IPP-Report

Max-Planck-Institut für Plasmaphysik

Frank Engel

Verteilung von Experiment-Software für Wendelstein 7-X

IPP 13/18
Dezember, 2010

fachhochschule
stralsund
university of applied sciences

fachbereich wirtschaft
school of business studies

Bachelor-Thesis
zur Erlangung
des akademischen Grades
„Bachelor of Business Informatics“
im Studiengang Business Informatics

Verteilung von Experiment-Software für Wendelstein 7-X

Erstgutachterin: Prof. Dr. Petra Scheffler
Zweitgutachter: Dr. Georg Kühner

Vorgelegt von: Frank Engel
Holzhausen 1
18435 Stralsund
Matrikelnummer: 8678

Vorgelegt am: 25.11.2009

Inhaltsverzeichnis

1 Einleitung 3

1.1 Motivation 4

1.2 Max-Planck-Institut für Plasmaphysik 4

2 Deployment im Softwareentwicklungsprozess 5

2.1 Die Abnahmephase 6

2.2 Die Einführungsphase 7

3 Anforderungsanalyse 8

3.1 Ist-Zustandsbeschreibung 8

3.2 Anforderungen 10

3.2.1 Zusammenfassung der Anforderungsanalyse 11

4 Umsetzung und Lösungsansätze 12

4.1 Welches Build-Tool soll verwendet werden? 12

4.1.1 Maven versus Ant 13

4.1.1.1 Entscheidung für Maven 14

4.1.1.2 Installation von Maven 15

4.1.1.3 Umstellung auf Maven 15

4.2 Stabile Produktivumgebung 16

4.3 Wiederherstellbarkeit von Produkten 18

4.4 Build-Prozesse 20

4.4.1 Phasen des Build-Prozesses 23

4.4.2 Integrations-Build 24

4.4.3 Modul-Release-Build 26

4.4.4 Produkt-Release-Build 28

4.5 Paketierung und Auslieferung 31

4.6 Versionsverwaltung 32

4.6.1 Interne und externen Bibliotheken 33

4.7 Benutzerfreundlichkeit 33

4.8 Probleme mit Maven 34

5 Stand der Entwicklung 37

6 Ausblick 39

7 Verzeichnisse ... 40
7.1 Quellenverzeichnis ... 40
7.2 Abbildungsverzeichnis ... 41
7.3 Tabellenverzeichnis ... 41
7.4 Glossar ... 42
7.4.1 Begriffe und Definitionen ... 42
7.4.2 Abkürzungen ... 43
8 Anhang ... 44
8.1 Ausschnitt des SVN-Repositorys zu DbUtil ... 44
8.2 Privates Maven-Repository für Build-Produkte ... 45
8.3 Umgebungsvariablen für Maven ... 46
8.4 Einbindung externer Bibliotheken in das Maven-Repository ... 47
8.5 Abhängigkeiten in der Eclipse-IDE ... 47
8.6 Repositorystruktur der W7X-CoDaC-Gruppe ... 48
8.7 Vorschlag für Änderungen an der Repositorystruktur ... 49

1 Einleitung

Ziel dieser Bachelorarbeit ist die Qualitätssicherung und -verbesserung der Prozesse zur automatischen Auslieferung von Software in einem laufenden Java-Softwareentwicklungs-projekt am Max Planck Institut für Plasmaphysik in Greifswald.
Den Leser dieser Arbeit erwartet Anfangs eine theoretische Einführung in die Thematik der Software-Auslieferung (Deployment), dabei geht es um die automatisierte Generierung von Software aus Quelldateien mit einer anschließender Auslieferung an die Zielplattformen oder -umgebungen. Anschließend folgt eine Analyse der Anforderungen als Grundlage für die praktische Arbeit und deren Beschreibung, welche auf den gewonnenen Erkenntnissen bei der Umsetzung für das Forschungsprojekt Wendelstein 7-X beruht. Daraufhin folgen eine zusammenfassende Beschreibung der erreichten Anforderungen und ein Ausblick auf Verbesserungsmöglichkeiten der Software-Auslieferungs-Prozesse. Zur Klärung verschiedener Begrifflichkeiten und Abkürzungen empfiehlt es sich beim Lesen der Arbeit den am Ende eingefügten Glossar zu verwenden.

Der praktische Teil dieser Arbeit ist sehr erfolgreich verlaufen und wird am Max-Planck Institut für Plasmaphysik in Greifswald eingesetzt, von daher kann diese Arbeit als Grundlage für ähnliche Projekte, welche sich mit der Softwareauslieferung beschäftigen, dienlich sein. Denkbar ist dies für Softwareentwickler und Projektleiter, die planen eine automatisierte Auslieferung zu verwenden und sich zuvor ein Bild verschaffen wollen, ob das in dieser Arbeit verwendete Build-Tool namens Maven für ihre Anforderungen an einen Deployment-Prozess in Java-Projekten geeignet ist.

1.1 Motivation

Die Motivation für dieses Projekt entstand, als eine seit längerer Zeit in Entwicklung befindliche Software zum Produktiveinsatz kam. Dadurch stiegen die Qualitätsanforderungen bei der Softwareentwicklung für das Forschungsprojekt Wendelstein 7-X deutlich an. Die Bestrebung ist mit den gegebenen Mitteln und Ressourcen, wie Personal, Hardware und Arbeitsstundenaufwand, einen möglichst hohen Grad der Kontrolle über laufende Projekte und deren Softwarequalität[1] zu erzielen. Daraus entstand die Frage, welche Software-Werkzeuge für einen automatischen Build- und Auslieferungs-Prozess geeignet sind, um somit Arbeitsaufwand einzusparen und die Qualität zu steigern.

1.2 Max-Planck-Institut für Plasmaphysik

Das Max-Planck-Institut für Plasmaphysik (IPP), mit Hauptsitz in Garching und einem Teilinstitut in Greifswald, erforscht die physikalischen Grundlagen der Kernfusion nach dem Prinzip des magnetischen Einschlusses. Mit der Erforschung wird die Grundlage für eine kommerzielle Nutzung in Fusionskraftwerken gelegt.

Das Greifswalder Institut ist Arbeitsplatz für mehr als 550 Mitarbeiter, die sich mit der Planung, Konstruktion und Montage für die Fusionsanlage vom Typ Stellerator namens „Wendelstein 7-X“ (kurz W7X) befassen. Das Großprojekt wurde seit 1980 geplant, der Institutskomplex entstand in den Jahren von 1997 bis April 2000 (vgl. IPP 09). Die vorrausichtliche Fertigstellung des W7X-Experimentaufbaus ist laut Planung im Jahre 2014 zu erwarten.
Experimentvorbereitung, Experimentdurchführung und Analyse der wissenschaftlichen Messdaten sollen softwaregestützt durchgeführt werden. Die dazu notwendige umfangreiche Software wird derzeit am IPP in Greifswald von der W7X-CoDaC-Gruppe[2] entwickelt.

---

[1] gemäß ISO/IEC 12207

[2] Die W7X-CoDaC-Gruppe (Wendelstein 7-X Control, Data Acquisition and Communication Gruppe) tritt im Rahmen der vorliegenden Arbeit als Auftraggeber auf.

2 Deployment im Softwareentwicklungsprozess

Bevor die praktische Beschreibung und die Lösungsansätze in Bezug auf die Arbeit am IPP dokumentiert werden, kommt zuallererst eine thematisch Einordnung des Deployments in den Gesamtablauf der Softwareentwicklung. Als Grundlage für die Ausführungen zum Softwareentwicklungsprozess dienten die Lehrveranstaltungen zum Thema Softwaretechnik an der Fachhochschule Stralsund und das „Lehrbuch der Software-Technik“ von Helmut Balzert.

Der Begriff „Deployment“ beschreibt die Auslieferung und Verteilung, sowie die Installation und den Einsatz von Software auf einer Zielplattformen (vgl. Wikipedia 09).
Als Deployment wird die Abnahme- und Einführungsphase einer Software bezeichnet. Für die Einordnung in den Softwareentwicklungsprozess bedeutet dies, dass das Deployment erst spät eintritt. Für gewöhnlich beginnen Softwareprojekte mit einer Planungsphase, wobei je nach Auftraggeber ein Lastenheft sowie mögliche Aufwandsabschätzungen entstehen um den Umfang und die Kosten, beispielsweise für Hardware und Personal, zu kalkulieren. Im Anschluss an die Planungsphase folgt die Projektdefinition. In dieser Phase werden die Angaben zur funktionalen Sicht und zur Umsetzung präzisiert und in Modellen und Konzepten abgebildet, wodurch letztlich das sogenannte Pflichtenheft entsteht.
Nach Abschluss der Planungsaktivitäten folgt der eher praktische Teil des Projektes, in dem die definierten Anforderungen des Pflichtenheftes in der Durchführungs- beziehungsweise Implementierungsphase umgesetzt werden. Die in dieser Phase entstandene Software wird regelmäßig unter Zuhilfenahme von Messdaten und Testsystemen überprüft bis schließlich das Projekt fertig implementiert ist. Im Anschluss folgt dann die Abnahme- und die Einführungsphase (Deployment), dabei wird die zu verwendende Revision[3] freigegeben und mit einer Freigabenummer versehen. Daran schließen sich die Paketierung und die Installation beziehungsweise die Verteilung der Software auf den Zielplattformen an.

[3] Revisionen und damit verbundene Revisionsnummern werden in einem Sourcecode Management System (Versionskontrollsystem für Quellcodedateien wie Subversion oder CVS) automatisch bei einer Transaktion (Änderung von Dateien oder an Verzeichnissen) fortlaufend vergeben. Eine Revision kennzeichnet demnach alle Änderungen einer Transaktion und markiert einen gewissen Stand im Quelltextverzeichnis (vgl. Popp 2008, S. 70 und S. 88).

Für gewöhnlich folgt nach der Übergabe des Produkts an den Kunden oder an einen anonymen Markt die Phase der Wartung und Pflege des Produktes. Als Grundlage für die Wartung ist es notwendig die Produkte und Teilprodukte der vorangegangenen Abnahme- und Einführungsphase zu archivieren, wobei man zwischen Produkt- und Wartungsarchiven unterscheidet. Ein Produktarchiv enthält vor allem verschiedene Versionen der Software, bei einem Wartungsarchiv kommen dann noch Informationen über die Installation und die vom Kunden verwendeten Versionen hinzu (vgl. Balzert 2000, S. 1090).

Jede der oben genannten Phasen bezieht sich im Wesentlichen auf die Vorgehensweise des Wasserfallmodells, was bedeutet, dass eine gradlinige Durchführung von der ersten bis zu letzten Phase möglich aber nicht zwingend ist, das heißt bei Komplikationen kann zu einer vorherigen Phase zurück gegangen werden[4].

2.1 Die Abnahmephase

In der Abnahmephase wird die Freigabe des Produktes mit anschließender Übergabe an den Kunden vorbereitet und umgesetzt. Dies beinhaltet unter anderem diverse abschließende Tests wie beispielsweise Stress- und Belastungstests[5]. Dazu kommt noch ein Abnahmetest durch den Auftraggeber, in dem überprüft wird, ob die gewünschten Anforderungen an die Software enthalten sind (vgl. Balzert 2000, S. 1086-1087).

Die Ergebnisse dieser Tests werden in einem Abnahmeprotokoll dokumentiert und dem Privatkunden, einer firmeninternen Instanz oder an einen anonymen Markt übergeben. Für den Fall, dass der Kunde die Wartung und Pflege der Software selbst durchführt, ist es zudem noch notwendig die Analyse-, Entwurfs- und Implementierungsdokumentationen an den Kunden zu übergeben und eine Einweisung in die Software-Architektur durchzuführen (vgl. Balzert 2000, S. 1086-1087).

---

[4] Die aufgeführten Phasen spielen ebenso eine Rolle bei anderen Vorgehensmodellen wie beispielsweise dem Rational Unified Process (kurz RUP) oder Extreme Programming (kurz XP).

[5] Stress- und Belastungstests werden zum Test von Software und Hardware eingesetzt um Extremsituationen zu simulieren. Über derartige Tests werden Fehler aufgedeckt, Massenverarbeitung getestet und Antwortzeiten unter Volllast ermittelt.

2.2 Die Einführungsphase

In der Einführungsphase wird das Produkt auf der Zielumgebung installiert und eingerichtet und anschließend in Betrieb genommen. Für den Fall, dass es sich um einen anonymen Markt handelt, sollte eine Pilotinstallation durchgeführt werden - eine sogenannte Betaphase. Ebenfalls abhängig von der Produktart sind möglicherweise Schulungen und Einweisungen für den Einsatz neuer Software an einem Unternehmen notwendig (Balzert 2000, S. 1088-1089).
Bei der Umstellung auf das neue Produkt (also die Inbetriebnahme) kann der Wechsel direkt vom alten auf das neue System oder im Parallelbetrieb durchgeführt werden. Der Vorteil beim zeitweisen parallelen Betrieb beider Systeme liegt in der Ausfallsicherheit begründet. Bei einer direkten Umstellung hingegen wird ausschließlich die neue Software verwendet und der Betrieb der ursprünglichen Software wird eingestellt, dieses Verfahren birgt ein erhöhtes Risikopotential und sollte möglichst vermieden werden (vgl. Balzert 2000, S. 1088-1089). In beiden Fällen der Auslieferung kommt die Software zum Einsatz beim Kunden und somit in den Produktiveinsatz. Das bedeutet, dass die Verfügbarkeitsanforderung der ausgelieferten Software sehr hoch ist und dadurch auch mit gehäuften Rückmeldungen von Fehlfunktionen oder Anforderungserweiterungen zu rechnen ist. Diesen sollte man mit einem schnellen Auslieferungszyklus mit hoher Qualität im Rahmen der Wartungs- und Pflegephase begegnen.

Falls es sich um Datenbank abhängige Anwendungen handelt, muss bei der Umstellung besonderes Augenmerk auf die Integration und die Zusammenfügung verschiedenartiger Datenbestände gelegt werden. Derartige Vorgänge sind häufig sehr zeitaufwendig und dementsprechend entsteht bei fehlerhafter Planung das Problem, dass der Zeitplan der Einführung nachhaltig negativ beeinflusst wird. Gegebenenfalls müssen bei derartigen Umstellungen Transformationsprogramme und Import-Anweisungen entwickelt oder angepasst werden, die speziell auf den Datenbestand abgestimmt sind (vgl. Balzert 2000, S. 1088-1089).

3 Anforderungsanalyse

3.1 Ist-Zustandsbeschreibung

Derzeitig werden die Java-Softwareprojekte der W7X-CoDaC-Gruppe in einem SCM (Sourcecode Management System) verwaltet. Die Programmtexte (Java-Quellcode) werden dabei je nach Entwicklungszustand historisch geordnet aufbewahrt. Automatisch gestartete Build-Prozesse erzeugen aus den aktuellen Entwicklungszuständen lauffähige Software (.jar Dateien), dies geschieht auf der Integrationsplattform namens Validator2.Hudson in einem täglichen Zyklus (vgl. Kühner 2009). Zu den Build-Prozessen gehören Arbeitsschritte wie Kompilieren, Paketieren, Ausliefern, Erzeugen von Fehlerberichten und Metrikberichten. Die ausgelieferte Software steht anschließend in Test- oder Produktivumgebungen zur Anwendung bereit.

