Maven dagegen, ist dies sehr einfach, man macht ein Check-Out und führt anschließend mvn install in der Kommandozeile aus.

Mit Maven sollen Software-Entwickler von dem Anlegen eines Projektes über das Kompilieren, Test und anschließendes Packen bis hin zum Verteilen der Software auf den Anwendungsrechnern so unterstützt werden, dass möglichst viele Schritte automatisiert werden können.

Wenn man sich beim Build eines Projekts an die von Maven vorgegebenen Standards hält, braucht man für die meisten Aufgaben des Build-Managements nur sehr wenige Konfigurationseinstellungen.

Weitere nützliche Funktionalitäten von Maven sind unter anderem das Anfertigen von Berichten / Auswertungen (Reports), Websites zu generieren und Kontakt zwischen den Mitgliedern des Teams zu ermöglichen.

Es gibt Maven-Archetpyen, mit diesen kann man ein Grundgerüst für unterschiedliche Softwareprojekte erstellen, die der Standardstruktur von Maven entsprechen. So gibt es zum Beispiel Archetypen für Hibernate oder für Portlet Anwendungen.

Um ein einfaches Projekt mit Maven zu beginnen setzt man das Archetype-Plugin von der Kommandozeile aus ein mit folgendem Befehl ein:

**mvn archetype:create**

**-DgroupId=[Die ID der Projektgruppe]**

**-DartifactId=[die Projekt Artifakt ID]**

Es gibt noch eine Reih anderer Befehle mit denen man das Projekt noch weiter spezifizieren kann.

Die Verzeichnisstruktur eines mit Maven erstellten Projektes hat die Struktur die in Abbildung 4. zu sehen ist.

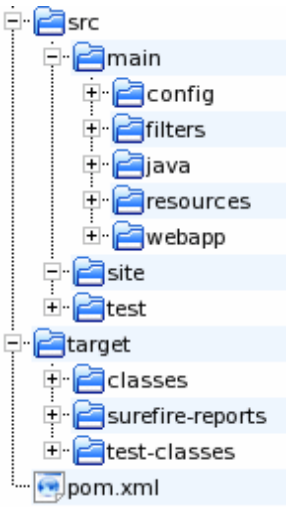


**Abbildung 4.**Verzeichnisstruktur eines Maven Projektes [11]

Der src (Source) Ordner hat eine Reihe von Unterordnern die verschiedene Zwecke habe:

Die Source Verzeichnisse enthalten die den Quellcode und die Ressourcen des Projektes, sowie die Unit Tests und deren Ressourcen. Das Verzeichnis target ist das Build Verzeichnis in dem die Erstellten Artefakte des Builds abgespeichert werden.

Wenn man ein Maven2 Projekt erstellt hat, werden alle Informationen über das Projekt ein einer XML-Datei abgespeichert, der pom.xml (Project Object Model). Diese Datei enthält alle Informationen zu einem Softwareprojekt und folgt einem Standardisierten Format.

Sie setzt sich zusammen aus vier Kategorien für Einstellungen und Beschreibungen (siehe Abbildung 5):

- **Die Hauptinformationen zum Projekt**

Diese enthalten den Namen des Projekts, die URL des Projekts, eine Liste der Entwickler des Projekts sowie die Lizenz des Projekts usw. (siehe Abbildung 6.).

- **Die Build-Einstellungen**

In diesem Teil werden kann man das Verhalten des Standard Maven Builds anpassen. So kann man bestimmen wo die Daten des Projekts abgespeichert sind, oder weitere Plugin Goals zum Lebenszyklus des Projekts hinzufügen.

- **Die Build-Umgebung**

Hier sind Profile für die Nutzung auf unterschiedlichen Build-Umgebungen enthalten. So kann es während der Entwicklung eines Projektes sinnvoll sein dieses für den Einsatz auf einem Deployment Server zu laden. Dieser Teil der pom.xml passt die Build Einstellungen an die unterschiedlichen Umgebungen an.

