FAKULTÄT FÜR !NFORMATIK

Projektautomatisierung und Build-Prozess

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des

Diplomstudium Informatik

ausgeführt von

Hasan Kılıç
Matrikelnummer: 9727056

am:

Institut für Gestaltungs- und Wirkungsforschung

Betreuung:

Betreuerin: Assoc. Prof. Dipl.-Ing. Dr. Hilda Tellio lu

Wien, 27.10.2008 _____

_____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Danksagung

Allen voran möchte ich mich bei meiner Betreuerin Frau Assoc. Prof. Dipl.-Ing. Dr. Hilda Tellio ǀu für Ihre fortlaufenden Bemühungen während der Abfassung dieser Diplomarbeit bedanken. Ohne Ihre kompetente Unterstützung wäre die planmäßige Fertigstellung meiner Arbeit nicht möglich gewesen.

An dieser Stelle gilt mein Dank auch meinen lieben Eltern und meiner Semra, die mich im Laufe meines Studiums tatkräftig unterstützt haben und mir stets motivierend zur Seite gestanden sind.

Abstract

Die Automatisierung des Build-Prozesses ist eine besonders wichtige und empfehlenswerte Vorgehensweise, die im Zuge jedes Softwareentwicklungsprozesses stattfinden sollte. Doch bevor es überhaupt möglich ist, dass sich Projektteams dieser Thematik zuwenden, muss vorerst die korrekte Gestaltung und Strukturierung des Build-Prozesses erfolgen, sodass man sich anschließend Gedanken über die Implementierung und Automatisierung des Prozesses machen kann. Um ein Softwareprodukt erstellen zu können, muss eine Kette von bestimmten Schritten bzw. Phasen aufeinanderfolgend ausgeführt werden, die im Build-Prozess zusammengefasst bzw. vereint werden. Insofern ist es wichtig zu wissen, aus welchen Teilen der Build-Prozess besteht und wie er aufgebaut ist. Außerdem sollten sich AnwenderInnen darüber im Klaren sein, was sie sich vom Build-Prozess erwarten und zu welchem Zweck sie ihn einsetzen.

Diese Diplomarbeit widmet sich der ausführlichen Untersuchung des Build-Prozesses, mitsamt all seinen Einzelheiten und Bestandteilen. Dabei werden jene Technologien, Werkzeuge und Entwicklungsmethoden in die Betrachtung mit einbezogen, die mit diesem Prozess in direktem Zusammenhang stehen.

Die Untersuchung hat gezeigt, dass der automatisierte Build-Prozess im Vergleich zu manuell gesteuerten Builds die effektivere und weniger fehleranfällige Variante ist. Daneben ließen sich eine Reihe von Vorteilen, wie beispielsweise der Wiederverwendbarkeit, Reproduzierbarkeit und Skalierbarkeit, erkennen, die mittels Automatisierung bestimmter Prozessaktivitäten genossen werden können. Außerdem ist ein deutlicher Trend Richtung vollständiger Automatisierung des Build-Prozesses erkennbar, der vor allem damit einhergeht, dass neue Entwicklungsmethoden, wie das Agile Development, einen besonders großen Wert auf automatisierte Builds legen.

Abstract

The automation of the build process is a very important and highly recommended strategy that should take place in the course of each software development process. Therefore it constitutes a great necessity for any project team to concentrate on the accurate configuration and structuring of the build process. In addition it is particularly advantageous to realize an early implementation and automation of the build process. There is a string of some specified steps that must be followed back to back in order to *build* a software product. Hence it is an essential issue to know how the build process is constructed and what its constituents are.

The present diploma thesis focuses on the *build process*, along with all its particulars and constituent parts. Withal, a vital part of the study will be about technologies, tools and development methods that make a significant impact on the build process.

The analysis of the mentioned issues has shown that the full automated build process is much more effective and less error-prone than a manually controlled build process. Automation also brings along a range of advantages, such as reusability, reproducibility, accessibility, scalability, etc. Additionally, in recent years a trend towards full automated builds originated due to new development methods (e.g. Agile Development), that set great value upon automation.

Inhaltsverzeichnis

1. Einleitung

Der Build-Prozess stellt eines der wichtigsten Elemente für die Softwareentwicklung dar, sodass jede Person, die sich mitunter an der Produktion von Software beteiligt hat, mit dem Begriff *Build* vertraut sein wird. Obwohl es sich um eines der geläufigsten Termini in der Softwarebranche handelt, wird es dennoch nicht selten falsch ausgelegt. Der Build-Prozess kann nicht einfach auf die Kompilierung reduziert werden, denn diese stellt nur einen Teil des Ganzen dar. Ein Build wird letztendlich nicht nur zur Erstellung eines Produkts ausgeführt, sondern leistet daneben auch weitere beachtliche Dienste. Insofern wäre hier als Beispiel der Nutzen des Integrations-Builds anzusprechen, bei dem nicht nur, von verschiedenen EntwicklerInnen angefertigte, individuelle Softwarestücke zusammengefügt werden, sondern nach und nach auch Gedanken und Lösungsansätze der ProjektmitarbeiterInnen miteinander verschmelzen und spezifische Herangehensweisen sowie Arbeitsmethoden aneinander angepasst werden (vgl. Lee 2008).

Ein gut strukturierter und durchdachter Build-Prozess ist der Grundstein für die erfolgreiche Softwareentwicklung, aber nicht jedes Unternehmen ist sich dieser Tatsache bewusst, sodass es umso wichtiger ist, bereits frühzeitig, in der Anfangsphase des Projekts, Überlegungen zu der Struktur und den Eigenschaften des Builds und seiner zeitgerechten Implementierung anzustellen.

Daher ist es Ziel dieser Arbeit, die Wichtigkeit der frühzeitigen Implementierung und Automatisierung eines Build-Prozesses und seiner einzelnen Schritte zu betonen und darzustellen, welche Maßnahmen diesbezüglich getroffen werden können. Eine wichtige Aufgabe besteht darin, die Anforderungen an einen automatisierten Build-Prozess zu veranschaulichen und darzulegen, welche Eigenschaften ein Build diesbezüglich zu erfüllen hat. Die Ergebnisse und Vorteile, die sich aus dieser Untersuchung ergeben, werden einmal mehr den Nutzen eines automatisierten Build-Prozesses in der Softwareentwicklung verdeutlichen. Als Vergleichspunkt werden an geeigneter Stelle auch die Auswirkungen und Nachteile eines manuell gesteuerten Build-Prozesses thematisiert, und zusätzlich angemerkt, welche möglichen Konflikte daraus resultieren können.

Dieser Arbeit liegt eine intensive und umfassende Literaturrecherche zu Grunde, bei der neben der Standardliteratur auch aktuelle wissenschaftliche Untersuchungen und Artikel aus dem World-Wide-Web berücksichtigt und miteinbezogen wurden.

Die vorliegende Arbeit gliedert sich in fünf Kernbereiche und wird in Kapitel 2 mit einer Definition und den zentralen Grundlagen des Build-Prozess eröffnet. Dieses einleitende Kapitel enthält eine allgemeine Definition der verwendeten Begriffe, sowie eine kurze Darstellung zu den Unterschieden des manuellen und automatisierten Build-Prozesses. Im Anschluss daran werden die Typen und speziellen Eigenschaften automatisierter Builds

aufgeschlüsselt und das Kapitel schließlich mit der Abhandlung zur Architektur des Build-Prozesses, den spezifischen Rollenverteilungen im Build-Management, der Bedeutung eines geregelten Build-Rhythmus und der exemplarischen Darlegung des Build-Prozessplans abgerundet.

Das dritte Kapitel beschäftigt sich mit jenen Werkzeugen, die zur Umsetzung des Build-Prozesses herangezogen werden. Diesbezüglich wurden die Build-Tools Make, Ant und Maven ausgewählt, die für unterschiedliche Probleme und Anforderungen im Build-Prozess verschiedene Lösungsansätze liefern und unter anderem die Automatisierung bestimmter Arbeitsbereiche befähigen. Die Darstellung der Funktionsweise sowie Vor- und Nachteilen der einzelnen Tools, wird deren aktuelle Bedeutung in der Softwareentwicklung, sowie Gründe für die Beliebtheit und den bevorzugten Einsatz bestimmter Tools im Gegensatz zu Anderen nahe bringen.

Einen wichtigen Teil dieser Arbeit stellt das vierte Kapitel dar, in dem die einzelnen Teilprozesse bzw. Phasen, aus denen sich ein typischer Build-Prozess zusammensetzt, theoretisch aufgearbeitet und ausführlich geschildert werden. An dieser Stelle werden Ablauf und Aufgaben der einzelnen Teilprozesse durchleuchtet und ihr Stellenwert innerhalb des Build-Prozesses herausgearbeitet. Daneben wird explizit auf die vorteilhaften Aspekte der Automatisierung dieser (Teil-)Prozesse hingewiesen.

Die inhaltliche Abrundung der Arbeit findet in den letzten beiden Kapiteln statt, die sich den verschiedenen Build-Varianten und Developmentmethoden zuwenden. Dabei wird aufgeklärt, welche Build-Variante in welcher Situation bzw. zu welchem Zeitpunkt im Build-Prozess eingesetzt wird und warum es erforderlich ist, dass der Build den Anforderungen des Projekts und den Bedürfnissen der ProjektmitarbeiterInnen angepasst wird. Hinzu kommen die Developmentmethoden, die im Wesentlichen die Gestaltung und Ausführung des Build-Prozesses beeinflussen, sodass es wichtig ist, auch deren Besonderheiten und ihre spezifischen Auswirkungen auf die Softwareentwicklung zu erläutern.