„Für die (eingebettete) Experiment-Software werden bisher die Freigaben von Revisionen, die Zuteilung von Freigabenummern, die Paketierung und Kennzeichnung vollständiger Laufzeitumgebungen, die Erstellung einer Installationsumgebung (xdvdeploy), die Verteilung auf die Zielplattform (Datenerfassungs-Rechner), die Benachrichtigung der Anwender sowie die Inbetriebnahme als Folge manueller Einzelaktivitäten (mit Skript-/Werkzeug-Unterstützung: Ant[6]/bat[7]/sh[8]) durchgeführt. […] Grafik-Anwendungen werden bisher direkt in der Integrationsumgebung im Shared File System AFS[9] […] getestet und betrieben“ (Kühner 2009).

Gegenwärtig ist die Entwicklungsumgebung nicht von der Produktivumgebung losgelöst, was bedeutet, dass Anwender wie auch Entwickler auf dieselben .jar Dateien zugreifen, welche sich in ständigem Wandel befinden. Die vorherrschende Verzeichnisstruktur ist

---

[6] Ant ist ein Werkzeug zur Erstellung von Programmen aus einem Quellcode.

[7] Eine .bat- oder auch Batch-Datei bezeichnet eine Stapelverarbeitung (also sequenzielle Aufgabenabarbeitung) unter Windows Systemen.

[8] „sh“ steht für Shell-Skripte die ähnlich einer Batch-Datei unter Unix Systemen verwendet werden.

[9] Das AFS (Andrew File System) ist ein verteiltes Dateisystem. Unter MS-Windows erscheinen AFS-Volumes als „Netzlaufwerke“, unter Unix als extern gemountete Verzeichnisse.

eher unübersichtlich ohne einheitliche Vergabe von Zeitstempeln als eine Form der Versionsverwaltung.

Die W7X-CoDaC-Gruppe betreibt derzeit einen sogenannten Hudson-Service, der als Open-Source Lösung zum alltäglichen automatischen Build des geänderten Quellcodes dient. Die automatisierten Builds sind ziemlich umfassend und beliefern im Wesentlichen eine Testumgebung. Der daran anschließende Release-Prozess ist nur wenig ausgeprägt, lediglich für die sogenannte CodaStation Software (Control and Data Acquisition Software) wird ein teilweise automatisierter Release-Prozess durchgeführt.

Zukünftig wird ein detailiert ausgearbeiteter Release-Prozess benötigt, der mit den folgenden fünf Produktarten, die sich im Ablauf unterscheiden, umgehen kann. Beispielsweise Web-Services werden in der Regel an eine Ausführungsumgebung wie einem Tomcat-Server ausgeliefert, wohingegen die Software ConfiX (grafischer Editor) in einem allgemein zugänglichen Verzeichnis abgelegt werden kann um von dort aus gestartet zu werden. Die Produktarten unterscheiden sich im Wesentlichen im Zielort der Auslieferung, welcher bei ausführbaren .jar Dateien Zielplattformen sind, auf denen die Prozesse arbeiten und von außen angesteuert werden. Bei .war Dateien handelt es sich um Anwendungen, welche auf einen Tomcat-Server ausgeliefert werden von dem sie dann via HTTP-Aufruf verwendet werden können. Eine weitere Produktart bilden die herkömmlichen Programme wie beispielsweise grafische Anwendungen, die auf die verschiedenen Rechner der Nutzer verteilt werden müssen und von dort eigenständig verwendet werden.

Tabelle 1: Produktarten des Release-Prozesses

Auslieferungszielorte	**Beschreibung und Beispiele**
ausführbare .jar Datei	1. Systemprozesse wie die CodaStation (Software zur Ansteuerung von Datenerfassungs- und Steuerelektronik sowie Vorverarbeitung und Archivierung)
	2. autonome Web-Services (ohne Ausführungsumgebung) Beispiel: Modellrechnungen
Upload der Daten in eine Zielumgebung (.war Datei)	3. Web-Service mit Auslieferung an eine Ausführungsumgebung (Tomcat-Server) Beispielprojekt: Web-Editor, dient zur Parametrierung von Experimentprogrammen

herkömmliches Programm (.jar Datei)	4. komplexe grafische Anwendungen häufig mit Datenbankzugriff Beispielprojekt: Confix, dient als Editor zur Konfiguration von Experimenten und zur Erstellung von Experimentprogrammen
spezielle Ausführungsumgebung	5. grafische Anwendungen auf OSGI Basis

(Quelle: Eigene Darstellung)

3.2 Anforderungen

Die wesentliche Anforderung an das Projekt bildet die Trennung der Entwicklungsumgebung vom Produktivbetrieb. Darüberhinaus gilt es einen möglichst automatisierten Ablauf von Build-Prozessen im Projekt für die W7X-CoDaC-Gruppe umzusetzen.

Entstandene Releases der verschiedenen Applikationen (Java Anwendungen, Rich-Client Anwendungen, Web-Anwendungen) sollen zukünftig mittels eines Deployment-Prozesses erzeugt und nach entsprechenden Tests paketiert und ausgeliefert werden. Da es sich um ein experimentelles Umfeld handelt, kann es vorkommen, dass neue Releases nicht sofort das gewünschte Ergebnis liefern, deswegen soll es die Möglichkeit zur Wiederherstellung vergangener Zustände geben. Bisher werden alte Zustände durch erneute Builds wiederhergestellt und lediglich die verwendete Fremdsoftware wird einer Versionsverwaltung von Binärdateien unterzogen, dieses Verfahren ist aufwendig und bietet nur wenig Sicherheit.

Sobald man Maven für Releases und zu Testzwecken verwendet, ist der damit verbundene Aufwand bei großen Softwareprojekten erheblich höher. Die Gründe dafür sind die verschiedenen Arten der Softwareauslieferung (beispielsweise die Auslieferungszielorte) und eine Eigenart von Maven für einen Release-Build nur bereits releaste Objekte zu verwenden. Für die W7X-CoDaC-Gruppe wurde daraufhin eine interne Nomenklatur der Begriffe der Module und Produkte eingeführt. Ein Modul ist ein Baustein mit einem bestimmten Funktionsumfang und besteht für gewöhnlich aus einer Java-Bibliothek, die eigenständig nicht lauffähig ist. Module bilden die Grundlage für komplexe Softwareprojekte. Produkte sind Aggregate[10] von Modulen, die direkt an den Nutzer oder Kunden ausgeliefert werden.

[10] Eine Aggregationsbeziehung ist laut der UML Notation eine Teil-Ganzes-Beziehung. Zum Beispiel können eine oder mehrere Vorlesungen (0 bis n) von keinem oder mehreren Studenten besucht werden (0 bis n).

Ausgelieferte Produkte dürfen in Maven nur ausgelieferte (releaste) Module enthalten, wobei Testprodukte in beliebigem Umfang Snapshots[11] und ausgelieferte Module mischen können.
Eine weitere Anforderung ist, dass die Anwender der Software die Möglichkeit bekommen beim Auftreten von inakzeptablen Fehlern in der laufenden Software zwischen mehreren älteren Releases auszuwählen und ein geeignetes Release zu reaktivieren um einen Zeitverlust möglichst gering zu halten (maximal eine Stunde). An Tagen, an denen Testläufe mit der Anlage durchgeführt werden, soll so die Ausfallzeit und das Risiko gesenkt werden.

Die durch die W7X-CoDaC-Gruppe stetig größer werdende Menge an Software und deren Einsatz zur Erfassung von Messdaten durch den Endnutzer bewirken die Notwendigkeit der Entwicklung und des Einsatzes von Qualitätsmanagement.

3.2.1 Zusammenfassung der Anforderungsanalyse

- stabile Produktivumgebung
- Wiederherstellbarkeit von erzeugten Produkten innerhalb einer Stunde (Minimierung der Ausfallzeiten)[12]
- automatisierter Integrations-Build und teilautomatisierter Release-Build[12]
- Paketierung und Auslieferung der Module und Produkte
- Versionsverwaltung für Produkte
- Benutzerfreundlichkeit (zentrale Verwaltungsmöglichkeit)[12]

[11] Snapshots bezeichnen die Zwischenstadien von Produkten bei der Entwicklung eines Releases (vgl. Popp 2009, S. 258).

[12] nichtfunktionale Anforderungen

4 Umsetzung und Lösungsansätze

Nach der grundsätzlichen Anforderungsanalyse schließt sich der praktische Teil an - die Umsetzung. Es folgt eine theoretische Beschreibung der Arbeitsabläufe, sowie eine Darstellung der Probleme mit anschließenden Lösungsansätzen, die bereits am Max-Plank-Institut durchgeführt worden sind.

4.1 Welches Build-Tool soll verwendet werden?

Ein zentrales Thema bei der Umsetzung der Anforderungen ist die Wahl des richtigen Build-Tools, hierzu wurde zuallererst untersucht welches der beiden in Frage kommenden Tools verwendet werden soll. Zum einen gibt es das weit verbreitete Apache Ant mit einer Vielzahl an Möglichkeiten zur Feinabstimmung der Ablaufprozesse und auf der anderen Seite Maven zur schnellen Umsetzung für mehrere Softwareprojekte. Die wesentlichen Vorteile bei Maven liegen zum einen im deklarativen Vorgehen, in dem man die gewünschten Ziele definiert und Maven die Durchführung übernimmt. Desweiteren bietet Maven in der Verwendung und Verwaltung von mehreren Repositorys[13] einen wesentlichen Vorteil im Umgang mit mehreren eigenen oder fremden Softwarekomponenten. Ant basiert auf einem prozeduralen Ansatz, bei dem die Abarbeitungsreihenfolge explizit vorgegeben werden muss.

Im vorliegenden Fall der W7X-CoDaC-Gruppe handelt es sich um eine Multi-Projektstruktur. Die entwickelte Software besteht zumeist aus mehreren selbstentworfenen Java-Klassenbibliotheken. Die Programmdateien dieser Java-Bibliotheken werden über ein SCM verwaltet, bei der W7X-CoDaC-Gruppe erledigt das ein Subversion-Repository (kurz SVN) [14].
Derzeitig wird der größte Teil der Projekte mit Ant gebaut, welches an ein Automatisierungs- und Überwachungssystem namens Hudson gebunden ist. Dieser Hudson-Service steuert somit nach einem vorgegebenen Zeitplan, häufig abends um den Arbeitsprozess

---

[13] Repositorys sind Verzeichnisse zur Speicherung und Verwaltung von Daten. Sie dienen beispielsweise der Versionsverwaltung (CVS und SVN) von Quellcodedateien oder wie im Fall von Maven vor allem der Lagerung von Programmpaketen und deren Metadaten.

[14] Siehe Anhang 8.1 Ausschnitt des SVN-Repositorys zu DbUtil.

nicht zu stören, den Ablauf des wiederkehrenden Builds-Prozesses. Der Vorgang ist in sofern komfortabel, dass sobald alle Abhängigkeiten und Einstellungen korrekt vorgenommen sind, der Build nach einer Veränderung an einem Softwareprojekt automatisch durchgeführt wird oder auch per Hand gestartet werden kann. Allerdings erfordert die Einrichtung eines Ant-Ablaufes einiges an Zeit, da jedes Projekt eine oder mehrere eigens zugeschnittene .xml Dateien besitzt, die die Umsetzung regeln.

4.1.1 Maven versus Ant

In diesem Abschnitt sollen die Gemeinsamkeiten und Unterschiede zwischen Maven und Ant verdeutlicht werden und eine Entscheidung für eines der beiden Tools getroffen werden.

Ant wird häufig als Nachfolger von Make beschrieben, da es die Funktionen für den Einsatz unter Java erweitert (vgl. Apache Ant 1.7.1 Manual 2009). Die Hersteller von Maven hingegen distanzieren sich von einem Vergleich mit anderen Build-Tools, vielmehr wird Wert auf die Andersartigkeit von Maven und Ant gelegt (vgl. Popp 2008, S. 4).

„Maven basiert auf einem deklarativen, modellbasierten Ansatz“ (Popp 2008, S. 4). Der deklarative Ansatz ermöglicht es, anhand eines vorgegebenen Abarbeitungsschemas, Ziele zur Umsetzung vorzugeben. Diese Ziele können mittels einer sogenannten POM (Projekt Objekt Model) Datei im Verlauf angepasst werden, wodurch das Standardabarbeitungsschema erweitert wird. Ant hingegen basiert auf einem prozeduralen Ansatz, in dem jeder einzelne Schritt mittels einer .xml Datei definiert werden muss.

Maven bietet als wesentliche Neuerung eine Repository-Struktur und -Verwaltung mit Versionierung. Über ein sogenanntes Remote-Repository erhält man die Möglichkeit Maven-Erweiterungen (Plugins) von Drittanbietern über das Internet zu beziehen. Dieser Abgleich, der verwendeten Plugin-Version, kann je nach Einstellung bei jedem Schritt eines Build-Prozesses automatisch durchgeführt oder geblockt[15] werden. Auf die gleiche Weise kann man aus dem Remote-Repository allgemein verwendbare, quelloffene Softwarekomponenten (.jar Dateien) beziehen, die somit nicht selbst entwickelt werden müssen.

[15] Das Blocken der Updatefunktion kann abhängig vom Softwareprojekt sinnvoll sein um stets dieselben Versionen eines Plugins zu verwenden und damit einen konsistenten Build-Prozess zu erhalten.

Dies entspricht den Anforderungen der W7X-CoDaC-Gruppe, die bei der Entwicklung im großen Maße auf quelloffene Software zurückgreift.
Die Repository-Struktur eignet sich ebenfalls hervorragend, um eigene Strukturen zur Ablage von entstandenen Build-Produkten zu verwalten[16]. Somit wird die Versionsverwaltung der entstandenen Produkte automatisiert und eine Wiederherstellung von älteren Releases kann gewährleistet werden.
Wie bereits in der Anforderungsanalyse (Siehe 3.2) erwähnt, soll der entstehende Build-Prozess eine stabile Produktivumgebung mit sich bringen. Im vorliegenden Fall kann Maven dies durch die Trennung der Entwicklungsumgebung von der Produktivumgebung gewährleisten, indem für die Auslieferung der jeweiligen Produkte eigenständige Schreibverzeichnisse angelegt werden können. Diese Verzeichnisse werden zumeist als Snapshot- und Release-Repositorys bezeichnet.

Beide Build-Tools haben gemeinsam, dass sie eine gute Integration in das durch die W7X-CoDaC-Gruppe am IPP schon eingeführte Hudson-System vorweisen. Dieses System ist auf eine Zusammenarbeit mit den Build-Tools ausgelegt und eignet sich zum Überwachen und Testen (vgl. Hudson 2009). Build-Systeme wie Ant und Make können vom Hudson nur als Prozesse gestartet werden und erfordern einen relativ hohen Aufwand zur Weiterverarbeitung der erzeugten Reporte. Maven-Builds hingegen sind viel einfacher zu handhaben, da Hudson die POM Dateien direkt liest und versteht.