- **POM relationship**

Der letzte Teil der pom.xml enthält Informationen zu anderen Projekten von denen es abhängt, enthält Einstellungen die es von anderen POM's erbt oder beschreibt enthaltene Submodule

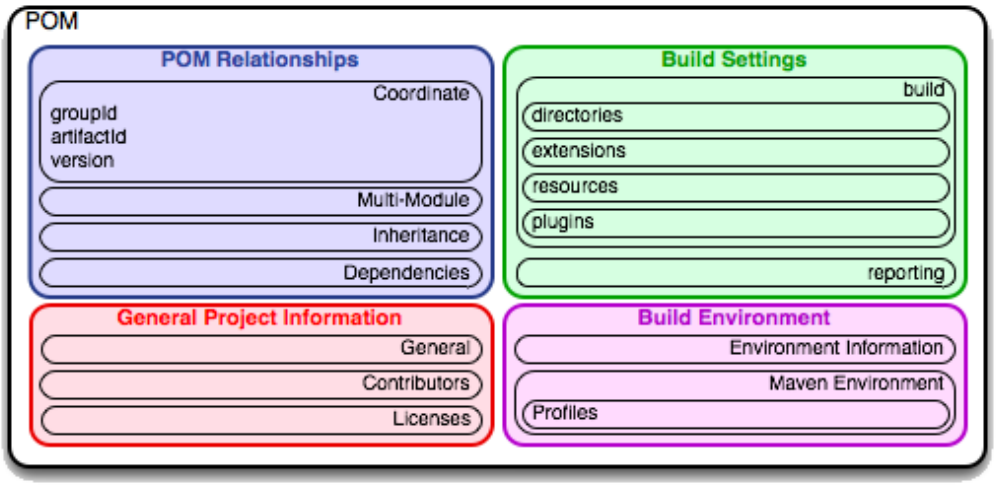


**Abbildung 5.** Die verschiedenen Elemente der pom.xml [4, Kapitel 9.2]

Eine einfache Beispiel pom.xml [4, Kapitel 9.2.2] ist hier zu sehen:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>
```

Ein Maven 2 Build Lebenszyklus setzt sich aus verschiedenen Phasen zusammen. Wenn man nun zum Beispiel mvn deploy eingibt, durchläuft maven folgenden Standardlebenszyklus:

- validate: Validierung ob das Projekt korrekt ist und alle notwendigen Informationen verfügbar sind
- compile: das Kompilieren des Quellcodes des Projekts
- test: durchführen der Unit Tests
- package: Packen des kompilierten Codes in ein JAR oder WAR Archiv
- integration-test: ausführen und anwenden der gepackten Daten, wenn nötig in eine Umgebung in der der Integrationstest durchgeführt werden kann
- verify: Überprüfung ob das Archiv gültig ist und die Qualitätskriterien erfüllt
- install: Installation des Archivs im lokalen Repository um es anderen Projekten zur Verfügung zu stellen
- deploy: wird in einer Integrations- oder Release Umgebung durchgeführt, kopiert die endgültige Version des Archivs in das online Repository um es anderen Entwicklern und Projekten zur Verfügung zu stellen

Werden zum Beispiel beim Build des Projektes bestimmte Abhängigkeiten zu anderen Projekten festgestellt, dann wird zunächst ermittelt ob diese bereits im lokalen Maven-Repository enthalten sind. Ist dies der Fall, werden diese beim kompilieren direkt verwendet. Ist dies nicht der Fall, versucht Maven sich zum Maven-Repository im Internet zu verbinden und von dort die Daten in das lokale Repository zu kopieren um sie verwenden zu können.

Fast alle Vorgaben die Maven macht können geändert werden, die Struktur der pom.xml ist allerdings fest und kann nicht verändert werden.

Für die gängigsten Entwicklungsumgebungen, so auch für Eclipse, gibt es Plugins mit denen es möglich ist Maven direkt aus der Entwicklungsumgebung heraus zu bedienen.

### *5.3.2 Features von Maven [3]*

Einfachstes Erstellen eines Projekts was den „best practice“ (bestes Verfahren / Erfolgsprinzipien) Richtlinien folgt innerhalb von kürzester Zeit.

Maven hat eine Abhängigkeitsverwaltung die automatische Updates und die Auflösung von Abhängigkeiten beinhaltet. Das bedeutet das jedes Projekte mit einer einzigartigen Gruppenkennung (groupID), Artefaktkennung (artefactID) und Version (version) identifiziert wird.

Es ist einfach und ohne Probleme möglich an mehreren Projekten zur selben Zeit zu arbeiten.

Es gibt ein großes und stetig wachsendes online Repository durch das es möglich ist Bibliotheken und Metadaten direkt zu benutzen.