Das Schlusswort der Arbeit wird mit abschließenden Bemerkungen zum untersuchten Gegenstand und der rückblickenden Auflistung der wichtigsten Ergebnisse gesetzt.

2. Grundlagen

Bevor eine vertiefende Auseinandersetzung mit der Themenstellung dieser Diplomarbeit stattfindet, wird eine allgemeine Definition der verwendeten Begriffe vorangestellt. Im Anschluss daran werden die verschiedenen Build-Strategien und spezielle Eigenschaften des automatisierten Build-Prozesses zu behandeln sein. Um einen grundlegenden Einblick in die Materie zu verschaffen, widmet sich dieses Kapitel auch der Darstellung der Build-Architektur, den Rollen im Build-Management und dem Build-Prozessplan.

2.1 Begriffsdefinition

„Ein Build-Prozess ist nichts weiter als eine Reihe von Schritten, die Ihre kreativen Artefakte in auslieferbare Software transformieren“ (Clark 2006, S. 12).

Diese Schritte werden in folgender Definition näher beleuchtet (it-agile.de, Automatisierter Build-Prozess): „Unter dem Begriff Build-Prozess werden alle Aktivitäten zusammengefasst, die für die Produktion und Bereitstellung von lauffähiger Software notwendig sind. Dazu gehört das Kompilieren des gesamten Systems, das Testen der erstellten Software, das Einrichten der Datenbank, die Migration bestehender Daten, das Erstellen der Dokumentation in elektronischer Form, das Deployment auf einem Applikationsserver und ggf. projektspezifisches Weiteres.“

In Berufung auf vorangehende Definition muss das zustande kommende Produkt auslieferbar sein, die Kriterien der Qualitätssicherung erfüllen, einem Test unterzogen und der gesamte Prozess dokumentiert worden sein. Die Definitionen von Cymerman, sowie Breu, Matzner und Nickl (2005) betonen zudem die Wichtigkeit des Build-Prozesses als verbindendes Element zwischen den verschiedenen Arbeitsschritten innerhalb des Softwareentwicklungszyklus (2000). So beschreibt Cymerman den Build-Prozess als die Verbindung zwischen Entwicklung, Integration, Test und Produktion.

Breu, Matzner und Nickl liefern eine ähnliche Definition zum Build-Prozess (2005, S. 222): „Der Build-Prozess ist ein wesentlicher Teil jedes interaktiven Entwicklungsprozesses, da er die Lücken zwischen Entwicklung, Integration und Produktion schließt. Er vereinfacht und beschleunigt die Migration zwischen verschiedenen Umgebungen und vereinheitlicht die Verwendung von Compilern, Testtools und Middleware.“

Obwohl der Aufbau eines Build-Prozesses stark von der Struktur und den Eigenschaften des Projekts und herangezogenen Technologien abhängig ist, können die allgemeinen Grundzüge, nach Popp, dennoch unter fünf gemeinsamen Schritten zusammengefasst

werden (2006). Bei diesen Schritten handelt es sich der Prozessreihenfolge nach, um die Ermittlung der Quellenelemente, die Erstellung des Produkts aus den Quellenelementen, die Prüfung des Produkts, die Auslieferung des Produkts und die Dokumentation der Ergebnisse. Die genannten Schritte werden in detaillierter Form in Kapitel 4 behandelt.

Der Build-Prozess kann grundsätzlich nach zwei verschiedenen Strategien durchgeführt werden (Richardson & Gwaltney 2006):

- als skriptbasierter (manueller) Build-Prozess
- als automatisierter Build-Prozess

2.2 Skriptbasierter Build-Prozess

Bei Teilprojekten von größeren Projekten bzw. Kleinprojekten wird die Erstellung des Produkts mittels Build-Skripten manuell gesteuert. Ulrich schildert die nachteilhaften Aspekte dieser Build-Skripte folgendermaßen (2008, S. 80): „Wenn keine Zeit zur Erstellung eines einheitlichen, übergreifenden Build-Managements eingeplant wurde, entstehen in jedem Teilprojekt meist proprietäre Build-Skripte […] Sie können ihre Aufgabe zwar durchaus erfüllen, aber mit wachsender Projektgröße und komplexeren Anforderungen sind sie immer aufwendiger zu pflegen. Das Deployment der Gesamtanwendung wird zu einem aufwändigen Prozess aus unterschiedlichen Skriptaufrufen. An ein automatisiertes Erstellen der Gesamtanwendung mit generierten Reporten etwa zu Testabdeckung oder Codequalität im Rahmen von Nightly Builds ist nicht zu denken."

Dieser manuell gesteuerte Build-Prozess stellt jedoch eine potenzielle Fehlerquelle dar. So klein der Projektumfang und das Projektteam auch sein mögen, sind bei einer manuellen Steuerung des Build-Prozesses durch Skripte zukünftige Probleme vorprogrammiert. Hier steht vor allem ein großer Zeitverlust im Vordergrund, der durch die Beseitigung von auftretenden Fehlern und die Notwendigkeit der Erklärung einzelner Schritte des Build-Prozesses an alle Teammitglieder und die erforderliche Prozessdokumentation bedingt ist.

2.3 Automatisierter Build-Prozess

Es liegt in der Natur von automatisierten Verfahren, dass sie Genauigkeit, Konsistenz und Wiederholbarkeit voraussetzen. Deshalb sollten Prozesse, die mehr als einmal durchzuführen sind, automatisiert werden (vgl. Gerlich & Gerlich 2005; Clark 2006; Hunt & Thomas 2003). Speziell in agilen Softwareprozessen ist es empfehlenswert, den Build-Prozess so oft wie möglich durchlaufen zu lassen. Da in agilen Softwareprozessen die

kontinuierliche Integration eine tragende Rolle spielt, wird daher den *Nightly Builds* ein wichtiger Stellenwert beigemessen, welche vergleichsweise bei skriptbasierten Build-Prozessen oft nicht unterstützt werden. Die Verbindung und Wechselwirkung zwischen agilen Softwareprozessen und dem Build-Prozess wird unter Kapitel 6.1 genauer dargestellt.

Hüttermann liefert folgende Argumente, um den Nutzen der Automatisierung eines agilen Softwareprozesses darzustellen (2008, S. 101): „Durch vollständige Automation einfacher Arbeitsschritte werden nicht nur Entwickler von Routinetätigkeiten entlastet, sondern die Qualität des Entwicklungsprozesses und der Software deutlich erhöht […] Ziel muss es sein, den gesamten Prozess der Erstellung der Software (den Build) sowie die Software selbst idiotensicher zu machen."

2.3.1 Definition eines Prozesses und eines automatisierten Build-Prozesses

„Ein Prozess ist eine Folge von zwangsläufig aufeinander aufbauender Tätigkeiten (Unterprozessen), die Eingaben (Inputs) in Ergebnisse (Outputs) umwandeln. Ein Prozess setzt sich im Regelfall aus mehreren Unterprozessen zusammen" (Schindler 2004, S. 3).

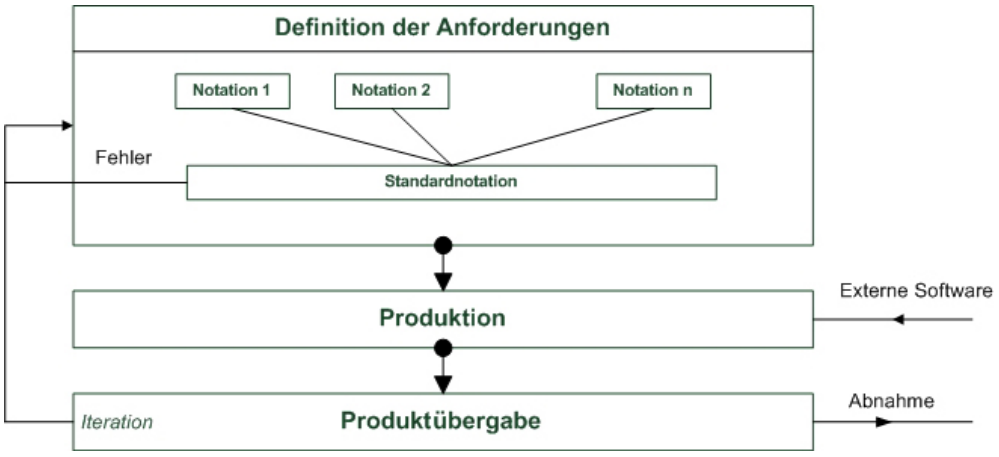


Abbildung 1: Automatisierter Produktionsprozess (aus Gerlich & Gerlich 2005, S. 297)

In Abb. 1 wird der Aufbau eines automatisierten Produktionsprozesses dargestellt. Sie zeigt den hierarchischen Aufbau eines Produktionsprozesses, der grundsätzlich aus drei Teilen besteht: der Definition der Anforderungen, der Produktion und der Produktübergabe. Im Folgenden werden diese drei Teile genauer beschrieben (Gerlich & Gerlich 2005):

- Definition der Anforderungen: Die Anforderungen eines Produktionsprozesses können durch verschiedene Notationen formuliert werden, die zu einer Standardnotation zusammengefasst werden, welche dann den Produktionsprozess steuert.

- Produktion: Hier werden alle Schritte, die zur Herstellung eines Produkts notwendig sind, durchlaufen. Treten keinerlei Fehler auf, welche die Erstellung des Produkts behindern könnten, so wird dessen Erzeugung gestartet.