4.1.1.1 Entscheidung für Maven

Für die vorliegenden Bedingungen und Anforderungen der W7X-CoDaC-Gruppe ist die Wahl auf Maven gefallen, da Maven besonders durch das Repository und einige andere Funktionalitäten besser geeignet ist. Allerdings ist Maven eine noch relativ neue Entwicklung, möglicherweise fehlende Funktionalitäten, die es unter Ant bereits gab, können mittels eines Plugins in Maven integriert werden, wodurch kein Nachteil bei der Benutzung von Maven entsteht.

Mavens größter Vorteil liegt in der Verwendung eines oder mehrerer Repositorys, wodurch gleich einige Anforderungen wie die Wiederherstellung von alten Zuständen, eine geordnete Ablagestruktur und die Versionsverwaltung abgehandelt werden können.

[16] Siehe Anhang 8.2 Privates Maven-Repository für Build-Produkte.

4.1.1.2 Installation von Maven

Grundsätzlich kann Maven auf dem privaten Rechner (oder wie es im IPP üblich ist) auf dem Netzlaufwerk AFS abgelegt werden. Die Software und eine englische Anleitung zur Installation sind kostenfrei auf der Herstellerseite[17] zu finden. Grundsätzlich ist nach der Installation das Setzen von Umgebungsvariablen notwendig, damit Maven über die Kommandozeile ansprechbar wird[18]. Falls alle Einstellungen korrekt vorgenommen worden sind, kann man Maven über die Kommandozeile mit dem Aufruf „mvn --version" testen. Zum Zeitpunkt der Erstellung der Bachelorarbeit wurde mit der derzeit aktuellsten Version 2.2.0 von Maven gearbeitet.

Für die Arbeit mit Maven als Entwicklungstool für Java Quellcode bietet sich die Möglichkeit die gängigen IDEs (Eclipse[19], NetBeans) über ein Plugin zur Unterstützung von Maven zu erweitern. Im Falle des Eclipse-Plugins wird die Arbeit an den POM Dateien dadurch erheblich erleichtert, dass man die .xml Dateien (die mitunter über 400 Zeilen lang werden) über den Editor direkt manipulieren kann oder einen mehrseitigen Spezialeditor mit guter Benutzerführung verwenden kann. Darüberhinaus gibt es noch eine Hierarchiefunktionalität (Dependency Graph), die hilfreich für einen Überblick über die Abhängigkeiten des zu betrachtenden Projektes ist. Zum Beispiel entsteht bei einem komplexen Produkt wie dem Confix[20], welches mehrere Abhängigkeiten zu verschiedenen Modulen aufweist, die wiederum Abhängigkeiten zu weiteren Modulen besitzen, ein vielschichtiges Geflächt von Abhängigkeiten. Die Hierarchiefunktionalität hilft vor allem beim Verwenden der korrekten Modul-Versionen und ist somit besonders für eine Multiprojektstruktur vorteilhaft.

4.1.1.3 Umstellung auf Maven

Eine Frage die sich jeder Entwickler von Software stellen wird ist, ob Maven sich mit einem bestehenden Build-System wie beispielsweise Ant oder Make verträgt, beziehungsweise ob es sich gut für ein neues Projekt einrichten lässt? Das kann relativ einfach mit einem ja beantwortet werden. Maven kann für ein neues Projekt einfach verwendet werden. Aber auch wenn zuvor beispielsweise Ant genutzt wurde, kann Maven parallel zu

---

[17] http://maven.apache.org/download.html

[18] Siehe Anhang 8.3 Umgebungsvariablen für Maven.

[19] Installation über den Updatemanager von Eclipse http://m2eclipse.sonatype.org/update/

[20] Siehe Abbildung 3: Hierarchiestruktur von Maven für Confix (Dependency Graph).

einem bestehenden System integriert werden, da die einzige Schnittstelle der beiden Systeme der Quellcode selbst ist. Darüberhinaus können möglicherweise Speziallösungen, die mit Ant direkt auf ein bestehendes Projekt angepasst worden sind, mit in den neuen Maven- Ablauf integriert werden.

Die Umstellung auf Maven ist lediglich für das erste Projekt etwas aufwendig, da die Abläufe für die einzelnen Schritte und Abfolgen konfiguriert werden müssen. Im Falle der W7X-CoDaC-Gruppe mussten erst Strukturen, wie Ablageordner und die Vergabe von Zugriffsberechtigungen, neu geschaffen werden. Für alle weiteren Projekte, die in der Regel nur in wenigen Punkten (beispielsweise im Auslieferungszielordner) abgeändert werden müssen, ist dies schnell zu bewerkstelligen. Im Grunde kann die POM Datei, die den Maven-Ablauf steuert, auch für ein neues Projekt angewendet werden und muss nur noch angepasst werden, somit reduziert sich der Aufwand für neue Projekte auf ein Minimum[21]. Erfahrungsgemäß würde dieser Prozess der Umstellung eines Projektes auf beispielsweise Ant erheblich länger dauern, da das gesamte Abarbeitungsskript verändert wird, weil die Pfade überarbeitet werden müssen, was wiederum eher zu einer Neuentwicklung des Abarbeitungsskripts führt. Allerdings können auch bei Ant gewisse Teile in separate Dateien ausgelagert und für andere Projekte wiederverwendet werden.

Auf der parktischen Seite bei der Umsetzung für die W7X-CoDaC-Gruppe sind diesbezüglich keine Probleme aufgetreten. Das laufende, auf Ant in Zusammenarbeit mit dem Hudson-Service basierende, System wurde kontinuierlich weiter betrieben und zusätzlich wurden mehrere Projekte auf Maven umgestellt und eingerichtet. Performance, Ausfallsicherheit und Stabilität hatten auf keines der beiden Systeme, Ant oder Maven, einen spürbar negativen Einfluss.

4.2 Stabile Produktivumgebung

Unter einer stabilen Produktivumgebung versteht man im Allgemeinen eine voneinander räumlich getrennte Umgebung für den Prozess der Entwicklung von Software zu der Umgebung in der es verwendet wird. Das ist im vorliegenden Fall der W7X-CoDaC-Gruppe nicht so, denn der Produktivbetrieb hatte sich historisch entwickelt und soll nun in organisierter Form eingeführt werden.

[21] durchschnittliche Dauer von etwa 2 Stunden, abhängig von dem Projektumfang

Wie in der folgenden Abbildung ersichtlich wird, gibt es nur ein einziges Zielverzeichnis mit einer flachen Struktur und keine einheitliche Versionsverwaltung. Die aktuellste und derzeit im Produktivbetrieb befindliche Version ist anhand der Abbildung nicht mit einem angehängten Zeitstempel versehen (confix.jar, objyaccess.jar, oojava.jar), ältere und zum größten Teil nicht mehr verwendete Dateien besitzen einen solchen Zeitstempel.

Name	Größe
confix.jar	771 KB
confix.jar.bis090424	793 KB
confix_010709Wed211421.jar	763 KB
confix_020609Tue211330.jar	760 KB
confix_020709Thu212333.jar	763 KB
confix_030609Wed212318.jar	760 KB
confix_030709Fri212353.jar	763 KB
confix_060709Mon212206.jar	763 KB
confix_070709Tue212349.jar	779 KB
confix_080709Wed212345.jar	765 KB
confix_090609Tue212306.jar	760 KB
confix_100609Wed211354.jar	760 KB
confix_100709Fri212233.jar	803 KB
confix_110609Thu211338.jar	760 KB
confix_130509Wed100411.jar	771 KB

Name	Größe
objyaccess2005.jar	627 KB
objyaccess2006.jar	612 KB
objyaccess2007.jar	363 KB
objyaccess2008.jar	681 KB
objyaccess11022009.jar	690 KB
objyaccess11032009.jar	659 KB
objyaccess12032009.jar	688 KB
objyaccess23042009.jar	689 KB
objyaccess24062009.jar	689 KB
objyaccess25022009.jar	691 KB
objyaccess28012009.jar	687 KB
objyaccess.jar	690 KB
objyaccesstest.jar	695 KB
oojava90.jar	617 KB
oojava.jar	617 KB

Abbildung 1: Ungeordnete Struktur der Entwicklungs- und Produktivumgebung
(Quelle: Eigene Darstellung)

Die Nutzer der entwickelten Software greifen mittels einer Batch-Datei (Stapelverarbeitung, .bat Datei) auf diese Java-Bibliotheken zu, was zu einigen Problem in der Vergangenheit geführt hat. Häufig wurden nach einer Änderung an der Software oder Teilen davon die neuen Bibliotheken mittels des Hudson-Systems in einem abendlichen Build-Prozess ausgeliefert. Die neu entstandenen Bibliotheken führten häufig zu Ausfällen bei laufenden Anwendungen am darauf folgenden Morgen. Da die Anwendungen häufig im Dauerbetrieb eingesetzt werden, sind sie anfällig für Änderungen. Die Folgen waren Abstürze der Software und im schlimmsten Fall Datenverlust.

Um diesem Problem entgegen zu wirken muss nun eine Struktur geschaffen werden, die es ermöglicht auf mehr als einen Zustand der Software zu zugreifen und die es gewährleistet, dass das Entwicklerteam die entstandene Software testen kann bevor sie zur Freigabe an die Nutzer weitergereicht wird.

Die Lösung des Problems kann zum einen durch die Trennung von Entwicklungs- und Produktiv-Software in der Repository Struktur von Maven gewährleistet werden und zum anderen durch Verwendung von räumlich getrennten Auslieferungsverzeichnissen für Entwicklung und Produktiveinsatz geschehen.

4.3 Wiederherstellbarkeit von Produkten

Nach einer Umstellung auf neue Releases ist die Chance auf einen möglichen Ausfall der Datenerfassungssoftware am wahrscheinlichsten. Am IPP in Greifswald wird bisher die von der W7X-CoDaC-Gruppe entwickelte Software schwerpunktmäßig an zwei Stellen eingesetzt, wöchentlich am Dienstag werden an einer Testanlage namens WEGA[22] Systemtests in Form von Experimentabläufen durchgeführt. Weiterhin wird dieselbe Software in Laboren eingesetzt, um die Messverfahren zu verarbeiten. Entstandene Probleme mit neuen Releases können in voller Breite erst zum Zeitpunkt des Testes auftreten. Um entsprechende Warte- und Ausfallzeiten so gering wie möglich zu halten, ist es vorgesehen, einen alten lauffähigen Zustand in weniger als einer Stunde wieder herstellen zu können beziehungsweise ein älteres Release erneut zu erzeugen.

Bisher wurden aufgrund des aktuellen Standes des Quellcodes, wie er im SCM (SVN-Repository) vorlag, neue .jar Dateien erzeugt. Das Problem dabei bestand darin, dass das ursprüngliche Releas-Produkt überschrieben wurde, wodurch alte Zustände verlorengegangen sind. Wenn unter diesen Bedingungen ein alter Zustand wiederhergestellt werden soll, müssen alle notwendigen Bibliotheken neu erzeugt werden. Durch die Nutzung von Maven und dessen Repository-Struktur ist es möglich alle Projekte (Module und Produkte) wiederzuverwenden, da jeder erfolgreiche Maven-Build seine fertigen Auslieferungsdateien in separate Verzeichnisse im Repository ablegt, die dann wiederverwendet werden können. Für eine Wiederherstellung ist somit lediglich das Wissen über die Versionsnummern beziehungsweise die Nummer des verwendeten Snapshots der abhängigen Bibliotheken notwendig.

Die geplanten Maven-Builds sehen für jeden Build-Prozess eine Erweiterung des ursprünglichen Ablaufes vor. Sie sind mit einem Ant-Script versehen, welches eine zusätzliche Markierung (SVN-Tag) im Quellcode-Repository des SCM erzeugt. Diese Markierung hat den Zweck den derzeitigen Stand des Quellcodes festzuhalten. Dateien, die über den Build-Prozess auf diese Weise automatisch gesichert werden, entsprechen somit einem Abbild des Entwicklungsstandes nach einer jeden Änderung. Bei einer erstmaligen Verwendung des Skriptes muss der Zielordner (wie zum Beispiel ein Snapshot-Verzeichnis) von Hand im SVN erstellt werden. Diese Markierungen entstehen demzufolge in jedem erfolgreichen

[22] WEGA bedeutet **W**endelstein **E**xperiment in **G**reifswald für die **A**usbildung.

Build-Prozess, was die Wiederherstellbarkeit von Produkten stark verbessert, allerdings auch für erheblich mehr Datenvolumen sorgt und dementsprechend bei Gelegenheit gesäubert werden sollte.

Am Max-Plank-Institut in Greifswald wurden zuvor derartige Sicherungsmechanismen nicht eingesetzt, wodurch eine Wiederherstellung von Produkten häufig erheblich länger dauerte.

Die im sogenannten Projekt Objekt Modell (kurz POM) verwendeten Abhängigkeiten zu Bibliotheken werden ebenfalls gesichert. Durch diese zusätzliche Funktionalität im Build-Prozess kann mit Hilfe der Markierung ein älterer Zustand schnell erneut erzeugt werden. In der folgenden Abbildung 2 zeichnet sich zum einen der Trunk (Hauptentwicklungszweig) des Projektes CodacUtil[23] ab, der den aktuellen Stand der Entwicklung wiederspiegelt, und zum anderen die Markierung beziehungsweise das Tag-Verzeichnis des Projektes. Dieses verändert sich ab dem Zeitpunkt des Build-Vorganges nicht mehr. Die enthaltenen Dateien sind namentlich im Trunk und im Tag-Verzeichnis gleich, allerdings sind gewisse Revisionsnummern im Trunk schon durch Weiterentwicklung höher.

[23] Das Projekt CodacUtil ist eine Bibliothek für allgemein verwendbare Funktionen/Algorithmen ohne Abhängigkeiten zu anderen Bibliotheken mit Ausnahme des JDK (Java Development Kit).