Maven ist erweiterbar und es ist sehr einfach Plugins in Java oder Skriptsprachen zu schreiben.

Durch die Plugin-Architektur und das einfach Installieren der Plugins ist sofortiger Zugang zu neuen Features mit wenigen oder keinen Einstellungen möglich.

Es ist in den meisten Fällen möglich Projekte in vordefinierte Ausgabe Typen, zum Beispiel JAR oder WAR, zu packen ohne dabei mit Skriptsprachen zu arbeiten (in den meisten Fällen).

Maven ist zudem in der Lage Web-Seiten oder PDFs zur Dokumentation zu erstellen, zum Beispiel Berichte über den Fortschritt der Entwicklung.

Mit Maven gibt es die Möglichkeit der Tool Portabilität und Integration. Vor Maven gab es für jede Entwicklungsumgebung eine spezifische Art und Weise auf welche ihre Projektinformationen abgelegt wurden. Das entspricht im Prinzip einer benutzerdefinierten pom.xml. Maven dagegen standardisiert diese Beschreibung. Während jede Entwicklungsumgebung weiterhin ihr eigenes

Datenablagesystem unterhalten kann, lässt sich die pom.xml nun leicht aus diesen Modellen heraus generieren.

### *5.3.3 Vergleich Maven – Ant*

Ant ist ein Build-System mit Targets und Abhängigkeiten. Jedes Target besteht aus einer Reihe von Anweisungen die in XML beschrieben werden. Wenn man Ant zur Erstellung von Projekten nutzt, muss man Ant mit einer speziellen und spezifischen Anweisung für die Zusammenstellung und dem Packaging des Projekts aufrufen. Man muss sehr genau definieren was man von Ant erwartet. Es muss angegeben werden wo die Quell- und Zieldaten abgelegt werden und man muss angeben aus den resultierenden Daten ein JAR- oder WAR-Archiv zu erstellen usw. Bei Maven dagegen reicht es aus eine einfache pom.xml zu generieren, den Quellcode und die Ressourcen in die von Maven vorgegebenen Verzeichnisse ablegen und mvn install von der Befehlszeile aus eingeben. Der Aufruf von mvn install wird die Ressourcen sowie die Quelldateien kompilieren, bestehende Unit-Tests durchführen, JAR-Archive erstellen und in einem lokalen Repository anderen Projekten zur Verfügung stellen.

Hier zu sehen ist ein Ant Skript, welches dasselbe tut wie die unter „Was ist Maven“ angegebene pom.xml:

```
<project name="my-project" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist"  location="target"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-
${DSTAMP}.jar file -->
```

```
    <jar              jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

### *5.3.4 Zusammenfassung Maven*

Maven arbeitet nach Konventionen. Wenn man diese beachtet, weiß Maven wo sich der Quellcode befindet. Maven ist deklarativ, alles was man tun muss ist eine pom.xml zu erzeugen sowie die Quelldateien im Standardverzeichnis abzulegen, Maven kümmert sich um den Rest.