- Produktübergabe: Der Produktionsprozess findet sein Ende mit der Fertigstellung der Erzeugung des Produkts, welches ab diesem Zeitpunkt zur Abnahme bereitgestellt wird.

Der Automatisierungsprozess gewährleistet insofern die automatische Ermittlung der Quellenelemente aus dem Repository, wobei Änderungen im Repository automatisch angezeigt werden (vgl. Popp 2006; Clark 2006). Anschließend wird durch die Kompilierung dieser Quellenelemente das Produkt erstellt. Hierauf folgt die Durchführung von automatisierten Unit- und Integrationstests, wobei das Resultat dieser Tests und im Zuge des Prozesses entdeckte Fehler in einem Dokument archiviert werden.

Von der anfänglichen Erstellung eines Prototyps bis zur Ausgabe des Endprodukts werden in regelmäßigen Intervallen automatisierte Builds durchgeführt und das aktuelle System in ausführbarer Form an einem bestimmten Ort gespeichert. Das aktuelle Produkt kann somit jederzeit vom Projektteam und den KundInnen eingesehen werden, die somit die Projektentwicklung Schritt für Schritt nachverfolgen können.

Diese Möglichkeit ist insofern von großer Bedeutung, da die Entdeckung und Behebung von Fehlern in der Entwicklungsphase wesentlich kostengünstiger ausfällt als jene, welche erst nachträglich in der Betriebsphase stattfindet (Gerlich & Gerlich 2005). Dieser Faktor wird dadurch bestimmt, dass die Kosten für die Fehlerbeseitigung mit zunehmendem zeitlichem Abstand zwischen Entstehung und Behebung ansteigen. D. h. „Je früher ein Fehler erkannt wird, umso niedriger sind die Kosten für die Beseitigung. Besonders viel spart man, wenn ein Fehler schon in der Spezifikationsphase erkannt wird“ (Gerlich & Gerlich 2005, S. 241).

Die Abb. 2 zeigt die Kosten der Fehlerbeseitigung im speziellen Umfeld des Wasserfall- und V-Models. Der Aufwand für die einzelnen Phasen wird, nach Gilb und Graham, relativ zur Spezifikationsphase angegeben (1993). Der Abb. 2 kann entnommen werden, dass der Aufwand für die Fehlerbeseitigung von Phase zu Phase erheblich ansteigt, sodass Fehler, die erst in der Betriebsphase entdeckt werden, einen 1000-fachen Aufwand bedeuten können.

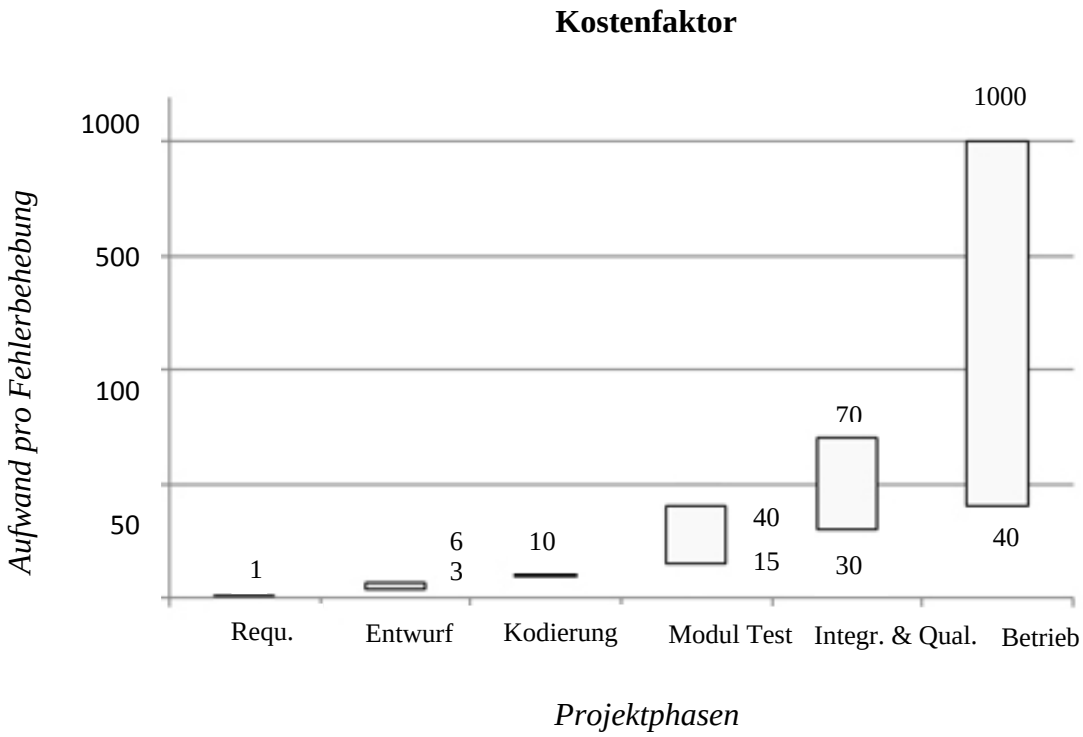


Abbildung 2: Kosten der Fehlerbeseitigung (nach Gerlich & Gerlich 2005, S. 241)

2.3.2 Automatisierungstypen

Wie Clark übersichtlich zusammenfasst, kann man drei verschiedene Automatisierungstypen unterscheiden, darunter befehlsgesteuerte, zeitgesteuerte und ereignisgesteuerte Automatisierung (2006):

- ***Befehlsgesteuerte Automatisierung:*** Jedes Mal, wenn ein Befehl ausgeführt werden soll, ist die Erzeugung eines Build-Prozesses notwendig, wobei der Build erst dann zu Stande kommt, wenn alle definierten Schritte konsistent durchgeführt werden. Nach der Erzeugung des befehlsgesteuerten Builds, gibt es die Möglichkeit, eine zeitgesteuerte oder ereignisgesteuerte Automatisierung ausführen zu lassen, die den menschlichen Beteiligungsfaktor am gesamten Prozessablauf aufhebt.

- ***Zeitgesteuerte Automatisierung:*** Die zeitgesteuerte Automatisierung wird vor allem beim Integrations-Build herangezogen, wobei der Build-Prozess somit in regelmäßigen Zeitabständen bzw. sooft wie gewünscht ausgeführt werden kann, ohne dass ein manueller Eingriff stattfinden muss.

- ***Ereignisgesteuerte Automatisierung:*** Die ereignisgesteuerte Automatisierung hingegen setzt nach bestimmten vordefinierten Ereignissen ein. Beispielsweise kann sie nach jeder Änderung am Quelltext durchgeführt werden, damit Folgefehler der

vorgenommenen Änderungen frühzeitig erkannt und kontinuierlich gemeldet werden. Es besteht auch die Möglichkeit, die ereignisgesteuerte Automatisierung mit der zeitgesteuerten Automatisierung zu kombinieren und gemeinsam ausführen zu lassen.

2.3.3 Eigenschaften des automatisierten Build-Prozesses

In Anbetracht der Funktionsfähigkeit des automatisierten Build-Prozesses müssen außerdem gewisse Eigenschaften erfüllt werden. Clark fasst diese Eigenschaften unter dem Namen **CRISP**-Builds zusammen und ordnet diesen folgende Begriffe zu, woraus diese spezifische Namensgebung resultiert: **C**omplete, **R**epeatable, **I**nformative, **S**chedulable und **P**ortable (2006).

- ***Complete Builds:*** Wie schon vom Begriff der Automation entnommen werden kann, darf in keinem Schritt des Prozessablaufs und am Produkt, welches zum Prozessabschluss entsteht, ein manueller Eingriff vorgenommen werden. Eine manuelle Intervention sollte vermieden werden, um der Fehleranfälligkeit innerhalb des Build-Prozesses vorzubeugen und eine größere Zuverlässigkeit des Prozessablaufs erfüllen zu können.

- ***Wiederholbarkeit und Reproduzierbarkeit***: Produktivität und Qualität können als wichtige Nebeneffekte der Projektautomatisierung angesehen werden. Das Hauptziel der Projektautomatisierung ist es jedoch, die Wiederholbarkeit und Reproduzierbarkeit des Prozesses zu ermöglichen (Popp 2006). Unter Wiederholbarkeit versteht man die Prozedur, in der Schritte, welche zur Erstellung des Produktes notwendig sind, jederzeit wiederholt werden können. Die Reproduzierbarkeit setzt die Wiederholbarkeit eines Prozesses voraus und ermöglicht indes den rückläufigen Eingriff auf einen solchen Prozess. Somit bedeutet dies, unter denselben Bedingungen, dasselbe Produkt herstellen zu können. Dabei ist die Versionskontrolle das grundlegende Element, welches die Reproduzierbarkeit überhaupt ermöglicht. Speziell die Versionierung von Inputartefakten und Build-Skripten bildet einen wichtigen Faktor, um einen älteren Stand des Produktes erneut erstellen zu können.

- ***Informative Builds:*** Die Aufgabe informativer Builds ist die Ausgabe einer Dokumentation und Bereitstellung von Informationen zu den gesamten Schritten des Build-Prozesses. Dabei wird die Funktionalität des Prozesses sichergestellt, indem die fehlerfreie Kompilierung des Codes, der erfolgreiche Testdurchlauf und die Korrektheit der übrigen Build-Artefakte bestätigt werden. Wichtig ist daneben die vollständige und detaillierte Auslieferung der Informationen, um einen Zeitverlust bei einer eventuellen Fehlersuche zu vermindern und ein ausführliches Protokoll zum entstandenen Produkt durch die automatisierte Dokumentation zu gewinnen.