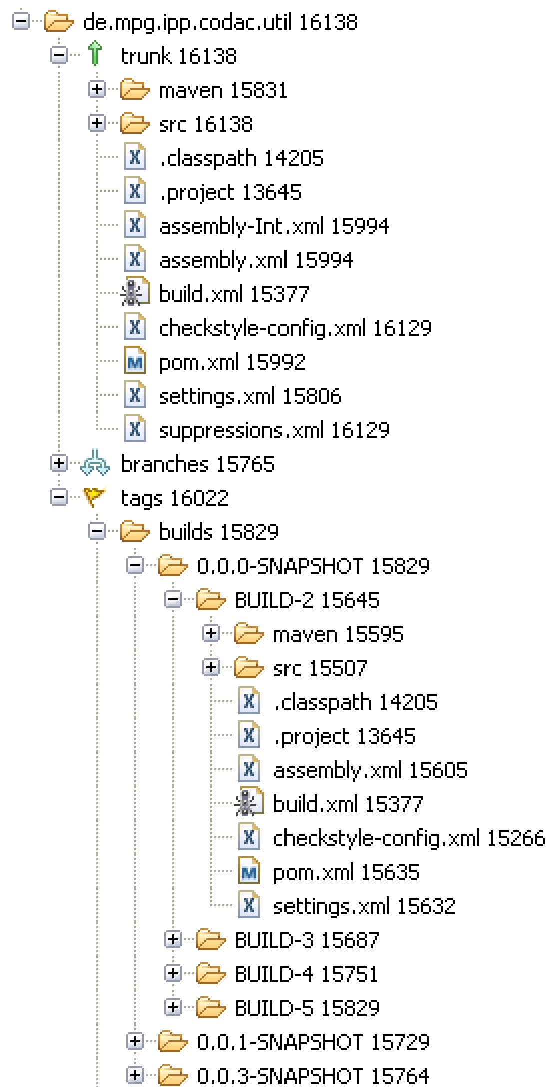

Abbildung 2: SVN-Tag von CodacUtil

(Quelle: Eigene Darstellung)

4.4 Build-Prozesse

Man sieht in der Abbildung 3 zum einen das Produkt Confix und dessen Abhängigkeiten, die sich nach unten weiter verzweigen und zum anderen mehrere Module, die vom Confix verwendet werden. Als Module werden in diesem Fall DbUtilLifecycle, ObjyAcces, CodacUtil, OoJava und noch einige mehr bezeichnet. Die Hierarchiedarstellung, welche durch den Editor des Eclipse-Plugins für POM Dateien erzeugt wurde, entsteht aufgrund der Voreinstellungen des Entwicklers und der durch Maven rekonstruierten vollständigen Hierarchie aus den zur Verfügung stehenden Repositorys. Entscheidend bei dieser Art der Aufteilung ist der Aufwand, der beispielsweise bei einem Produkt, welches an den Nutzer ausgeliefert wird, erheblich höher ist als es für ein Modul erforderlich ist.

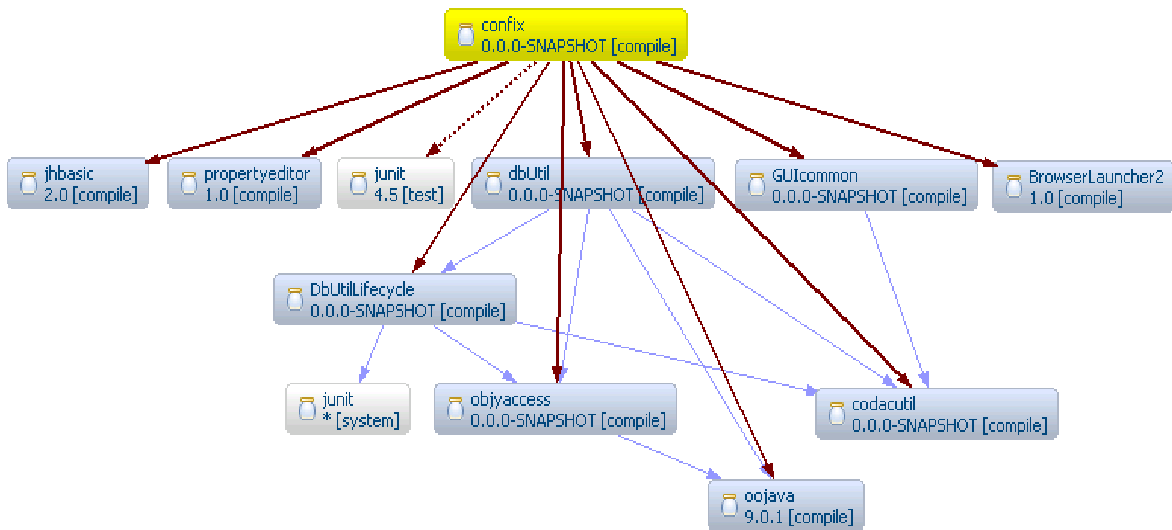

Abbildung 3: Hierarchiestruktur von Maven für Confix (Dependency Graph)
(Quelle: Eigene Darstellung)

In Zusammenarbeit mit der W7X-CoDaC-Gruppe wurden drei wesentliche Build-Prozesse erarbeitet, die sich für die Entwicklung als sinnvoll herausgestellt haben. Gemeint sind die bereits thematisierten Modul- und Produkt-Release-Builds sowie eine Integrations-Build, der sich speziell für den Entwicklungszyklus von Software eignet. Das folgende Aktivitätsdiagramm in Abbildung 4 zeigt einen standardmäßig vorgesehenen Entwicklungsprozess einer Software. Gestartet wird mit der Entwicklung der Software, woraufhin schon Integrations-Builds durchgeführt werden können. Diese produzieren nach erfolgreichem Abschluss die sogenannten Snapshots, welche an die Testumgebung ausgeliefert werden. Die Testumgebung wird für Systemtests der Steuer- und Datenerfassungssoftware sowie für Tests von GUI-Anwendungen verwendet.

Ein erfolgreicher Integrations-Build beinhaltet einen Integrationstest, der durch den Integrations-Build neu entstandene Snapshot wird an das Verzeichnis der Testumgebung ausgeliefert. Der Entwickler testet im Anschluss seinen Snapshot in Verbindung mit anderen bereits vorhandenen Modulen. Für den Fall, dass einer der beiden Prozesse (Integrationsbuild oder Integrationstest) fehlgeschlagen ist, wird zum Ausgangspunkt also zur Entwicklung zurückgekehrt.

Dieser erste Abschnitt wurde im Rahmen der Bachelorarbeit zwar überarbeitet und auf Maven umgestellt, allerdings ist nur die Auslieferung der Snapshots an einen separaten Testbereich eine wirkliche Neuerung, da ein vergleichbares Verfahren unter Ant bereits etabliert ist.

Der folgende Schritt ist durch die Form der auszuliefernden Software geprägt. Wie bereits erwähnt gibt es die Unterscheidung in Module und Produkte. Wenn es sich also um ein Modul handelt, wird der entsprechende Modul-Release-Build durchgeführt, der seine Dateien an das Maven-Repository für Releases ausliefert. Diese Moduldateien sind somit in releaster Form für einen Produkt-Release-Build zugänglich. Die zweite Möglichkeit ist der Produkt-Release-Build. Wenn alle Abhängigkeiten in releaster Form im Maven-Repository vorliegen oder von externen Quellen beziehungsweise Bibliotheken mittels Maven importiert[24] wurden, kann der Produkt-Release-Build durchgeführt werden. Dieser liefert seine Daten an ein weiteres Verzeichnis (die Laufzeitumgebung), von wo es für die Endnutzer möglich ist darauf zu zugreifen.

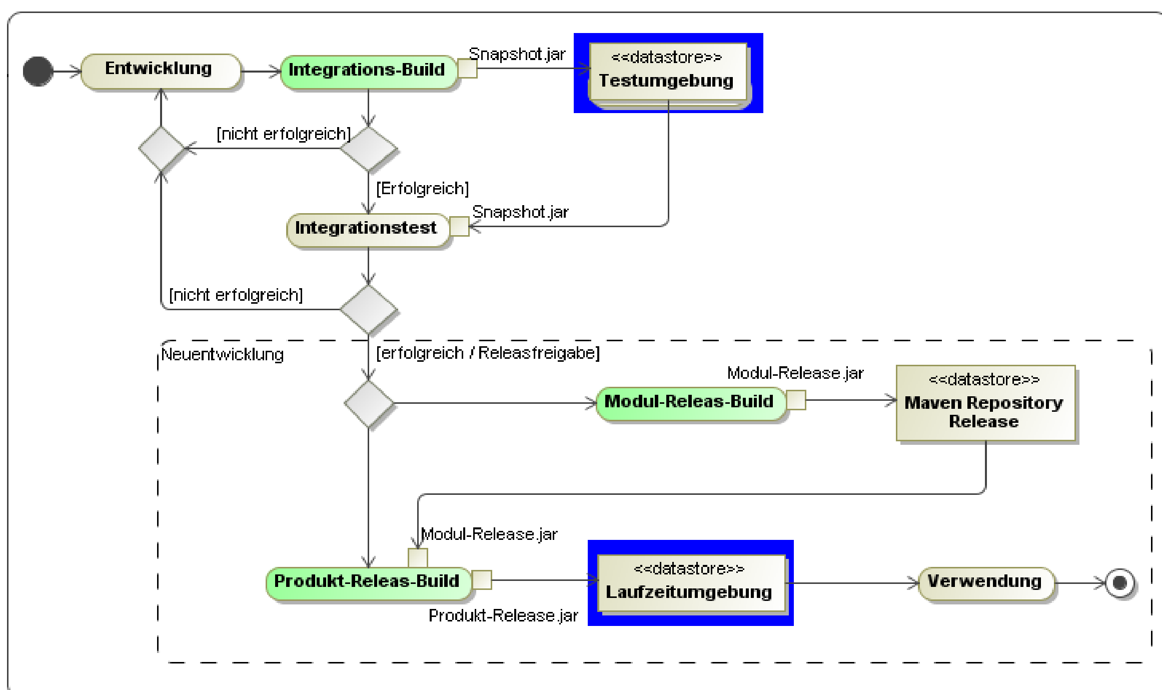

Abbildung 4: Aktivitätsdiagram der Reihenfolge für Build-Prozesse
(Quelle: Eigene Darstellung)

Die in der obigen Grafik als Testumgebung beziehungsweise Laufzeitumgebung aufgeführten Verzeichnisse befinden sich im Falle des IPP auf dem Netzlaufwerk AFS.

Unter den Builds vereinfacht sich vor allem der Produkt-Release-Build im hohen Maße, je mehr Abhängigkeiten des Produktes bereits auf Maven umgestellt sind, denn diese Abhängigkeiten lassen sich sehr einfach auflösen durch die Wahl der entsprechenden .jar Datei im lokalen Maven-Repository. Allerdings bietet Maven auch für noch nicht im Maven-

[24] Siehe 4.6.1 Interne und externen Bibliotheken.

Repository vorliegende .jar Dateien eine Art Übergangslösung, indem man Plattform abhängige Systempfade verwendet. Diese Lösung ist für den Einstieg in Maven durchaus geeignet, für einen dauerhaften Betrieb wie bei der W7X-CoDaC-Gruppe jedoch nicht optimal. Das Problem bei der Plattformabhängigkeit ist, dass man für jede Plattform eine eigene POM Datei benötigt.

4.4.1 Phasen des Build-Prozesses

Der Vorteil eines Build-Prozesses liegt eindeutig in der Ansteuerung der Phasen eines jeden Prozessschrittes. Somit können beispielsweise zu Testzwecken alle Abhängigkeiten bei der Entwicklung überprüft werden, sprich ob Bibliotheken vorhanden sind und deren Versionen stimmen. Die sequenzielle Abarbeitung ermöglicht somit Maven beispielsweise bis zur Phase des Testens durchlaufen zu lassen um die Ressourcen und das Kompilieren zu überprüfen, ohne die neu entstandenen Ergebnisse im letzten Deploymentschritt auszuliefern.

Eine Zusammenstellung der wichtigsten Phasen im Maven Build-Prozess kann man der anschließenden Tabelle 2 entnehmen. In der ersten Phase werden alle benötigten Abhängigkeiten zusammengetragen, woraufhin der Quellcode kompiliert wird. Es folgen zwei Phasen zum Testen, in denen eine Reihe von Testverfahren[25] wie beispielsweise Checkstyle oder PMD zur Erstellung von Metriken genutzt werden können. Ebenfalls in der Testphase erfolgt die Erstellung einer Projektwebseite, die zur Auswertung der Metriken verwendet werden kann. Die folgenden drei Phasen des Build-Prozesses befassen sich mit der Auslieferung der Ergebnisse, diese werden zunächst standardmäßig als .jar Datei verpackt und im Installationsprozessschritt an das Maven-Repository ausgeliefert. Der letzte Schritt liefert die Dateien an das gewünschte Verzeichnis zur Verwendung durch den Endnutzer aus.

Für die W7X-CoDaC-Gruppe wurde die letzte Phase (deploy) durch ein zusätzliches auf Ant basierendes Skript erweitert, um eine Sicherung des Quellcodes im SVN zu gewährleisten[26].

---

[25] Als Grundlage im Umgang mit Testverfahren diente die Lehrveranstaltung Case Tools - Softwarequalitätssicherung an der Fachhochschule Stralsund.

[26] Nähere Erläuterung über das Ant-Skript und die Sicherung befinden sich unter 4.3 Wiederherstellbarkeit von Produkten.

Tabelle 2: Lifecycle-Phasen für den Datentyp .jar

Lifecycle-Phase	Plugin	Build-Ziel
process-resources	maven-resources-plugin	resources
compile	maven-compiler-plugin	compile
process-test-resources	maven-resources-plugin	testResources
test-compile	maven-compiler-plugin	testCompile
test	maven-surefire-plugin	test
package	maven-jar-plugin	jar
install	maven-install-plugin	install
deploy	maven-deploy-plugin	deploy

(Quelle: Popp 2008, S. 248)

4.4.2 Integrations-Build

Der Integrations-Build ist der grundlegendste und damit der durch den Entwickler am häufigsten verwendete der drei Build-Prozesse, da er für Produkte und Module gleichermaßen zutreffend ist. In der folgenden Abbildung 5 ist der Ablauf schematisch dargestellt. Als Entwickler eines entsprechenden Softwareprojekts beginnt man für gewöhnlich mir dem Auschecken des Quellcodes beziehungsweise der Dateien aus einem Repository, wie dem SVN. Das Projekt liegt nun zur Entwicklung im Arbeitsbereich des Entwicklers vor und die eigentlichen Änderungen und Neuentwicklungen am Projekt können durchgeführt werden. Ein Entwicklungsschritt wird immer abgeschlossen mit einer Sicherung des Projektzustandes im SCM.

Der nun folgende Schritt kann entweder manuell gestartet werden oder alternativ durch ein entsprechendes System, wie dem Hudson, servergesteuert in einem automatisch gestarteten Build-Prozess (beispielsweise am Abend) nach einer Änderung im SVN durchgeführt werden. In beiden Fällen ist es vorher notwendig den aktuellen Stand des Quellcodes beziehungsweise des Softwareprojektes auf den SVN-Server (SCM) zu sichern.

Der Maven-Integrations-Build benötigt zum Aufruf verschiedene Ziele und Übergabeparameter, diese werden in der Literatur zu Maven häufig als Goals bezeichnet. Die angegebenen Ziele werden entsprechend ihrer Reihenfolge im Build-Prozess nacheinander abgearbeitet, solange keine Fehler entstehen wie zum Beispiel durch fehlende Abhängigkeiten oder dergleichen. Die durch den Prozess erzeugten Snapshots werden an das Maven-

Repository für Snapshots und an die Testumgebung ausgeliefert. Mit Hilfe der Testumgebung können die Entwickler, getrennt von der Produktivumgebung, Integrationstest durchführen. Die dabei erzeugten Snapshots werden im Maven-Repository abgelegt und dienen als Grundlage für darauf folgende Release-Builds und zur Sicherung, um ältere Zustände wiederherzustellen. Am Ende eines jeden erfolgreichen Integrations-Builds wird ein SVN-Tag erstellt, der eine nach außen sichtbare Markierung und damit einen gesicherten Zustand darstellt.