Maven verfügt über einen vorgegebenen Lebenszyklus der mit mvn deploy (oder einem vorherigen Schritt) gestartet wird. Maven arbeitet dann eine Abfolge von Schritten ab bis das Ende des Lebenszyklus erreicht ist. Dabei führt Maven eine Reihe von Standard-Goals aus, diese erledigen dann die Arbeiten wie z.B. das kompilieren oder das Erstellen des JAR-Archives.

## 5.4 Hudson

**Die URL von Hudson ist: https://hudson.dev.java.net/**

### *5.4.1 Was ist Hudson*

Hudson ist ein einfach zu nutzendes, webbasiertes System zur kontinuierlichen Integration von Softwareprojekten. Es macht es einfach für Entwickler Änderungen in das System einzufügen und für Nutzer einen neuen Build zu erhalten. Durch das Prinzip der kontinuierlichen Integration wird die Produktivität in Softwareprojekten erhöht.

Unter Kontinuierlicher Integration versteht man das regelmäßige Neubilden und Testen einer Anwendung. Das bedeutet, dass jeder Entwickler frühzeitig und oft Änderungen in die Versionsverwaltung eincheckt und so zum Beispiel große Änderungen inkrementell durch funktionsfähige kleinere Änderungen einbringt. Normalerweise tägliches Einchecken der Änderungen. Wenn ein Entwickler

Änderungen in die Versionsverwaltung eincheckt, wird das Gesamtsystem neu gebaut und automatisch getestet und dem Entwickler so schnell wie möglich Feedback gegeben um so auf mögliche Integrationsprobleme der Änderung hinzuweisen.

Das Prinzip der kontinuierlichen Integration hilft Entwicklern folgende Dinge einfacher zu realisieren:

- Automatisierung des Software Builds
- Automatische Verifikation des Builds (Überprüfung das der Build von neuem Code nicht beschädigt wird)
- kontinuierliches und automatisches Testen des Builds
- Automatisierung aller Maßnahmen die nach dem Build folgen

Um einen KI Server aufzusetzen benötigt man mindestens ein dauerhaft erreichbares Quellcode-Repository, ein Build Tool, sowie eine Folge von Tests mit denen die erstellten Artefakte getestet werden können.

Im Falle des Studienprojekts SIMPL nutzen wir ein Subversion Repository und setzen Maven 2 als Build Tool ein. Alles weitere, wie das Planen von Builds usw. wird direkt von Hudson übernommen.

Die Installation von Hudson ist sehr einfach, man muss nur die Datei Hudson.war herunterladen und diese anschließend mit dem Befehl java-jar hudson.war ausführen. Wenn das Programm gestartet ist, ist es möglich dieses als einen Windows Dienst einzurichten.