- ***Zeitgesteuerte Builds:*** Innerhalb eines Softwareentwicklungsprozesses werden, je nach unterschiedlichen Anforderungen, in regelmäßigen Zeitabständen Build-Prozesse durchlaufen. Diese Wiederholungen in regelmäßigen Zeitintervallen werden seitens McCarthy mit *Herzschlägen* verglichen. Somit würde es bei einem Herzstillstand zum Scheitern des Projekts und bei unregelmäßigen Herzschlägen zu einem unvollständigen und fehlerhaften Projektablauf kommen (1995). Unter der Bedingung, dass alle für einen Build notwendigen Dateien in der Versionsverwaltung verfügbar sind, kann ein zeitgesteuertes Build zu einer bestimmten Uhrzeit, in regelmäßigen Abständen, zu einem Ereignis oder Eines nach dem Anderen stattfinden. Zur Festlegung dieser Zeitintervalle gibt es verschiedene Meinungen und Vorschläge. Dabei sprechen sich Cusumano und Selby für die tägliche Wiederholung des Build-Prozesses (sog. *Nightly Builds*) aus und betonen die Wichtigkeit der Synchronisierung des Pulses eines Projekts (1995). Fowler hingegen empfiehlt die Aktivierung des Build-Prozesses im Zeitabstand von allen 30 Minuten (2006). Seinen eigenen Erfahrungen und Untersuchungen zufolge vertritt Popp die Ansicht, dass die Intervalle den unterschiedlichen Projektphasen angepasst werden sollten. Somit hält er in frühen Phasen die Ausführung des Build-Prozesses ein Mal pro Woche, und in der Wartungsphase mindestens ein Mal täglich für ausreichend (2006). Ob nun ein Mal in der Woche oder mehrmals täglich, wichtig ist es, einen zeitgesteuerten Build-Prozess durchlaufen zu lassen.

- ***Portable Builds:*** Die sicherste Methode, um die Eigenschaften Complete, Repeatable und Scheduleable eines Builds zu erfüllen, ist die Einrichtung eines eigenen Build-Servers. Durch die Nutzung eines solchen Servers können die, in Anwendungen oder auf der Dateiebene, bestehenden Sicherheitsrisiken minimiert werden. Dabei ist es nicht ausschlaggebend, auf welchem Rechner oder Betriebssystem dieser Server ausgeführt wird. Daher sollte es möglich sein, eine Anwendung, welche auf einem bestimmten Betriebssystem gebaut wurde, auf irgendeinem anderen Rechner mit demselben Betriebssystem zu bauen. „Ebenso sollte das Ausführen eines Builds nicht von einer bestimmten IDE, der IP-Adresse eines Rechners oder dem Verzeichnis abhängig sein, von wo der Build gestartet wird“ (Clark 2006, S. 14).

2.3.4 Weitere Eigenschaften des automatisierten Build-Prozesses

Abgesehen von den Eigenschaften der **CRISP**-Builds, gibt es eine Reihe von weiteren charakteristischen Merkmalen, die auf der Ebene des automatisierten Build-Prozesses erfüllt werden sollten. Dabei handelt es sich um die Skalierbarkeit, Integrierbarkeit, Wiederverwendbarkeit und Erreichbarkeit.

- ***Skalierbarkeit:*** Von Skalierbarkeit ist die Rede, wenn „der Build-Vorgang parallelisiert und die einzelnen Aktionen auf verschiedene Prozessor- oder Rechnerinstanzen verteilt“ (Hoffmann 2008, S. 464) werden. Es gibt zwei Arten von Skalierbarkeit (Gerlich & Gerlich 2005):

  - Art und Größe einer Anwendung sind skalierbar
  - Die Produktionszeit ist skalierbar

  Skalierbarkeit der Anwendung: Ist eine Anwendung skalierbar, so können beliebige Mengen an Daten bearbeitet und auf verschiedene Rechner verteilt werden. Dabei ist zu beachten, dass im Falle einer Änderung der jeweiligen Datenmengen kein manueller Eingriff unternommen, sondern der Umbau des Produktionsprozess automatisch vollzogen wird. Damit wird sichergestellt, dass die Verteilung der Datenmengen reibungslos ablaufen kann. Wichtig ist auch, dass „nicht nur Mengen, sondern auch Funktionalität sowie Performance und Ressourcen“ (Gerlich & Gerlich 2005, S. 334) skaliert werden können.

  Skalierbarkeit der Produktionszeit: Die Skalierbarkeit der Produktionszeit ist ein Phänomen, welches nur bei vollautomatisierten Produktionsprozessen funktionsfähig ist und bei manuellen Eingriffen seine Gültigkeit verliert. So ist in automatisierten Produktionsprozessen die Produktionszeit direkt von der Leistungsfähigkeit des verwendeten Rechners abhängig und wird in folgender Formel zusammengefasst (Gerlich & Gerlich 2005, S. 334):

  $$T_{prod} = const \ / \ CPU_{Power}$$

  Je höher die Leistungsfähigkeit (CPU_{Power}) des Rechners ist, umso geringer fällt die gesamte Produktionszeit (T_{prod}) aus.

- ***Integrierbarkeit:*** Um die Funktionsfähigkeit eines Build-Prozesses zu ermöglichen, ist die Integrierung von verschieden Tools erforderlich, die auch untereinander in Verbindung stehen und kommunizierend arbeiten können. Dabei ist vor allem die Integrierbarkeit der Version Control Tools und Integrated Development Environments (IDE) wichtig und erforderlich.

- ***Wiederverwendbarkeit:*** Der Build-Prozess sollte prinzipiell so konzipiert sein, dass dieser bei mehreren bzw. weiteren Projekten erneut zum Einsatz kommen kann. Wenn sie richtig eingesetzt wird, kann die Wiederverwendbarkeit einen wesentlichen Beitrag zur Verringerung von Zeit und Aufwand leisten. In diesem Sinne ist es von Vorteil, wenn Build-Skripte, die in einem Projekt mit zahlreichen Komponenten oder aber

mehreren Projekten zum Einsatz kommen, wiederverwendbar gemacht werden (Lee 2008).

- ***Erreichbarkeit:*** Da innerhalb eines Entwicklungsteams mehrere Mitglieder (EntwicklerInnen, TesterInnen, Build-ManagerInnen) am Build-Prozess arbeiten und mitwirken, ist es notwendig, diesen stets zeit- und ortsunabhängigen Zugang zum Build-Prozess zu verschaffen. So muss es möglich sein, dass Teammitglieder den Build-Prozess über verschiedene Zugangspunkte erreichen können.

2.4 Architektur des Build-Prozesses

Die Architektur des Build-Prozesses setzt sich aus einem dreiteiligen Gerüst zusammen, das aus der Build Infrastruktur, dem Build Scripting und der Build Kontrolle besteht.

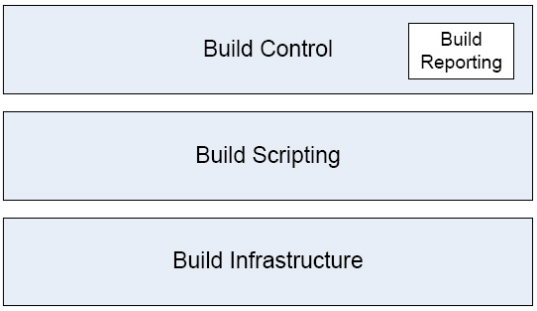


Abbildung 3: Ressourcen des Build-Prozesses (aus Lee 2008, S. 108)

In Abb. 3 ist eine Kombination typischer Ressourcen dargestellt, welche die Implementierung des Build-Prozesses ermöglichen. Im Allgemeinen handelt es sich bei diesen Ressourcen oftmals um bestimmte Hardware und Tools, sowie einige Build-Skripte, wobei aber auch die Implementierung sonstiger Ressourcen in den Build-Prozess möglich ist (Lee 2008).

2.4.1 Build Infrastruktur

Lee definiert die Build Infrastruktur als die kollektive Infrastruktur, die für die Umsetzung und Unterstützung eines Build-Prozesses notwendig ist. Dabei beinhaltet die Infrastruktur eine Reihe von Build- und Versionskontroll-Tools und auch die erforderliche Hardwareunterstützung (2008). Die Abb. 4 zeigt die Standardinfrastruktur eines Builds und die Beziehungen zwischen den verschiedenen Komponenten.