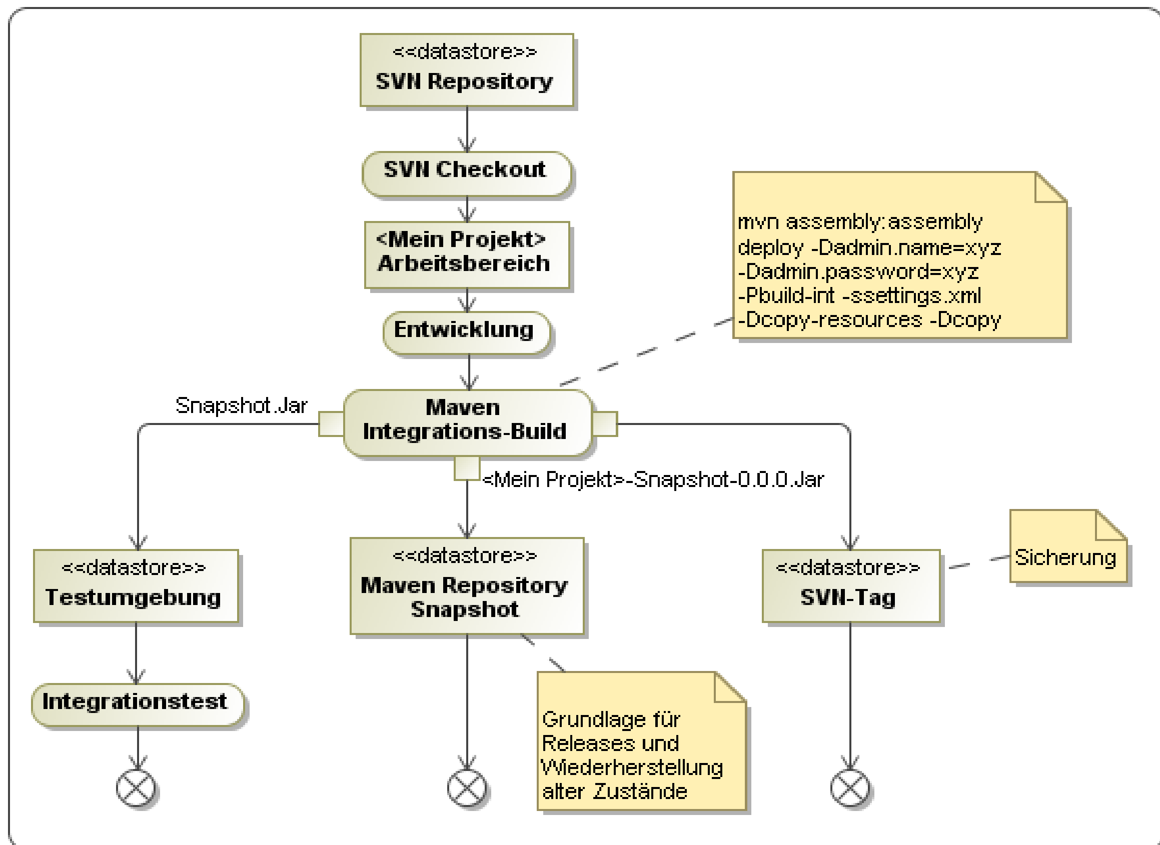

Abbildung 5: Aktivitätsdiagram für einen Integrations-Build
(Quelle: Eigene Darstellung)

Die Ziele (Goals) des Integrations-Builds, welche in der obigen Grafik durch eine Infotafel gekennzeichnet sind, sind zum einen der Assembly-Aufruf, welcher zum Packen der Auslieferungsdateien in beispielsweise .zip, .jar oder .war Format zuständig ist, und zum anderen das Deploy, welches die Tiefe des Build-Prozesses angibt (in diesem Fall ein kompletter Durchlauf). Darüberhinaus werden noch einzelne Parameter übergeben. Einer dieser Parameter ist der Adminname, ein anderer das zugehörige Password, um Schreibberechti-

gung auf dem SVN zu erlangen. Zugegebenermaßen ist dies - aus Gründen des Datenschutzes - die denkbar schlechteste Lösung aber zu Testzwecken die einfachste[27].

In einer POM Datei können verschiedene Profile voreingestellt werden, diese werden über einen Parameter beim Start von Maven aufgerufen (in diesem Fall „-Pbuild-int“) und erlauben somit Grundeinstellungen zu verändern oder zu erweitern. Beispielsweise können so verwendete Plugins zur Auslieferung oder zum Verpacken der Dateien für verschiedene Profile wie Integrations-Build, Modul- oder Produkt-Release-Build unterschieden werden. Hier wird als Parameter „-Pbuild-int“ verwendet, der für den Integrations-Build entsprechende Teil des Plugins und die Konfigurationen werden geladen und über den Aufruf der Settings.xml kann der Ort des entsprechenden Maven-Repositorys vordefiniert werden. Der Aufruf einer Settings.xml Datei ist für ein Standardverfahren nicht zwingend notwendig, da das zu verwendende Maven-Repository auf dem lokalen Rechner verwendet oder angelegt wird.
Die letzten beiden Parameter („-Dcopy -recources“ und „-Dcopy“) sind Aufrufe für Kopierbefehle, die zum einen zusätzliche Ressourcen wie beispielsweise Icons oder Bilder enthalten und zum anderen das Endprodukt des Build-Prozesses in die Testumgebung ausliefern. Die neu entstandene Testumgebung behebt das bei der W7X-CoDaC-Gruppe bestehende Problem der Vermischung der Produktiv- und der Entwicklungsumgebung, indem verschiedene Auslieferungsverzeichnisse für jeden Zweck erstellt worden sind.

4.4.3 Modul-Release-Build

Der Modul-Release-Build nimmt eine Sonderstellung in der Reihe der Build-Prozesse ein. Derartig entstandene Modul-Releases sind notwendig für einen Produkt-Release-Build, da dieser bereits releasete Produkte voraussetzt. Der dabei nötige Aufwand jedes Modul einem vollen Produkt-Release-Build[28] zu unterziehen ist unverhältnismäßig, da Module nicht direkt an den Nutzer ausgeliefert werden. Insbesondere kann auf die Erstellung eines eigenen Release-Zweiges im SVN verzichtet werden. Aus diesen Gründen wurde bei der Entwicklung des Modul-Release-Builds spezieller Wert auf die Minimierung von Aufwand und Durchlaufzeit gelegt.

---

[27] Näheres zur Datenschutzproblematik kann unter 4.8 Probleme mit Maven nachgelesen werden.

[28] Siehe 4.4.4 Produkt-Release-Build.

Dieser Build-Prozess basiert wesentlich auf ein Ant gesteuertes Skript welches im Hintergrund Maven aufruft und ein Release-Paket direkt aus einem Snapshot erzeugt. Der Vorteil dieser Methode, gegenüber dem herkömmlichen Release-Build und dem Produkt-Release-Build, ist vereinfachte Bedienung und Zeitersparnis bei der Erstellung.

Eine mögliche Frage beispielsweise eines Entwicklers könnte nun lauten, warum ein Modul-Release-Build nicht auch auf ein Produkt angewendet wird, da dieser einfacher und schneller ist? Diese Möglichkeit besteht durchaus, allerdings wird die Einfachheit dieses Prozesses durch das Streichen verschiedener Sicherungen bei der Softwareerstellung erkauft. Für ein Modul, wie beispielsweise eine Bibliothek, die als Grundlage für andere Softwareprojekte dient, ist dies durchaus vertretbar, da gleichartige Tests bereits in einem Integrationstest durchgeführt werden können. Für ein Produkt, welches an einen Kunden ausgeliefert werden soll, müssen jedoch Qualitätsstandards zur Überprüfung[29] eingehalten werden.

Die folgende Abbildung 6 zeigt den Ablauf eines Modul-Release-Builds. Dieser startet ähnlich wie bei einem Integrations-Build mit dem Auschecken des Quellcodes aus dem SVN-Repository und der entsprechenden Entwicklungstätigkeit. Nach diesen Phasen schließt sich der Maven-Build an, welcher bei erfolgreicher Abarbeitung eine Sicherung im SVN als Tag erstellt. Außerdem liefert der Prozess die fertigen Dateien an das Maven-Repository für Releases aus. Grundsätzlich werden die in diesem Build-Prozess entstandenen Dateien nicht an ein extra Verzeichnis ausgeliefert, wie beispielsweise dem Laufzeit- oder Testumgebungsverzeichnis, da Tests schon im vorhergehenden Integrations-Build vorgesehen waren und die Dateien lediglich zur Weiterverwendung im Maven-Repository bereitgestellt werden müssen und somit den Nutzer nicht direkt erreichen.

[29] Überprüfung der Softwarequalität über Metriken wie PMD, Cobertura, Checkstyle, FindBugs usw.

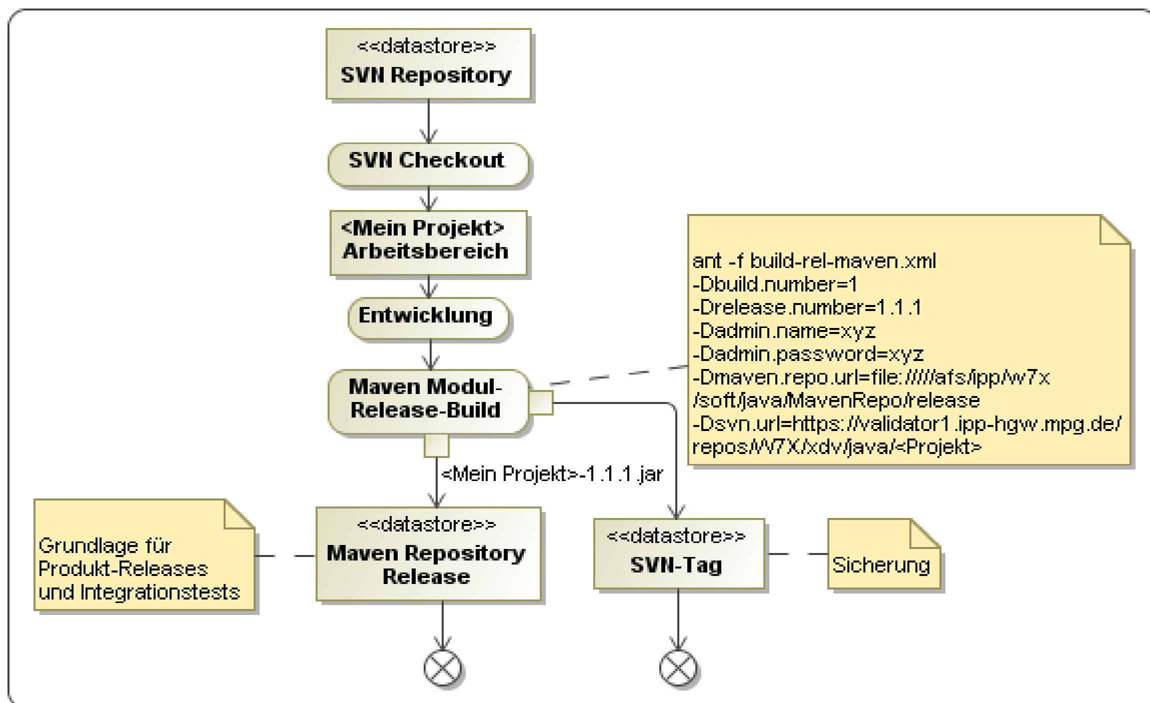

Abbildung 6: Aktivitätsdiagram für einen Modul-Release-Build
(Quelle: Eigene Darstellung)

Die Durchlaufzeiten des Modul-Release-Builds sind durch die Einsparungen relativ kurz und betragen in der Regel etwa drei Minuten, dazu kommt noch, dass das Verfahren gegenüber einem Produkt-Release-Build weniger fehleranfällig ist. Als mögliche Fehlerquellen bei diesem Prozess können fehlende Ablageordner im SVN oder Abhängigkeiten, die noch nicht in releaster Form Vorliegen, im Gegensatz zum Modul-Release-Build auftreten.

Der Aufruf der Modul-Release-Builds geschieht über Ant, dabei werden wie auch schon beim Integrations-Build einige Parameter übergeben wie beispielsweise die zu verwendende Release-Nummer und die zu verwendende Build-Nummer des zugrunde liegenden Integrations-Builds. Ebenso ist es nötig zwei URL-Pfade einzutragen, die zum einen auf das Maven-Repository als Schreibpfad zeigen und zum anderen den Hauptordner des Projekts im SVN angeben, um von dort aus einen Tag im SVN setzen zu können.

4.4.4 Produkt-Release-Build

Der Produkt-Release-Build ist in der Regel der vorläufige Abschluss der Entwicklung in einem Softwareprojekt. Sobald alle nötigen Bibliotheken und Abhängigkeiten in einer releasten Form vorliegen, kann dieser Prozess durchgeführt werden. In der anschließenden

Abbildung 7 wird der schematische Ablauf für ein Produkt-Release-Build anhand eines Aktivitätsdiagrams dargestellt. Der Prozess beginnt ähnlich wie die vorhergehenden Prozesse mit dem Auschecken der Daten aus dem SVN, hierbei bietet sich eine Aufteilung des Trunks (Hauptentwicklungszweig) an, indem im Trunk weiterhin die Neuentwicklungen durchgeführt werden und im neu entstandenen Release-Branch zum größten Teil nur noch Wartungsarbeiten organisiert werden. Durch die Teilung des Release-Branches und des Trunks kann an einem beliebigen Zeitpunkt der Entwicklung unterbrochen und zum Teil parallel entwickelt und gewartet werden.

Nachdem die nötigen Stabilisierungsarbeiten am Release-Branch abgeschlossen worden sind, kann mit dem Produkt-Release-Build fortgefahren werden. Dieser liefert, ähnlich dem Integrations-Build, seine Ergebnisse an das Maven-Repository für Releases und an das neu erzeugte Verzeichnis der Laufzeitumgebung aus. Zudem wird im SVN ein Tag zur Markierung erstellt.

Die Ablage der Daten im Maven-Repository dient einer möglichen Weiterverwendung durch andere Projekte oder der Wiederherstellung von älteren Zuständen. Die .jar Datei, welche an die Laufzeitumgebung ausgeliefert wird, enthält sämtliche abhängigen Bibliotheken, wodurch zwar die Dateigröße ansteigt, aber die benötigten Referenzen des Quellcodes zur Laufzeit des Produktes automatisch mittels einer Manifest-Datei[30] aufgelöst werden können.

Der Produkt-Release-Build ist im Gegensatz zum Modul-Release-Build aufwendiger und eher fehleranfällig, da aus Gründen der Sicherheit und zur Steigerung der Softwarequalität die Anzahl der Tests deutlich höher ist. Dadurch erhöht sich die Wahrscheinlichkeit, dass Fehler auftreten und der Prozess abgebrochen wird.

---

[30] Manifest-Dateien sind in Java Archiven enthalten und speichern Informationen über die zur Laufzeit verwendeten Klassenbibliotheken sowie der Main Methode (Beginn der Abarbeitung einer ausführbaren Java Anwendung).