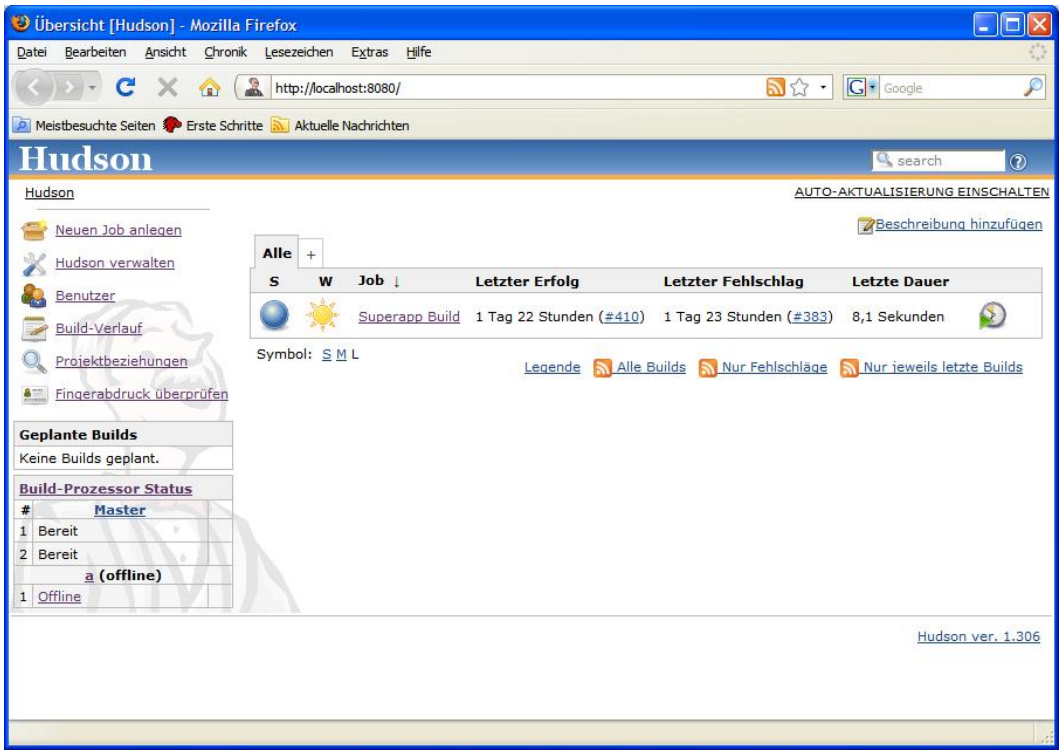


**Abbildung 6.** Hudson Webinterface

Wenn Hudson installiert wurde ist der Zugriff auf das Hudson Webinterface einfach über http://localhost:8080 möglich (siehe Abbildung 6).

Dies ist das Hauptfenster von Hudson in dem neue Benutzer und Jobs verwaltet und erstellt werden können. Außerdem gibt es hier eine Übersicht über alle aktuellen Jobs, in denen gezeigt wird wann ein Build zu letzte erfolgreich war, zuletzt fehlgeschlagen ist, die Dauer des Tests und die Ergebnisse des letzten Builds und seiner Tests.

Wenn man einen neuen Job unter Hudson anlegt, gibt es eine Auswahl von verschiedenen Jobs die angelegt werden können, zum Beispiel ein „Free-Style“-Softwareprojekt, oder ein Maven2 Projekt. Nach Auswahl der Art des Jobs kommt man zu den Einstellungen des Projekts. Hier kann man Software zur Versionskontrolle wie Subversion oder CVN einbinden, das Build Tool und Build Verfahren auswählen (bei einem Maven 2 Projekt, natürlich Maven 2) und weitere Einstellungen wie zeitgesteuerte Builds aktivieren (zum Beispiel automatischer Start eines neuen Builds alle 30 Minuten). Weiterhin gibt es noch eine Reihe von Aktivitäten die Hudson nach dem Build eines Projektes durchführen kann. So kann Hudson die Javadocs und die JUnit-Testergebnisse veröffentlichen, die während des Builds erstellten Artefakte archivieren und E-Mail Benachrichtigungen verschicken ob der Build erfolgreich war oder fehlgeschlagen ist.

### *5.4.2 Features von Hudson [5}*

Hudson hat eine einfach Konfiguration die komplett über eine Web-GUI läuft, es ist daher nicht notwendige die Einstellungen direkt an XML Dateien zu erledigen.

Es ist möglich, anhand von CVS / Subversion eine Liste der Änderungen an dem Build zu erstellen.

Permanente Links zu den URLs „last build“ / „latest succesfull build“ werden erstellt. Damit ist es möglich diese von anderen Rechnern aus direkt aufzurufen.

Es gibt weiterhin die Möglichkeit Meldungen über fehlerhafte oder erfolgreiche Builds über RSS, Email oder Instant Messenger in Echtzeit bekannt zu geben.

JUnit Testberichte können zusammengefasst und tabellarisch dargestellt werden, mit History Informationen. Damit kann man zum Beispiel erkennen wann der erste Test fehlgeschlagen ist und es ist weiterhin möglich den Verlauf der Tests (zum Beispiel von erfolgreich zu fehlgeschlagen) in einem Diagramm darzustellen.