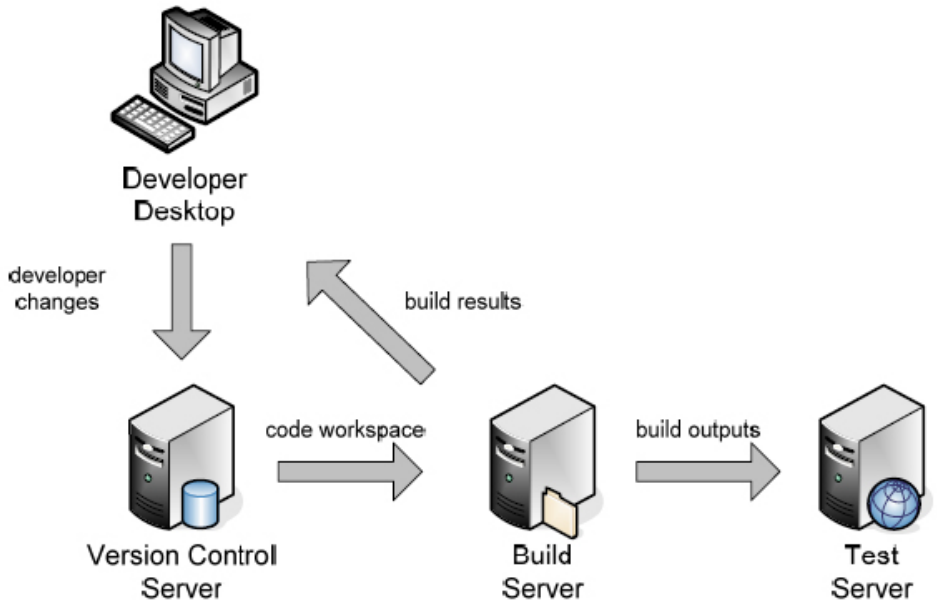


Abbildung 4: Standard Build Server Infrastruktur (aus Lee 2008, S. 40)

Bei dem Developer Desktop handelt es sich um eine eigene Workstation für die EntwicklerInnen. Mittels geeigneter IDE auf dieser Workstation, stehen den EntwicklerInnen eigene interne Build-Funktionen zur Verfügung. Im Version Control Server werden die Repositories gespeichert und alle Änderungen festgehalten, welche dann zu den Integrations- oder Release-Builds hinzugefügt werden. Auf dem Build Server wird die Durchführung von Integrations- oder Release-Builds vollzogen. Grundsätzlich besteht die Möglichkeit, Tests, die ein fester Bestandteil des Build-Prozesses sind, auf einem eigenen Server, dem sog. Test Server, ausführen zu lassen.

Die Build Infrastruktur ist maßgeblich von projektspezifischen Zeit- und Kostenfaktoren abhängig. Deshalb sollte besonders die gesamte Laufzeit des Build-Prozesses berücksichtigt werden, sodass eine zeitliche Anpassung an den Build-Rhythmus stattfindet. Bei komplexeren und aufwändigeren Projekten kann ein solcher Build-Prozess mit enormen Laufzeiten verbunden sein. Daher ist es wichtig, die erforderliche Zeit in akzeptablen Grenzen zu halten. Der Zeitaufwand kann beispielsweise durch den Einsatz von großen Rechnerfarmen (sog. *cluster computing*) relativiert werden. Um die Kosten bei der Festlegung einer Build-Infrastruktur möglichst gering zu halten, können die sog. Rechnerfarmen bei mehreren laufenden Projekten verwendet werden (vgl. Lee 2008; Hoffmann 2008). Eine genauere Behandlung der Versionskontrolle wird in Kapitel 4, und der Build-Tools in Kapitel 3 erfolgen.

2.4.2 Build Scripting

Zur Umsetzung des Build-Prozesses werden verschiedene Skripte benötigt, „ […] in denen die durchzuführenden Aufgaben in einer speziellen Skriptsprache formuliert werden“ (Popp 2006 S. 47). Grundsätzlich kann zwischen zwei Skriptarten unterschieden werden: den Build-Skripten und Shell-Skripten (auch *Command-*, *Controlling-* oder *Subskript* genannt).

- Build-Skripte: Die Grundbestandteile von Build-Tools sind Build-Skripte, bei denen deklarative oder imperative Skriptsprachen verwendet werden. In den Build-Skripten wird die allgemeine Beschreibung eines Build-Prozesses, mit den einzelnen auszuführenden Prozessschritten, strukturiert. Beispielsweise kommt beim Build-Tool Ant die *imperative Skriptsprache XML* zum Einsatz. Dadurch ergeben sich zahlreiche Vorteile, wie die Plattformunabhängigkeit, Erweiterbarkeit und Wiederverwendbarkeit. Im Gegensatz dazu, verwendet Maven eine *deklarative Skriptsprache*, die *Project Object Model (POM)* genannt wird. Hier wird für alle Projekte ein globales Modell herangezogen, das aber auch die Option bietet, jedes individuelle Projekt durch erforderliche Parameter zu ergänzen (Popp 2006).

- Shell-Skripte: Diese Skripte werden zur Durchführung von logischer oder datenbasierter Manipulation verwendet (Lee 2008), um damit den Build-Prozess zu unterstützen. Ein großer Nachteil der Shell-Skripte ergibt sich durch die Plattformabhängigkeit, sodass unterschiedliche Plattformen innerhalb eines Projekts die mehrfache Erstellung und parallele Betreuung von Skripten erforderlich machen (Popp 2006).

2.4.3 Build Kontrolle

Nach Blanton kann die Build Kontrolle folgendermaßen definiert werden: Build Kontrolle bedeutet, dass jede Datei und jedes Parameter, die in jedem einzelnen Schritt eines Build-Prozesses zum Einsatz kommen, kontrolliert werden, bevor das Build ausgeführt wird und folglich auch eine zeit- und ortsunabhängige Reproduktion des Builds durchgeführt werden kann (2006). Bei der Build-Kontrolle werden die in der Build Scripting Phase implementierten Build-Skripte in einer kontrollierten bzw. automatisierten Art und Weise ausgeführt.

2.5 Rollen im Build-Management

In Anlehnung an die Definition von Lee ist es möglich, die „KundInnen“ (2008) eines Build-Prozesses zu benennen, bei denen es sich zu verschiedenen Zeiten um verschiedene Personen handeln kann. So kann es sich bei diesen KundInnen um EntwicklerInnen, TesterInnen, ProjektmanagerInnen, QualitätssichererInnen und letztlich auch um EndbenutzerInnen handeln. In erster Linie sind EntwicklerInnen als SchlüsselkundInnen im Prozess anzusehen. Sie haben die Aufgabe den Quellcode zu erstellen, die Inputartefakte des Build-Prozesses zu betreuen und nach den Tests die notwendigen Änderungen durchzuführen. Die TesterInnen sind verantwortlich für die Bereitstellung von Unit-Tests bzw. Testdaten und müssen nach Abschluss der Tests, anhand der ausgelieferten Prozessdokumentation, Berichte verfassen. Die zuständigen MitarbeiterInnen für die Qualitätssicherung sind indes für die Überwachung des Produkts verantwortlich. Ihre Aufgabe ist es, die Qualität des Produkts zu messen, indem sie die gesamten Messdaten, die nach jedem Build-Prozess ausgegeben werden, überprüfen.

Die zentrale Rolle im ganzen Build-Prozess hat der/die Build-ManagerIn inne und trägt die alleinige Verantwortung für diesen. Der/die Build-ManagerIn kann optional auch als Release-ManagerIn bezeichnet werden. So ist er/sie für den gesamten Aufbau des Build-Prozesses zuständig und muss direkt selbst für die Ausführung des Build-Prozesses sorgen oder die Automatisierung des Prozesses veranlassen bzw. verbessern. Die Sammlung, Bestimmung und Auslieferung von jeglichen Tools, die für die Erstellung einer bestimmten Software notwendig sind, gehört zu den Aufgabenbereichen des Build-Managers/der Build-Managerin, um die verlässliche Reproduktion und Instandhaltung von Software, auch Jahre nach dem erstmaligen Release, in den Dienst der KundInnen zu stellen (Wikipedia, Release-Engineering).

Nach Lee ist der/die Build-ManagerIn für folgende Arbeitsbereiche verantwortlich (2008, S. 163):

- „defining the project’s build process based on the input of the Developers
- implementing, automating and securing the build process
- liaising with the Configuration Manager to ensure that the build process correctly interfaces to the SCM [1] environment.
- Automating or initiating the deployment of the application to Development and Acceptance Test environments.“

Abhängig von der Größe des jeweiligen Projekts, muss nicht ein/e Einzelverantwortliche/r als Build-ManagerIn auftreten, sondern es können auch mehrere Verantwortliche in einem sog. *Build- oder Release Engineering Team* beauftragt werden.

[1] Software Configuration Management.

2.6 Build-Rhythmus

Im Allgemeinen ist der Build-Rhythmus abhängig von der Art des jeweiligen Builds und folglich ist es wichtig zu unterscheiden, wofür ein Build zum Einsatz kommt und in welchen Zeitintervallen und/oder zu welchen Ereignissen dieser ausgeführt werden soll. So kann es sich bei diesem Ereignis um die Veröffentlichung eines neuen Release für die Endkunden handeln, welches schon zuvor im Releaseplan zeitlich festgelegt wurde. Ein weiterer ereignisgesteuerter Build ist der Commit-Build, der nach jedem vollzogenen Check-In den Start des Build-Prozesses veranlasst. Die kontinuierliche Integration hingegen, würde die Ausführung des Builds in regelmäßigen Zeitintervallen, beispielsweise wöchentlich, täglich oder alle 20 Minuten, verlangen. Scumniotales zieht einen Vergleich zwischen dem komplexen Rhythmus eines Projekts und einer musikalischen Komposition und fasst seine Überlegungen in Abb. 5 zusammen (2006):