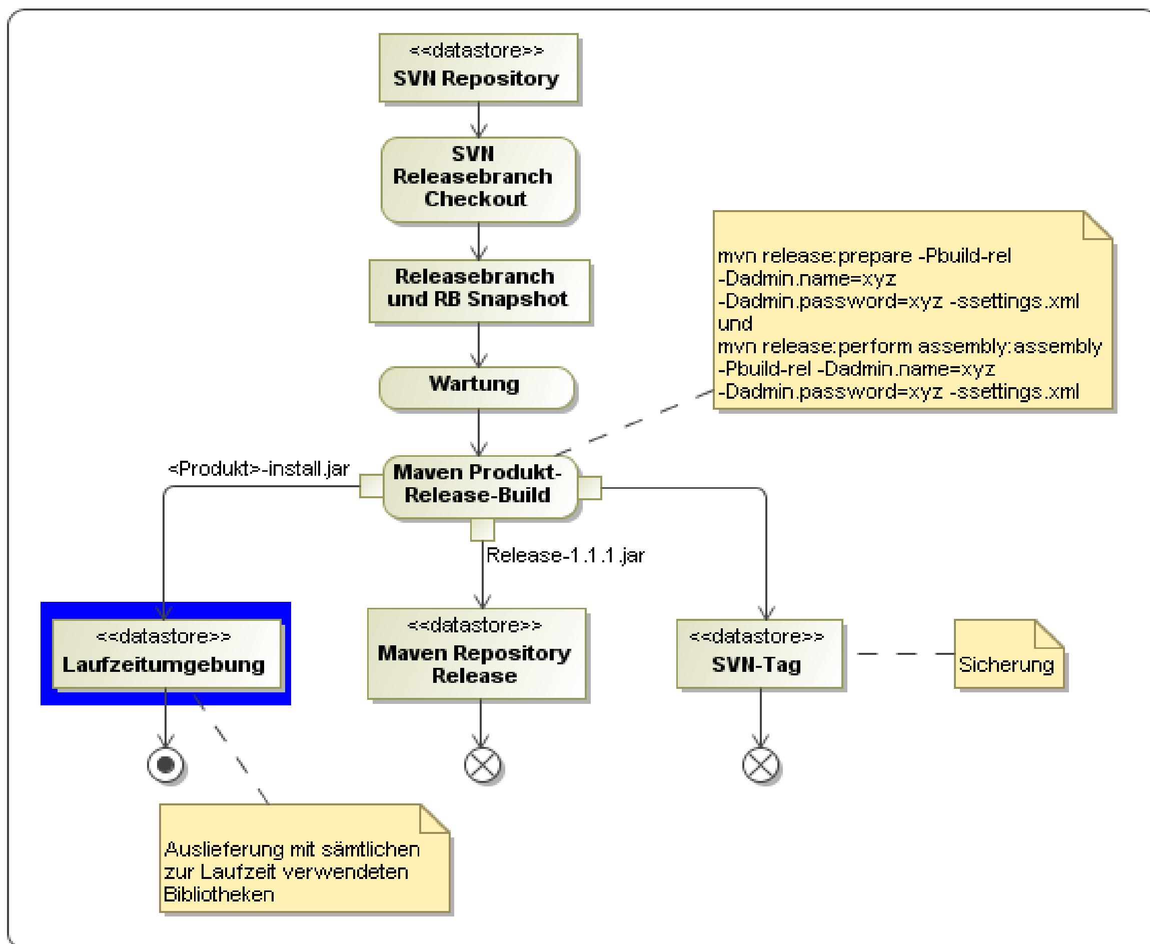

Abbildung 7: Aktivitätsdiagram für einen Produkt-Release-Build
(Quelle: Eigene Darstellung)

Eine durchschnittliche Prozedur dauert zwischen sechs und zehn Minuten, wobei der eigentliche Produkt-Release-Build im Maven-Verfahren zweigeteilt ist. In der ersten Phase wird über das Ziel „release:prepare" das Release vorbereitet, indem Tests durchgeführt werden und per Hand die Nummer des Releases festgelegt werden muss. Im Anschluss wird die Struktur der ursprünglichen POM Datei verändert, es entsteht eine sogenannte effektive POM Datei, die nur noch den Anteil des tatsächlich verwendeten Skripts enthält. Die zweite Phase namens „release:perform" setzt letztlich die überprüften Abläufe des effektiven POM in die Tat um.

Bei einem neu zu erstellenden Release mit einem separaten Release-Branch ist darauf zu achten, dass die benötigten Schreibverzeichnisse für die Build-Ergebnisse und der Release-Branch selbst im SVN angelegt werden müssen, da dieser Vorgang die Vergabe einer Versionsnummer durch den Entwickler oder der Projektleitung vorsieht.

Bei der W7X-CoDaC-Gruppe wurde die sogenannte Laufzeitumgebung auf dem Netzlaufwerk AFS für jeden zugänglich eingerichtet. Die Neuerung dabei besteht zum einen in der Trennung der Entwicklungsumgebung und der Produktivumgebung, zum anderen in der Art der ausgelieferten Dateien. Die ursprünglich ausgelieferten Dateien wurden in Form von .jar Dateien häufig in einem zentralen Ordner abgelegt, dieses Verzeichnis enthielt alle benötigten Bibliotheken der Produkte und war dadurch häufig unübersichtlich[31], allerdings für eine Integrations-Testumgebung durch seine hochgradige Dynamik für die W7X-CoDaC-Gruppe vorteilhaft um Aufwand zu sparen. Durch die Verwendung von Batch-Dateien, wurden die benötigten Bibliotheken zur Laufzeit aufgerufen.
Dieses Verfahren kann im Zuge der Umstellung des Deployments vereinfacht und besser strukturiert werden. Über Maven besteht die Möglichkeit, die durch ein Projekt verwendeten Abhängigkeiten mit in die Auslieferungsdatei zu integrieren, diese müssen lediglich einen automatisch generierten Eintrag in der sogenannten Manifest-Datei bekommen. Durch die Gesamtauslieferung verbessert sich einerseits die Übersichtlichkeit in der Verzeichnisstruktur und andererseits lösen sich mögliche Probleme auf, die häufig entstehen, wenn einzelne für ein Projekt benötigte Abhängigkeiten sich ändern. Somit ist die ausgelieferte Datei nach ihrer Erstellung autonom und kann ohne zusätzliche Bibliotheken verwendet werden.
Zusätzlich kann mittels der Manifest-Datei die entsprechende Main Methode deklariert werden, wodurch die entstandene .jar Datei der Auslieferung zu einem startbaren eigenständig laufenden Programm wird.

4.5 Paketierung und Auslieferung

Die Paketierung und Auslieferung der Ergebnisse des Build-Prozesses können über die Plugin-Konfiguration in der jeweiligen POM des Projektes angepasst werden. Über das Assembly-Plugin besteht die Möglichkeit zusätzlich zu den normalen Ausgabedateien verschiedene Formate wie .zip, .war und .jar zu erstellen und diese zusätzlich mit auszuliefern. Die versendeten Daten können (wie schon in 4.4.4 Produkt-Release-Build beschreiben) über die Manifest-Datei als ausführbare .jar Dateien konfiguriert werden. Zudem können alle benötigten Bibliotheken in diesem Paket mit enthalten sein, wodurch der Zustand nach der Erstellung des Releases unabhängig gegenüber Änderungen wird und somit erheblich weniger störanfällig ist.

[31] Siehe Beispiel in Abbildung 1: Ungeordnete Struktur der Entwicklungs- und Produktivumgebung.

Einige Projekte enthalten Icons oder Grafiken, die häufig erst zur Laufzeit benötigt werden und zum Zeitpunkt des Kompilierens im Build-Prozess keinen Fehler verursachen. Diese Ressourcen müssen über einen gesonderten Schritt über das sogenannte Resource-Plugin in die Auslieferungsdatei kopiert werden.

Im Falle der W7X-CoDaC-Gruppe wurde häufig auf eine zusätzliche Auslieferung beispielsweise von .zip Dateien verzichtet, da diese lediglich mehr Datenmenge produzieren. Deswegen wurden die zum Einsatz kommenden Produkte in nicht komprimierter Form verwendet.

Die derzeitige Auslieferung der Build-Produkte geschieht an das Maven-Repository[32] und je nach Build-Variante an das Verzeichnis der Test- oder Laufzeitumgebung. Alle aufgezählten Auslieferungsorte befinden sich auf dem Netzlaufwerk im AFS. Die Nutzer können direkt auf die Releases zugreifen, die in der Laufzeitumgebung abgelegt sind. Aus Gründen der Ausfallsicherheit und der Zugriffsgeschwindigkeit wurde im Rahmen der Bachelorarbeit eine mögliche Umstrukturierung der Infrastruktur[33] entworfen. Diese sieht im Wesentlichen die Auslagerung der Speicherorte aus dem AFS auf einen neuen Server vor. Zusätzlich sollen auf dem schon bestehenden Hudson-System die automatisch ablaufenden Arbeitsprozesse unter Ant auf Maven umgestellt werden, was dann täglich nach einer Änderung am Quellcode einen Integrations-Build nach sich zieht.

4.6 Versionsverwaltung

Die Versionsverwaltung unter Maven wird zum größten Teil automatisch durchgeführt, es muss lediglich in der POM Datei die Version eingetragen werden mit der ausgeliefert wird. Alle im Laufe eines Integrations-Builds erzeugten Dateien erhalten im Dateinamen einen Anhang mit einer fortlaufenden Nummer. Für gewöhnlich wird die Entwicklung im Trunk nicht nummeriert und bleibt somit auf der Version 0.0.0 stehen.
Sobald die Entscheidung für ein neues Release getroffen wurde, wird standardmäßig in einem Release-Branch stabilisiert und gewartet. Die daraus entstehenden Build-Produkte

---

[32] Siehe Übersicht Abbildung 11: Implementierungsdiagramm der Repositorystruktur der W7X-CoDaC-Gruppe im Anhang 8.6.

[33] Siehe Übersicht Abbildung 12: Implementierungsdiagramm der möglichen Änderungen an der Repositorystruktur im Anhang 8.7.

bekommen dementsprechend eine von Hand festgelegte Nummer der neuen Releases und werden durch Maven mit dem Anhang der Build-Nummer versehen.

4.6.1 Interne und externen Bibliotheken

Eines der zentralen Themen bei der Erstellung einer POM ist die Einbindung von Bibliotheken. Unterschieden wird dabei in interne Bibliotheken, die beispielsweise durch die W7X-CoDaC-Gruppe selbst entstanden sind, und die externen Bibliotheken, welche von Drittanbietern stammen. Interne Bibliotheken lassen sich über Maven sehr komfortabel verwalten, da jedes bereits auf Maven umgestellte Projekt seine erzeugten Dateien unter anderem in das Maven-Repository für Snapshots oder Releases ablegt. Diese Dateien können sehr einfach als Abhängigkeit in einem neuen Projekt eingefügt werden und liefern damit standardmäßig die neuste Version der Bibliothek. Externe Bibliotheken hingegen können über einen Aufruf[34] in das Maven-Repository übertragen werden und stehen dann, ebenso wie interne Bibliotheken, zur Verfügung.

Eine unternehmensweite Umstellung von mehreren Projekten auf Maven benötigt einiges an Zeit. Dieser Prozess profitiert jedoch stark von der Menge der bereits auf Maven umgestellten Projekte. Für den Anfang bietet sich die Möglichkeit Bibliotheken mittels fester Systempfade[35] zu verwenden, diese sind besonders vorteilhaft, wenn nur wenige Dateien im Maven-Repository vorhanden sind, oder für gewisse externe Bibliotheken, die sich nur mit größerem zeitlichen Abstand verändern. Die Folge dieser Systempfade ist eine vorübergehende Plattformabhängigkeit.

4.7 Benutzerfreundlichkeit

Maven zeichnet sich im Gegensatz zu Ant durch eine bessere Benutzerfreundlichkeit aus. Ein Punkt ist hierbei die deklarative Methode, durch die man bei Maven lediglich die gewünschten Ziele angeben muss. Ein weiterer Vorteil liegt in der einfachen Möglichkeit verschiedene Abhängigkeiten in ein Projekt einzubauen, was die Arbeit ebenfalls erleichtert. Neue Projekte können mit geringem Zeitaufwand umgestellt werden, wobei große

---

[34] Siehe 8.4 Einbindung externer Bibliotheken in das Maven-Repository.

[35] Siehe 8.5 Abhängigkeiten in der Eclipse-IDE (Systempfade).

Teile, in einer standardisierten Entwicklungsumgebung, einer bereits verwendeten POM, übertragen werden können.
Die Arbeit mit Maven wird zudem noch durch eine IDE-Unterstützung[36] vereinfacht, denn diese erleichtert die Einbindung von Abhängigkeiten und die Einstellungen beispielsweise von Lese- und Schreibpfaden.

Die größten Vorteile in puncto Benutzerfreundlichkeit kommen den Entwicklern zu gute, denn sie Profitieren auch von einem geringen einmaligen Umstellungsaufwand von Projekten grade in Bezug auf das vorher verwendete Ant. Aber auch die Nutzer der entstandenen Software haben einen nennenswerten Vorteil aus der Umstellung auf Maven. Sie erhalten durch die seit längerem gewünschte Trennung der Produktiv- von der Entwicklungsumgebung eine höhere Stabilität der Software. Bei den Programmen, welche im Dauerbetrieb laufen, kommt es somit nach einem Softwarewechsel nicht mehr zu einem Absturz und damit verbundenem Datenverlust, da ein Wechsel der Software nur noch zum Zwecke eines neuen Releases nötig wird und im Rahmen eines koordinierten Verfahrens durchgeführt werden kann.

4.8 Probleme mit Maven

In den bisherigen Kapiteln der Arbeit wurden besonders die positiven Eigenschaften und die Verbesserungen durch Maven hervorgehoben, speziell in Bezug auf die Anforderungen der W7X-CoDaC-Gruppe. Im nun folgenden Abschnitt werden einige bekannte und aufgetretene Probleme bei der praktischen Arbeit mit Maven beleuchtet.

Auffällig sind zuallererst eine eher dürftige Dokumentation von Maven und vor allem dessen Plugins. Die Dokumentationen enthalten zum Teil nur die notwendigsten Grundlagen sowie Befehle, die nicht ausreichend erklärt werden. Dies ist vor allem hinderlich, wenn spezielle Anpassungen an den Maven-Ablauf erforderlich werden.

Beim Editieren von POM Dateien ist es ratsam, wenn ein bereits funktionierender Zustand erreicht worden ist, diesen möglichst nach wenigen Änderungen an der POM zu überprüfen. Das liegt an der teilweise ungenauen Fehlerausgabe durch Maven. Beispielsweise führen gewisse leere Ausdrücke in einer POM Datei (besonders wenn es sich dabei um die

[36] Für weitere Informationen zur IDE Unterstützung siehe 4.1.1.2 Installation von Maven.