Hudson beobachtet, welcher Build welche Jars erzeugt hat und welcher Build welche Jars verwendet. Das ist ideal um die verschiedenen Abhängigkeiten in einem Projekt zu verfolgen.

Hudson verfügt außerdem über einen Plugin Support und kann durch verschiedene plugins erweitert werden, außerdem ist es einfach möglich neue Tools für Hudson zu schreiben.

Hudson unterstützt den „Master / Slave“ Betrieb. Dies dient dazu die Auslastung die für den Build von Projekten notwendig ist auf mehrere sogenannte „Slave“ Knoten aufzuteilen, oder aber um das Projekt in verschiedenen Umgebungen zu erstellen oder zu testen.

Wenn man den Master / Slave Betrieb nutzt ist die Rolle des Masters immer noch dieselbe. Er ist weiterhin für die http requests verantwortlich und ist immer noch in der Lage selbst Builds auszuführen. Slaves sind Computer die dafür aufgesetzt sind Builds für einen Master durchzuführen. Dazu führt Maven ein Programm was „slave agent“ genannt wird auf diesen Slave-Computern aus. Um diese zu testen muss auf dem Computer der als Slave genutzt werden soll einfach die entsprechende Hudson Seite aufgerufen werden und dort der Slave dienst gestartet werden.

Wenn ein Slave bei dem Master registriert hat, beginnt der Master damit die Daten auf den Slaven zu laden. Die exakte Art und weise der Übertragung hängt von den Einstellungen des jeweiligen Projektes ab. Während einige Projekte stets einen bestimmten Slave nutzen, kann es sein das andere sich frei zwischen den Slaves bewegen. Für Leute die die Hudson Website besuchen ist es immer noch möglich das Javadoc durchzusehen, die Testresultate zu sehen, die Resultate des Builds von dem Master herunterzuladen ohne zu merken das der Build auf einem Slave stattgefunden hat.

# 6 Fazit

Diese Ausarbeitung hat zunächst grundlegende Unterschiede zwischen dem Softwarepraktikum im Rahmen des Vordiploms und dem Studienprojekten im Hauptdiplom erläutert und verschiedene Probleme aufgezeigt die auftreten können. Dies ermöglicht es bereits im Voraus die Planung und Durchführung des Studienprojektes dahingehend zu organisieren, dass diese Probleme vermieden werden können.
Im Hauptteil wurden verschiedene Entwicklungstools und die Möglichkeiten ihrer Einsetzung erläutert. Dies ermöglicht es für das Studienprojekt SIMPL das bereits zu Beginn des Projektes mit einer einheitlichen Entwicklungsumgebung gearbeitet werden kann. Dies ist ein großer Vorteil den dadurch können Integrationsprobleme die durch uneinheitliche Entwicklungsumgebungen entstehen vermieden werden.
Eine Einarbeitung in dieses Thema zeigte weiterhin wie in der Praxis Softwareprojekte erstellt werden und welche Bedeutung eine gute Infrastruktur hat und welche Vorteile man durch diese hat, im Gegensatz zur einfachen Programmierung „nur“ mit Eclipse.

# Literatur


1. Ludewig, J., Lichter, H.: Software Engineering
2. Wikipedia die freie Enzyklopädie: http://www.wikipedia.de
3. Maven – Welcome to Maven: http://maven.apache.org/
4. Maven Book: The Definitive Guide: http://www.sonatype.com/books/maven-book/reference
5. Hudson: an extensible continous integration engine: https://hudson.dev.java.net/
6. BPEL Project home: http://www.eclipse.org/bpel
7. Apache ODE – Index: http://ode.apache.org
8. Version Control with Subversion: http://svnbook.red-bean.com
9. DB2 Universal Database: http://publib.boulder.ibm.com/infocenter/db2luw/v8//index.jsp
10. TortoiseSVN Documentation: http://tortoisesvn.net/docs/release/TortoiseSVN_de/index.html
11. An introduction to Maven 2: http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html