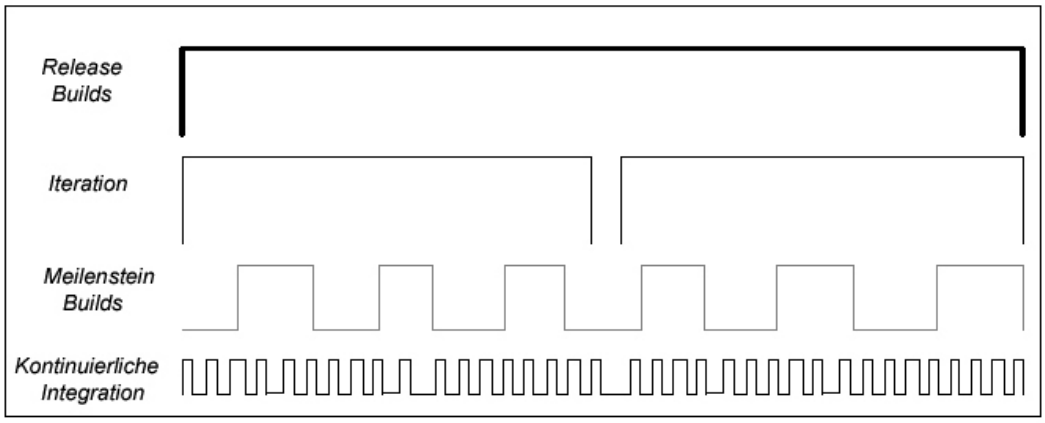


Abbildung 5: Build-Rhythmus (nach Scumniotales 2006)

Auch wenn der Quellcode aus mehreren tausend Zeilen besteht und noch so umfangreich ist, zeigt sich dennoch ein enormes Zeitsparpotential bei der Durchführung von Builds in kürzeren Zeitintervallen im Vergleich zu wöchentlichen Builds. Dadurch können EntwicklerInnen schneller auf entstandene Konflikte reagieren und diese leichter lösen. So lautet die Devise der kontinuierlichen Integration, dass es umso besser für das Projekt ist, je öfter der Build-Prozess ausgeführt wird. Bei häufiger Ausführung der Integration sollte sich ein schmerz- und konfliktloser Prozess mit einer Menge Zeitersparnis ergeben (Fowler 2006).

Da für die Qualitätssicherung die Durchführung von Meilenstein-Builds notwendig ist und auch die Produkt-ManagerInnen über Iterationen High-Level-Feedbacks zur Integration und Usability des Projekts bekommen, ist hier nochmals zu betonen, dass in dieser Hinsicht die Ausführung von Builds in regelmäßigen Abständen wichtig für den internen

Informationsfluss und Arbeitsfortschritt ist. Lee definiert einige für den Build-Rhythmus allgemein gültige Fragen, so z.B. (2008):

- Gibt es irgendwelche Abhängigkeiten zwischen internen und externen Komponenten?
- Wie viele EntwicklerInnen arbeiten an dem Projekt? Wie oft wird von ihnen die Auslieferung von Änderungen für die Integration erwartet?
- Welche Eigenschaften erwarten die TesterInnen vom Integrations-Build für ihre eigenen Tests?
- Wie lange dauert es einen Integrations- oder Release-Build durchzuführen?
- Wie oft wird die Durchführung von Integrations-Builds erwartet?
- Wie oft wird die Durchführung von Release-Builds erwartet?

Die Beantwortung dieser Fragen sollte im gemeinsamen Gespräch der Build-ManagerInnen mit den verschiedenen Teammitgliedern stattfinden, damit ein geeigneter Build-Rhythmus des Projekts und in weiterer Folge auch die gesamte Build-Struktur festgelegt werden kann (Lee 2008).

Ein geregelter Build-Rhythmus ist für ProjektmitarbeiterInnen hilfreich bei der organisatorischen Bewältigung des Projektablaufs. Da nun Vorteile und Notwendigkeit eines regelmäßigen Build-Rhythmus ausführlich dargestellt worden sind, ist an dieser Stelle nochmals anzumerken, dass es innerhalb eines Projekts/Projektteams wichtig ist, nicht nur das individuelle Tempo eines Teammitglieds zu berücksichtigen, sondern ein übergeordnetes Tempo, d.h. einen allgemein gültigen Projektrhythmus zu definieren.

Da die manuelle Steuerung von Builds, die in regelmäßigen Zeitintervallen wiederholt werden, sich als aufwendige und zeitraubende Aufgabe herausstellt, ist in diesem Zusammenhang der Einsatz eines Automatisierungsprozesses von großem Nutzen.

2.7 Metriken

DeMarco liefert uns zum Thema Metriken eines der bedeutendsten Zitate, wenn er sagt (1982, S. 3): „You cannot control what you cannot measure.“

Nach IEEE Standard 1061 wird eine Metrik als Funktion betrachtet, „die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.“
Wichtig ist, dass Metriken zielorientiert sind und nach einem bestimmten Verfahren, dem „*Goal-Question-Metric*“ *(GQM)* - Verfahren, identifiziert werden (Fleischer 2007). Das GQM-Verfahren gliedert sich in drei Schritte (Basili et al. 1994):

- Goal: Das Ziel wird festgelegt.
- Question: Geeignete Fragen werden bestimmt, um die Erreichung des Ziels überprüfen zu können.
- Metric: Metriken, welche die Anforderungen der gestellten Fragen erfüllen, werden ausgewählt.

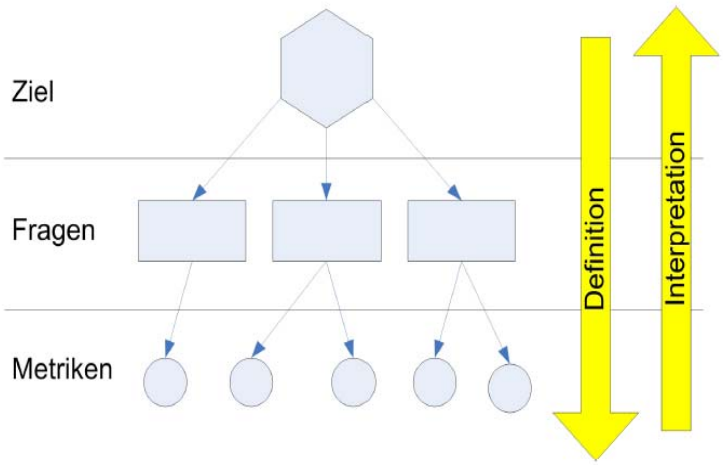


Abbildung 6: Goal-Question-Metric-Vefahren (aus Böhmer 2006)

Die Abb. 6 zeigt die Struktur des GQM-Verfahrens und die Abfolge der einzelnen Schritte. An oberster Stelle steht das festgelegte Ziel, das es zu erreichen gilt, wobei hierfür geeignete Fragen und entsprechende Metriken definiert werden müssen. Den Weg, der umgekehrt von den Metriken zum Ziel beschritten wird, bezeichnet man als Interpretation. Wenn es um den Build-Prozess geht, dann stehen zwei Metriken im Mittelpunkt des Geschehens: die *Produktmetriken* und die *Prozessmetriken*.

2.7.1 Produktmetriken

Produktmetriken dienen zur Messung der Eigenschaften und Qualitätsmerkmale des hergestellten Produkts. Dabei ist zu beachten, dass es verschiedene Varianten von Metriken gibt. Eines der bekanntesten unter diesen ist die LOC-Metrik (*Lines-Of-Code*), welche die gesamte Zeilenanzahl des Quellcodes angibt. Es kann eine weitere Untergliederung von LOC, in die NCSS-Metrik (*Non-Commenting-Source-Statement*) und die Kommentar-Metrik unternommen werden. Diese Metriken liefern uns jedoch sehr grobe Werte, die zur umfangreichen Messung des Produkts nicht ausreichend sind (vgl. Popp 2006; Fleischer 2007). An dieser Stelle kann mittels der CCN-Metrik (*Cyclomatic*

Complexity Number) die strukturelle Komplexität einer abgeschlossenen Komponente gemessen werden, „die für den Menschen nicht mehr begreifbar ist“ (Wikipedia, McCabe-Metrik). Die Messung beruht hier auf der Grundlage des Kontrollflussgraphen. „In der Praxis hat sich 20 als zuverlässiger Grenzwert für die CCN pro Methode bewährt. Ein hoher CCN deutet auf eine komplexe Methode hin, die schwer zu ändern ist und einen hohen Testaufwand erzeugt“ (Fleischer 2007, S. 61).

Eine weitere wichtige Variante ist die ACD-Metrik (*Average Component Dependency*), welche zur Messung der Abhängigkeiten zwischen Komponenten herangezogen wird, um Hinweise bezüglich der Qualität der Architektur zu erhalten. Unter den Komponenten kann es entweder zu direkten oder indirekten Abhängigkeiten kommen (Fleischer 2007).

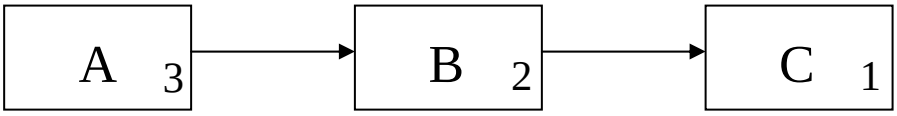


Abbildung 7: Average Component Dependency

In Abb. 7 sind die direkten und indirekten Abhängigkeiten der Komponenten und ihre Abhängigkeitswerte ersichtlich. A ist direkt abhängig von B und indirekt von C. B ist direkt abhängig von C. Da C nur von sich selbst abhängig ist erhält es den Wert 1. B ist wiederum von sich selbst und direkt von C abhängig und erhält den Wert 2. Durch seine Abhängigkeit von sich selbst, der direkten Abhängigkeit von B und indirekten von C, erhält A den Wert 3. Die gesamten Abhängigkeitswerte im Durchschnitt ergeben die ACD-Metrik, woraus sich der Kopplungsgrad ermitteln lässt.