Auslieferungsverzeichnisse handelt) zu einer Fehlermeldung, die gänzlich irreführend wirkt, da der Entstehungsort des Fehlers nicht angegeben ist. Derartige Probleme traten regelmäßig bei der Verwendung des bei der W7X-CoDaC-Gruppe eingesetzten M2Eclipse-Plugins auf. Das hat bereits beim Öffnen des Einstellungsmenüs für die Schreibverzeichnisse und Repository-Einträge einen leeren Eintrag in der POM hinterlassen, wodurch ein Fehler nach dem Speichern und Ausführen aufgetreten ist. Andere leere ausdrücke wie beispielsweise <site> Einträge führen nicht zu Fehlern.
Dennoch ist das M2Eclipse-Plugin eine sinnvolle Erweiterung bei der Verwendung von Maven, besonders weil der Umgang mit den POM Dateien stark vereinfacht wird. Beispielsweise erleichtert es im Gegensatz zu herkömmlichen Editoren die Übersichtlichkeit der POM, vor allem da diese mitunter 400 Zeilen und länger werden können.

Anfänglich wurde in der W7X-CoDaC-Gruppe ausschließlich unter Windows mit Maven gearbeitet. Das änderte sich mit dem Wunsch die Maven-Abläufe zeitlich gesteuert und automatisch über einen Hudson-Service laufen zu lassen. Der Hudson läuft derzeit auf einem auf Unix basierenden Betriebssystem. Allgemein bekannt ist die Problematik der Schrägstrichkonventionen beider Betriebssysteme, beispielsweise um FTP und HTTP Pfade festzulegen. Die Lösung dieses Problems besteht in der generellen Verwendung von Schrägstrichen wie sie unter Unix verwendet werden, da diese von beiden Betriebssystemarten unterstützt werden. Dabei bleibt die Menge der Zeichen gleich und hängt lediglich vom entsprechenden Aufruf[37] ab.

Die W7X-CoDaC-Gruppe hat eine Möglichkeit zur Datensicherung und Wiederherstellung in der Anforderungsanalyse gefordert, dies birgt allerdings bei Verwendung von Maven den Nachteil, dass es keine Säuberungsfunktion oder dergleichen gibt, was wiederum bedeutet, dass sich Unmengen von Daten im Laufe der Zeit ansammeln. Jeder Build-Prozess hinterlässt fast alle Daten des jeweiligen Projektes, zum einen im Quellcode-Repository (SVN-Tag), zum anderen im Maven-Repository. Zugegebenermaßen ist heutzutage, durch die Entwicklung der Technik beispielsweise im Serverbereich, die Menge an Speicherplatz nicht mehr eines der zentralen Themen. Dennoch ist es empfehlenswert gelegentlich überflüssige beziehungsweise veraltete Daten, Snapshots und Build-Produkte in den Repositorys zu löschen.

[37] Beispiel: FTP://///afs/ipp/w7x/...

Eine weitere Problematik ist durch die Markierung der Build-Prozesse im SVN als Tag entstanden. Jeder Build-Prozess verwendet ein Ant-Skript, wodurch die Markierung erstellt wird. Damit dies durchgeführt werden kann, ist es bisher nötig gewesen mittels Parameter das Password und den Namen eines berechtigten Nutzers zu übermitteln (stellvertretend kann auch das Benutzerkonto einer beliebigen anderen Person mit Zugang zum SVN verwendet werden).
Wie bereits in 4.4.2 (Integrations-Build) erwähnt ist diese Methode der Passwordübergabe denkbar ungünstig und praktisch nur zu Testzwecken verwendbar. Als mögliche Lösung kann für die W7X-CoDaC-Gruppe eine Art Super User eingerichtet werden, dieser dient dann stellvertretend für den Abgleich der Daten mit dem SVN.

5 Stand der Entwicklung

In diesem Kapitel sollen die Aufgaben für die Arbeit bei der W7X-CoDaC-Gruppe aus der Anforderungsanalyse zusammengefasst und kurz deren Umsetzung beschrieben werden.

Die Wiederherstellbarkeit von erzeugten Produkten und die damit einhergehende Bestrebung zur Minimierung von Ausfallzeiten, sowie eine geordnete Ablagestruktur der Build-Produkte samt Versionsverwaltung, ließen sich durch die Verwendung von Repositorys unter Maven realisieren.
Die eingerichteten Repositorys wurden dabei in Dateien, die zur Auslieferung dienen, wie das Snapshot- und Release-Repository und in Lib- sowie Plugun-Repository für Quell- bzw. Systemdateien, welche die Herstellung von Auslieferungsdateien unterstützen, aufgeteilt.
Ein weiterer Punkt war die Schaffung einer stabilen Produktivumgebung, in der es eine Trennung der Entwicklungs- von der Produktivumgebung gibt. Ebenso sollte ein Verfahren zur Auslieferung von Produkten und Modulen entstehen. Die beiden Punkte wurden durch die Einführung von Produkt- und Modul-Release-Builds verwirklicht.
In der Anforderungsanalyse wurde ebenfalls ein automatisierter Integrations-Build und teilautomatisierte Release-Builds gefordert, dies konnte anhand von prototypischen Umsetzungen einzelner Build-Prozesse demonstriert werden. Auf längere Sicht wurde ein Vorschlag zu Veränderung der Infrastruktur[38] erarbeitet, wobei die neu entwickelten Build-Prozesse unter Maven die ursprünglichen Verfahren mit Ant komplett ablösen können.
Ein weiteres Thema der Anforderungsanalyse, das nicht außer Acht gelassen werden sollte, besteht aus einer möglichst guten Benutzerfreundlichkeit, zum einen für die Nutzer und zum anderen für die Entwickler der entstandenen Software. Sie können dabei schnell und einfach ihre Projekte auf Maven umstellen. Die Entwickler und vor allem die Nutzer der entwickelten Software profitieren von der Trennung der Entwicklungs- und Produktivumgebung, da die verwendeten und betriebenen Programme der Nutzer bei einem Softwarewechsel zu Testzwecken nicht mehr täglich und ohne Ankündigung, sondern nur noch bei neuen Releases verändert werden. Die Folgen ohne dieses System waren für gewöhnlich

---

[38] Siehe Abbildung 11: Implementierungsdiagramm der Repositorystruktur der W7X-CoDaC-Gruppe und Abbildung 12: Implementierungsdiagramm der möglichen Änderungen an der Repositorystruktur.

Softwarefehler und -abstürze. Insofern haben sich die Umstellung der Abläufe und die Verwendung von Maven positiv auf die W7X-CoDaC-Gruppe ausgewirkt.
Alle Punkte der Anforderungsanalyse und die Unterscheidung der Produktarten nach deren Auslieferungszielorten[39] wurden somit erfüllt. Als ein ergänzender Punkt wäre noch die Umsetzung für Eclipse OSGI Bundles[40] zu nennen. Allerdings funktioniert dieses Verfahren bisher noch nicht, da diese Technik noch nicht ausgereift ist und konkurrierende Konzepte enthält, wie beispielsweise verschiedene Definition von Abhängigkeiten.

Das durch diese Bachelorarbeit erarbeitete Wissen wurde dem Entwicklerteam der W7X-CoDaC-Gruppe im Rahmen einer umfangreichen Präsentation zum Thema „Verteilung von Experiment-Software für Wendelstein 7-X“ vorgestellt, wodurch die Vorteile einer Umstellung verdeutlicht wurden. Zukünftig werden deshalb die entworfenen Maßnahmen verwirklicht.

[39] Siehe Tabelle 1: Produktarten des Release-Prozesses.

[40] Zeitgleich wurde im Rahmen einer Masterarbeit für die W7X-CoDaC-Gruppe ein Verfahren zur Erzeugung von OSGI Bundles mit Maven getestet, jedoch ohne den Einsatz von Maven-Repository.

6 Ausblick

Das erworbene und dokumentierte Wissen in Form von Schaubildern und UML-Diagrammen, sowie die beschriebenen Abläufe der Build-Prozesse und der Umstellung auf Maven, können die Grundlage bilden für diverse andere Softwareprojekte in Unternehmen oder in forschenden Einrichtungen, die Planen eine Deployment-Struktur einzurichten oder diese zu verbessern. Software-Deployment wird derzeit eher selten in der Fachliteratur thematisiert, wodurch der Eindruck entsteht das Deployment eine Trivialität ist. Die Erfahrung in Bezug auf diese Bachelorarbeit hat bewiesen, dass dies nicht zutrifft und im Falle der W7X-CoDaC-Gruppe im Vorfeld zu Problemen geführt hat.

Derzeit wird häufig noch auf das weit verbreitete Deployment-Werkzeug Ant zurückgegriffen. Im Rahmen dieser Bachelorarbeit konnte gezeigt werden, dass auch Maven geeignet ist Deployment-Prozesse zu unterstützen und darüberhinaus noch weitere Vorteile mit sich bringt. Im Bezug auf die Verwendung bei der W7X-CoDaC-Gruppe hatte besonders die Verwendung von Repositorys eine nachhaltig positive Wirkung.

Für die Zukunft des Deployments bei der W7X-CoDaC-Gruppe sind einige Veränderungen vorstellbar. Beispielsweise wäre es sinnvoll die Auslieferungs-Repositorys sowie die Verzeichnisse der Test- und Laufzeitumgebung auf verschiedene Server aufzuteilen, da dies derzeit noch nicht der Fall ist, könnte ein Serverausfall den Produktivbetrieb stark beeinträchtigen und eine Wiederherstellung erschweren. Damit bei einem Serverausfall der Produktivbetrieb reibungslos fortgesetzt oder zumindest aufgrund der Quelldaten schnellstmöglich wieder in Betrieb genommen werden kann, sollten die Daten möglichst auf verschiedenen Servern gelagert werden.
Durch die positive Resonanz des Entwicklerteams der W7X-CoDaC-Gruppe auf die Präsentation und die Vorstellung des Projektes, ist mit einer flächendeckenden Umstellung der bestehenden Projekte auf Maven zu rechnen. Dazu kommt noch, dass die umgestellten Maven-Projekte restlos in das Hudson-System integriert werden um automatische Build-Prozesse und deren Überwachung durchzuführen.
Die Auslieferung von Releases an die Nutzer könnte möglicherweise verbessert werden, indem eine Strategie mit festen Release-Zyklen beziehungsweise Release-Terminen, an denen sämtliche Produkte ausgeliefert werden sollen, vorgeschrieben werden.

7 Verzeichnisse

7.1 Quellenverzeichnis

Apache Ant 1.7.1 Manual: Introduction, URL http://ant.apache.org/manual/index.html (abgerufen am 02.07.2009)

Balzert, Helmut: Lehrbuch der Software-Technik (Software-Entwicklung), 2. Auflage, Spektrum Akademischer Verlag, 2000

Hudson: Meet Hudson, URL http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson (abgerufen am 1.11.2009)

IPP: Wendelstein 7-X Meilensteine, URL http://www.ipp.mpg.de/de/for/projekte/w7x/for_proj_w7x_meilensteine.html (abgerufen am 7.8.2009)

Isele, Stefan : Eclipse versus Maven, In: eclipse Magazin, Heft 4 (April 2009), S. 68-74, Software & Support Verlag GmbH

Kühner, Dr. Georg: Verteilung von Experiment-Software am W7X,
Dok.-Kennz.: -F0004.0, KKS.Nr.: 1-NCV, 07.05.2009

Lang Code Project: Install an external jar into local Maven repository, URL http://jeff.langcode.com/archives/27 (abgerufen am 25.10.2009)

Popp, Gunther: Konfigurationsmanagement mit Subversion, Ant und Maven (Ein Praxis-handbuch für Softwarearchitekten und Entwickler), 2. Auflage, dpunkt.verlag, 2008

Wikipedia: Deployment, URL http://de.wikipedia.org/wiki/Deployment (abgerufen am 15.07.2009)

7.2 Abbildungsverzeichnis

Abbildung 1: Ungeordnete Struktur der Entwicklungs- und Produktivumgebung 17
Abbildung 2: SVN-Tag von CodacUtil 20
Abbildung 3: Hierarchiestruktur von Maven für Confix (Dependency Graph) 21
Abbildung 4: Aktivitätsdiagram der Reihenfolge für Build-Prozesse................................ 22
Abbildung 5: Aktivitätsdiagram für einen Integrations-Build 25
Abbildung 6: Aktivitätsdiagram für einen Modul-Release-Build 28
Abbildung 7: Aktivitätsdiagram für einen Produkt-Release-Build 30
Abbildung 8: Ausschnitt des SVN-Repositorys zu DbUtil 44
Abbildung 9: Privates Maven-Repository für Build-Produkte... 45
Abbildung 10: Abhängigkeiten in der Eclipse-IDE 47
Abbildung 11: Implementierungsdiagramm der Repositorystruktur der W7X-CoDaC-Gruppe 48
Abbildung 12: Implementierungsdiagramm der möglichen Änderungen an der Repository-struktur... 49

7.3 Tabellenverzeichnis

Tabelle 1: Produktarten des Release-Prozesses... 9
Tabelle 2: Lifecycle-Phasen für den Datentyp .jar 24

7.4 Glossar

7.4.1 Begriffe und Definitionen

AFS - bedeutet Andrew File System und ist ein verteiltes Dateisystem. Unter MS-Windows erscheinen AFS-Volumes als „Netzlaufwerke“, unter Unix als extern gemountete Verzeichnisse.

Ant - erstellt aus dem Quellcode ausführbare Programme, ähnlich wie Make.

Deployment - bezeichnet die Auslieferung und Verteilung sowie die Installation von Software (vgl. Wiki 2009).

Hudson - ist ein Service der fortlaufend je nach Update eines Softwareprojekts diese neu ausliefert, außerdem eignet es sich auch zum Testen und Überwachen von Software (vgl. Hudson 2009).

Metrik - eine Metrik beschreibt eine Maßzahl zur Qualitätsabschätzung von Software, häufig auf Grundlage mathematischer Formeln. Beispiele für Metriken sind die Anzahl der Codezeilen, die Funktion-Point-Analyse oder die Berechnung der Zyklomatischen-Zahl.

Modul - bezeichnet Softwareprojekte einer niedrigen Abstraktionsebene, die die Grundlage für eines oder mehrere Produkte bilden.

Produkt - sind Softwareprojekte einer höheren Abstraktionsebene. Sie verwenden Module (teilweise auch andere Produkte) und werden an den Endbenutzer ausgeliefert.

Repository – ist ein Verzeichniss zur Speicherung und Verwaltung von Daten. Es dient beispielsweise der Versionsverwaltung (CVS und SVN) von Quellcodedateien oder wie im Fall von Maven vor allem der Lagerung von Programmpaketen und deren Metadaten.

Tag - ist eine Markierung in einem SCM-System und spiegelt einen Zustand zu einem Zeitpunkt wieder, desweiteren wird es zur Sicherung und Wiederherstellung verwendet.

Trunk - ist der Hauptentwicklungszweig in einem SCM-System bei der Softwareentwicklung.