Zur Gruppe der meistverwendeten Metriken für objektorientierte Software gehören auch:

- CBO (*Coupling between Objects*): Anzahl der Klassen
- DIT (*Depth of Inheritance Tree*): Anzahl der Oberklassen einer Klasse
- NOM (*Number of Methods*): Anzahl der Methoden in einer Klasse
- *Efferent Coupling* (ausgehende Abhängigkeiten), *Afferent Coupling* (eingehende Abhängigkeiten)

Alle oben genannten Metriken werden entweder unter dem Begriff der sog. „technischen“ (Fleischer 2007) oder „automatisierten“ (Popp 2006) Metriken zusammengefasst. Mittels Integrierung von Werkzeugen in den Build-Prozess kann eine automatisierte Bestimmung von Metriken gewährleistet werden. Als Gegenpart zu den automatisierten Metriken, gibt es auch solche, die manuell ermittelt werden. Nicht-technische Metriken, die manuell ermittelt werden, sind durch Use-Cases formuliert und haben die Aufgabe, die aktuelle

Größe des Produktes und seinen Fertigstellungsgrad zu bestimmen. Nach Popp werden folgende manuelle Metriken als sinnvoll gewertet (2006):

- Neu entdeckte Fehler: Hier wird die Anzahl der neuen Fehler, die in einem bestimmten Intervall entdeckt wurden, angegeben.

- Behobene Fehler: Hier wird die Anzahl der behobenen Fehler, die in einem bestimmten Zeitintervall entdeckt wurden, angegeben.

- Aufwand pro Fehlerbehebung: Hier wird im Zuge der Release-Vorbereitung der ungefähre Aufwand für die erforderliche Fehlerbehebung ermittelt.

- Fehlerdichte: Hier wird die Häufigkeit der auftretenden Fehler pro bestimmten Lines-Of-Code ermittelt. Die Ermittlung der Fehlerdichte ist aus dem Grund wichtig, da sie eine wesentliche Hilfestellung bei der Vorausschätzung der erwarteten Fehleranzahl in zukünftigen Releases darstellt. So sieht Fleischer die Fehlerdichte als einen guten Indikator, der einen wichtigen Anhaltspunkt liefert, ob eine Erhöhung oder Reduzierung der Intensität des Tests notwendig ist (2007).

Produktmetriken ermöglichen die Ermittlung der Qualität des Produkts mittels Messungen, sowie die dauerhafte Qualitätssicherung. Zudem werden Informationen zum aktuellen Status, der Wartbarkeit und Erweiterbarkeit des Produktes geliefert (Fleischer 2007).

2.7.2 Prozessmetriken

Die Prozessmetriken sind für die Messung der Eigenschaften, Aktivitäten und Instrumente des Softwareentwicklungsprozesses vorgesehen. Sie geben jene Messdaten an, die auf den Build-Prozess selbst bezogen sind, wie z.B. den Ressourcenverbrauch, die Häufigkeit bestimmter Ereignisse, sowie Aufwand, Kosten und Dauer der Projektaktivitäten (vgl. Amelingmeyer & Specht 2005; Holzmann 2003). Die Prozessmetriken gewährleisten die Abdeckung von vier Aspekten: Qualität, Performance, Nutzen und Compliance (Buchsein et al. 2007). Die Analyse dieser Metriken ermöglicht die Beurteilung des Prozesses und ist folglich für die Optimierung und fortlaufende Verbesserung des Build-Prozesses äußerst hilfreich. Daneben ist auch das Monitoring ein bedeutender Dienst dieser Metriken.

Prozessmetriken können zudem zur Nachverfolgung des Prozessfortschritts, der Projektdynamik und des Projektklimas herangezogen werden. Der Projektfortschritt kann insofern anhand der Anzahl der definierten, entworfenen, codierten, geprüften und getesteten Module oder der Anzahl der fertiggestellten Code-Zeilen und Dokumentationsseiten gemessen werden. Auch die Verfolgung von Fehlermeldungen, Testaktivitäten und Reviews können wichtige Hinweise zum Projektfortschritt liefern. Die

Projektdynamik kann indes durch die Anzahl beantragter, genehmigter und realisierter Änderungen bestimmt werden. Aussagen zum Projektklima erhält man, indem mittels regelmäßiger Team- und Stakeholderbefragungen ein Klimaindex ermittelt wird (Amelingmeyer & Specht 2005). Als wichtige Beispiele für Prozessmetriken wären die *Maturity-Metriken, Management-Metriken* und *Life-Cycle-Metriken* zu nennen (Dumke 1997).

2.8 Build-Prozessplan

Ausgehend von den Anforderungen des Produkts kann der Build-Prozessplan als ein schriftliches Dokument beschrieben werden, bei dem die notwendigen Entscheidungen und Kompromisse, sowie spezifische Definitionen des Build-Prozesses angegeben werden. In diesem Plan werden die Prioritäten des Build-Prozesses festgehalten und alle Fragestellungen im Bezug auf diesen beantwortet. Nach Nabrzyski sollten diese Priöritäten gemeinsam mit den KundInnen bestimmt werden (2002). Lee beschreibt vier Bereiche, denen beim Build-Prozessplan besondere Beachtung geschenkt werden sollte (2008):

- Organisation des Build-Management Teams und Bestimmung der Verantwortlichkeiten
- Build-Tools, Umgebung und Infrastruktur
- Build-Aktivitäten, inklusive Identifikations- und Kontrollmechanismen
- Build-Accounting, inklusive Metriken, Reports und Audits

Bei der Festlegung der organisatorischen Struktur des Build-Managements werden die Zuständigkeiten und Verantwortlichkeiten der einzelnen Teammitglieder und ihre Beziehungen untereinander bestimmt. Außerdem ist zu entscheiden, welche Build-Tools zu verwenden sind und welche Aufgabe diese erfüllen werden, wobei sie in eine funktionierende Infrastruktur eingebettet werden sollten. Zu den Identifikationsmechanismen gehört die Festlegung der Build-Frequenz. Daneben muss die Vergabe einer Release-Nummer und Build-Nummer zu den einzelnen Builds stattfinden, um jedes einzelne Build identifizieren zu können. Insofern muss auch eine Hierarchie der Build-Verzeichnisstruktur erstellt werden. Kontrollmechanismen dienen zur Gewährleistung der Sicherheit und der Autorisierung eines Builds. So werden hier die Zugriffsrechte der ProjektmitarbeiterInnen bestimmt. Im Build-Accounting wird angegeben, wie die Rollenvergabe bei der Erstellung und Durchführung von Reports und Audits aussieht, auf welche Aktivitäten diese bezogen sind und welcher Umfang dafür vorgesehen ist. Zuletzt ist noch anzumerken, dass die Gestaltung des Build-Prozessplans maßgeblich von den Bedingungen und Anforderungen des jeweiligen Projekts abhängt (Lee 2008).

3. Build-Tools

Build-Tools stellen einen besonderen Teil des gesamten Softwareentwicklungssystems dar. Ihre Aufgabe besteht darin, die Beschreibung eines Builds zu lesen und anschließend weitere Tools zum Laufen zu bringen, die das Ergebnis des Builds produzieren. Im Allgemeinen kommen im gesamten Build-Prozess mehrere Tools zur Anwendung, darunter auch das Build-Tool selbst. Während des Build-Prozesses können zudem die Ergebnisse des Builds verwaltet und Beschreibungen des Builds produziert oder angepasst werden, bevor diese überhaupt vom Build-Tool genutzt werden (Neagu 2005).

3.1 Grundlegende Technologie

Bevor hier genauer auf die Eigenheiten und Funktionsweisen einzelner ausgewählter Build-Tools eingegangen wird, ist es wichtig, im Vorhinein einen kurzen Überblick über jene Kerntechnologien zu geben, die von diesen Build-Tools herangezogen werden. Dieses Kapitel wird sich den Technologien RegExp, XML und XSLT widmen.

3.1.1 Reguläre Ausdrücke

Reguläre Ausdrücke (auch RegExp, Regex oder RES) „sind Muster, die auf eine Zeichenkette angewendet werden können, um beispielsweise feststellen zu können, ob die Zeichenkette bestimmten Vorgaben entspricht, oder um bestimmte Teile des Textes, die zum Teil variabel sein können, zu ersetzen“ (Polzer 2002).

Reguläre Ausdrücke sind leicht verfügbare, flexible und leistungsstarke Mittel, die zur Verarbeitung von Texten und Daten herangezogen werden (msdn.microsoft.com, Reguläre Ausdrücke). Sie können eine große Hilfestellung bei der Lösung von komplexen und subtilen Textverarbeitungsprogrammen sein, und somit viel Arbeit und Komplikationen ersparen. Lee spricht von bestimmten Situationen während des Build-Prozesses, in denen die Implementierung solcher regulärer Ausdrücke notwendig sein kann (2008):

- Bei der Beschreibung des Speicherorts und des Namens der Quelldateien in den Build-Scripting Tools – anstatt jede einzelne Datei manuell spezifizieren zu müssen.
- Bei der Ersetzung eines Platzhalter-Strings in einer Quelldatei mit einer spezifischen Build-Nummer und der Build-Zeit.
- Bei der Ausführung der Befehle *change* und *history* in der Versionskontrolle und der Formatierung der Outputs in lesbare Release-Reports.

3.1.2 XML

XML (*eXtensible Markup Language*) ist eine textbasierte Markup-Sprache, welche die Verwaltung, Darstellung und Organisierung von Daten, auf konsistente und Tool-unabhängige Art, ermöglicht (Lee 2008). XML ist kein neues Konzept, sondern vielmehr die „vereinfachte Form“ (Partl 2000) von *Standard Generalized Markup Language* (SGML).