7.4.2 Abkürzungen

bat	Stapel- oder auch Batchverarbeitung
CoDaC	Control, Data Acquisition and Communication
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IPP	Institut für Plasma Physik
JDK	Java Development Kit
POM	Project Object Model
sh	eine Shell unter Unix
SCM	Sourcecode Management System
SVN	Subversion-Repository
W7X	Wendelstein 7-X

8 Anhang

8.1 Ausschnitt des SVN-Repositorys zu DbUtil

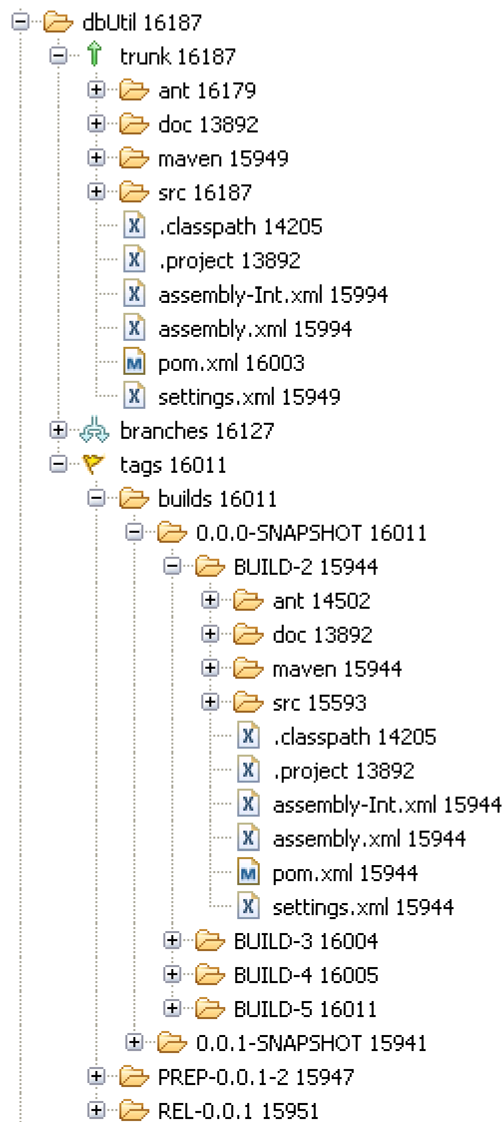

Abbildung 8: Ausschnitt des SVN-Repositorys zu DbUtil

(Quelle: Eigene Darstellung)

8.2 Privates Maven-Repository für Build-Produkte

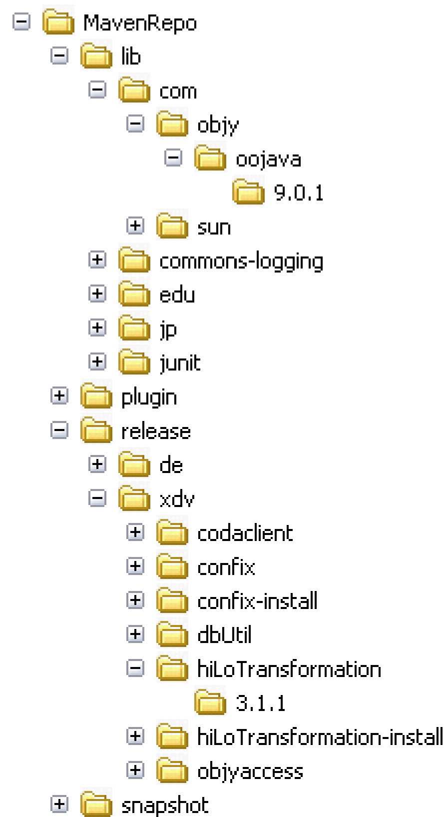

Abbildung 9: Privates Maven-Repository für Build-Produkte

(Quelle: Eigene Darstellung)

8.3 Umgebungsvariablen für Maven

Unter Windows Betriebssystemen können die Variablen in den Systemeigenschaften (Windowstaste + Pause) unter „Erweitert“ bei Umgebungsvariablen nachgesehen werden. Falls die Benutzervariable „M2_HOME“ mit dem Wert „C:\...\apache-maven-2.2.0“ und die Benutzervariable „M2“ mit dem Wert „%M2_HOME%\bin“ nicht schon vorhanden sind, müssen sie ergänzt werden[41]. Um Maven über die Kommandozeile aufrufen zu können, muss bei den Benutzervariablen unter „Path“ der Eintrag „%M2%“ gesetzt werden, falls es schon Path-Einträge gibt, müssen diese mit einem Semikolon abgetrennt werden.
Da Maven auf Java basierende Anwendung ist, muss sichergestellt werden, dass die Umgebungsvariable namens „JAVA_HOME“ und der entsprechende Path-Eintrag „%JAVA_HOME%\bin“ vorhanden sind und auf eine möglichst neue Version des JDK (Java Development Kit) zeigt. Die aktuelle Anforderung der Java-Version von Maven kann auf der Herstellerseite nachgelesen werden.

[41] Hinweis: die Benutzervariablen wurden nur zur Verdeutlichung in Anführungszeichen gesetzt, beim Selbsterstellen müssen diese weggelassen werden.

8.4 Einbindung externer Bibliotheken in das Maven-Repository

Der Kommandozeilenaufruf zum Einbinden externer Bibliotheken in das eigene Maven-Repository kann folgendermaßen aussehen:

mvn install:install-file -DgroupId=<your_group_name>
-DartifactId=<your_artifact_name>
-Dversion=<snapshot>
-Dfile=<path_to_your_jar_file>
-Dpackaging=jar
-DgeneratePom=true
(Lang Code Project 2009)

8.5 Abhängigkeiten in der Eclipse-IDE

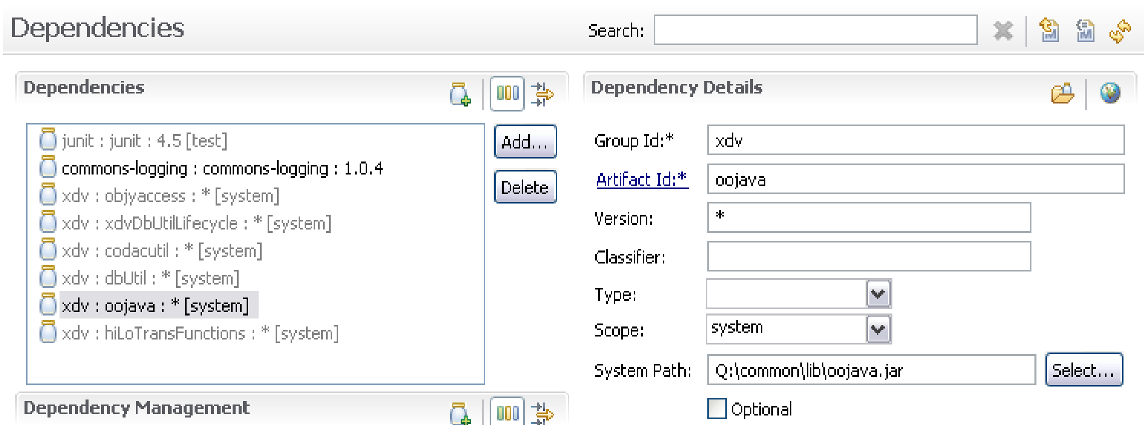

Abbildung 10: Abhängigkeiten in der Eclipse-IDE
(Quelle: Eigene Darstellung)

8.6 Repositorystruktur der W7X-CoDaC-Gruppe

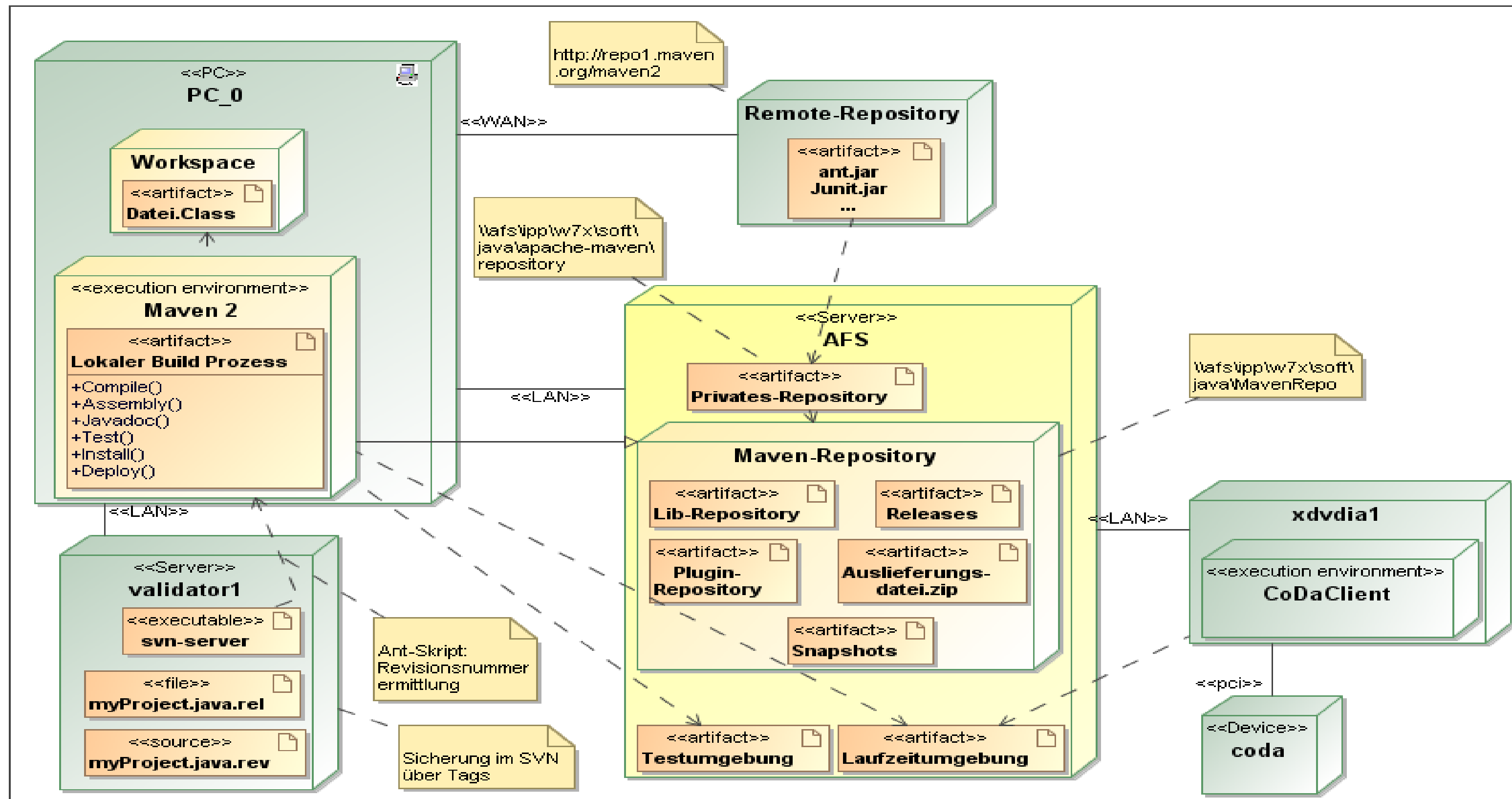

Abbildung 11: Implementierungsdiagramm der Repositorystruktur der W7X-CoDaC-Gruppe

(Quelle: Eigene Darstellung)

8.7 Vorschlag für Änderungen an der Repositorystruktur

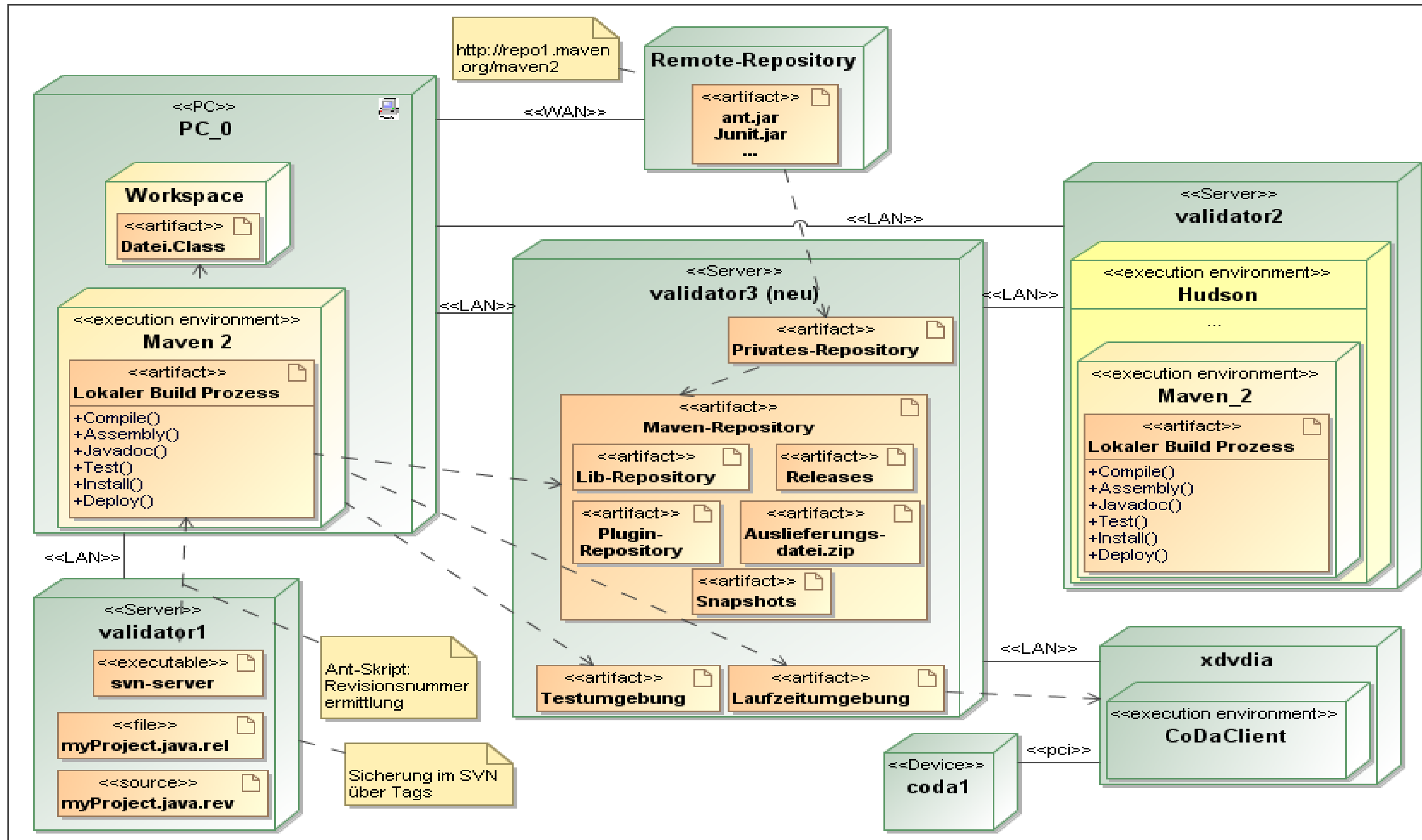

Abbildung 12: Implementierungsdiagramm der möglichen Änderungen an der Repositorystruktur

(Quelle: Eigene Darstellung)

Selbstständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel sind angegeben. Die Arbeit hat mit gleichem bzw. in wesentlichen Teilen gleichem Inhalt noch keiner Prüfungsbehörde vorgelegen.

____

Ort, Datum, Unterschrift