Ein XML-Dokument besteht aus physikalischen und logischen Strukturen, die folgendermaßen beschrieben werden können: Die physikalische Struktur eines XML-Dokuments setzt sich aus Speichereinheiten zusammen, die im Allgemeinen eine Kette von bestimmten Zeichen sind. Diese Zeichenkette wird durch XML in verwertbare Stücke zerlegt, die unter dem Begriff Entitäten zusammengefasst werden. Dabei kann sogar ein einzelnes Zeichen als Entität gelten, wobei das gesamte Dokument die Bezeichnung „Dokument-Entität“ (Vonhoegen 2004, S. 44) trägt. Indes beruht der logische Aufbau eines XML-Dokuments auf einer Baumstruktur mit hierarchischen Informationseinheiten. Diese Einheiten bestehen hauptsächlich aus Elementen und ihren zugehörigen Attributen. So ist ein Element ein XML-Objekt, welches über einen Start- und einen End-Tag verfügt. Ein XML-Attribut hingegen ist ein Schlüsselwort-Werte-Paar, mit dem verschiedene Eigenschaften bzw. Besonderheiten eines Elements festgehalten oder auch geändert werden können (vgl. Vonhoegen 2004; Partl 2000; Wikipedia, Extensibler Markup Language).

Lee betont zudem, dass XML eine äußerst nützliche Rolle im Build-Prozess spielen kann, wenn alle Outputs im XML-Format ausgegeben werden. Somit können die Output-Dateien standardisiert und der Austausch von Informationen zwischen verschiedenen Tools und NutzerInnen erleichtert werden (2008).

Der Grund, weshalb die Programmiersprache XML an dieser Stelle in die Betrachtung miteinbezogen wird, ergibt sich daraus, dass diese sowohl als Input- als auch als Output-Format genutzt werden kann. So begegnet uns XML nicht nur bei dem Beispiel Apache Ant, welches im Kapitel 3.3 genauer behandelt wird, sondern auch bei weiteren zahlreichen Build-Scripting-Tools als Input-Format.

3.1.3 XSLT

XSLT (*X Stylesheet Language Transformations*) ist eine empfohlene XML Stylesheet-Sprache, deren primäre Aufgabe darin liegt, den NutzerInnen die Transformation von XML zu XSL-FO zu ermöglichen und so die Beschreibung zur Darstellung von XML-Dokumenten zu liefern. Mit XSLT können beliebige Transformationen von einem XML-Dokument in ein Anderes durchgeführt werden. Es zählt zu den am häufigsten

verwendeten Basis-Tools bei XML-Transformationen und wird in einer XML Abfragesprache dargestellt (Kepser 2004).

Um eine beliebige XML-Transformation durchzuführen, ist es notwendig eine XSLT Stylesheet-Datei zu erstellen. Diese Stylesheet-Datei „[…] besteht aus einer Reihe von einzelnen Transformationsregeln, die Templates heißen. Ein Template besitzt ein auf XPath basierendes Pattern, das beschreibt, für welche Knoten es gilt, und einen Inhalt, der bestimmt, wie das Template seinen Teil des Zielbaums erzeugt“ (Wikipedia, XSL Transformation).

Einige Build-Tools, darunter Apache Ant, verfügen über alle notwendigen Libraries, die zur Ausführung einer XSLT-Transformation notwendig sind. So gibt es in Ant einen eigenen <*xslt*> Task, der in einer solchen Situation zum Einsatz kommt. Deshalb spielen XSLT und XML eine bedeutende Rolle als Mechanismen innerhalb des Build-Prozesses, da sie für die Produktion von Berichten und die Formatierung der Outputs für unterschiedliche NutzerInnen sorgen (Lee 2008).

3.2 Make

Make ist ein Werkzeug mit dem Teile von Shell-Skript-Kommandos ausgeführt werden können. Entwickelt wurde Make Ende der Siebzigerjahre von Stuart Feldman in den Bell Laboratories in Murray Hill (vgl. Hatcher & Loughran 2004; Hoffmann 2008). Im Laufe der Jahre wurde das Tool Make kontinuierlich weiterentwickelt und, neben Windows, auch für weitere Betriebssysteme funktionsfähig gemacht und angepasst. Seit dem Zeitpunkt seiner Erfindung bis zur Gegenwart ist Make ein beliebtes Tool geblieben, dass besonders bei C/C++ Projekten häufig Einsatz findet.

Die von Make benutzten Grundeinstellungen werden vorerst in einer Kontrolldatei, dem sog. *Makefile*, abgelegt. Makefiles haben die Besonderheit, dass sie in einer eigenen Beschreibungssprache formuliert werden. Nach Hoffmann werden die Bestandteile eines typischen Makefile folgendermaßen beschrieben (2008):

- Declarations: Deklarationen erlauben die Definition von symbolischen Bezeichnern, welche üblicherweise am Anfang des jeweiligen Makefile angegeben werden.

- Targets: Dies sind Ziele, die im Zuge des Make-Prozesses bestimmt und erzeugt werden und setzen sich im Grunde aus den Namen der Zieldateien, den sog. Unterzielen, zusammen.

- Dependencies: Hier werden die Abhängigkeiten zwischen einem Ziel und seinen Unterzielen festgelegt und beschrieben, wobei all jene Dateien, die für die Erreichung des Ziels erforderlich sind, in einer Abhängigkeitsliste aufgezählt werden.

- Tasks: Sobald das Ziel und seine internen Abhängigkeiten feststehen, werden mit den Shell-Kommandos Aktionen erzeugt, unter anderem der Compiler und Linker aufgerufen, die wiederum zur Übersetzung und Ausführung aller Objektmodule und der finalen Applikationsdatei notwendig sind.

3.2.1 Rekursive Makes

Große Softwareprojekte setzten sich oft aus mehreren Komponenten zusammen, die sich in einem eigenen Unterverzeichnis befinden. Im Falle des rekursiven Make, bekommt jedes einzelne Unterverzeichnis ein eigenes Makefile ohne externe Abhängigkeiten, wobei diese Verzeichnisse über ein weiteres Makefile in einem übergeordneten Verzeichnis verwaltet werden. Das übergeordnete Makefile hat somit die Aufgabe inne, die verschiedenen Makefiles in den Unterverzeichnissen einer bestimmten Reihenfolge entsprechend aufzurufen. „Im Allgemeinen ist es aber effektiver, diese Aufgabe Make zu überlassen, weil man sich damit die Möglichkeit von Abhängigkeiten zwischen den verschiedenen Verzeichnissen offen hält, auch wenn diese nicht gewünscht sind“ (Mecklenburg 2005, S. 118).

3.2.2 Non-rekursive Makes

Als Alternative zu den rekursiven Makes ist es auch möglich, Quelltexte aus mehreren Verzeichnissen mit einem einzigen Makefile zu übersetzen. Diese Variante wird als non-rekursives Make betitelt, bei dem üblicherweise ein großes Makefile zur Sammlung aller Ziele, Abhängigkeiten, Referenzen und zu erstellenden Dateien dient. Der Schöpfer des non-rekursiven Ansatzes, Miller, schlägt eine weitere Vorgehensweise vor, da sich die zuvor Genannte als unvorteilhaft erwiesen hat. Er empfiehlt, dass jedes Verzeichnis mit einer, von Make geladener, Include-Datei versehen wird, die verzeichnisspezifische Dateilisten und modulspezifische Regeln enthält. Somit kann das, im obersten Verzeichnis angelegte, Makefile jene Include-Dateien in den Prozess einbinden (vgl. Miller 1998; Mecklenburg 2005).

3.2.3 Probleme bei Make

An dieser Stelle, soll noch auf wichtige Probleme bzw. Nachteile, die durch den Einsatz von Make entstehen, eingegangen werden. So ist es ein wichtiges Problem, dass Make-basierte Builds nicht portabel und zugleich stark von den Eigenschaften der Systemumgebung abhängig sind. Dieses Problem beruht auf der Tatsache, dass die Befehle im Make-Tool mit Shell-Skripten ausgeführt werden, die nicht von einer Plattform in die Andere direkt übertragbar sind (Neagu 2005).

Eine Schwierigkeit bei der Nutzung von Make stellt die Dokumentation des Builds dar. Außerdem kann die Verwendung eines einzigen Namespace zu einem unüberschaubaren Chaos führen, wodurch es beinahe unmöglich wird, erstellten Dateien einen eigenen Namen zu vergeben. Im Normalfall sollte es bei einem Build kein Problem darstellen, neue Codes mit Hilfe einer einfachen Makefile in das Endprodukt zu integrieren, doch bei komplexeren Projekten entsteht ein großes Geschwindigkeitsdefizit, wodurch das Build nicht so schnell wie gewünscht ausgeführt werden kann. Bei solchen großen Projekten kommen häufig rekursive Makes zum Einsatz, die jedoch bewirken können, dass gewisse Probleme auftreten (Neagu 2005).

Nach Miller werden die möglichen Konflikte übersichtlich zusammengefasst (1998):

- Es erweist sich als besonders schwierig, die Reihenfolge der Rekursion richtig auf die Unterverzeichnisse zu übertragen. Diese Reihenfolge ist oft sehr unstabil und muss zudem regelmäßig manuell eingestellt werden. So kann die Vermehrung der Verzeichnisanzahl oder die Erweiterung der Verzeigungen im Verzeichnisbaum, zu einer noch größeren Instabilität dieser Reihenfolge führen.

- Oftmals ist es notwendig, mehr als nur ein Unterverzeichnis zu überspringen, um das gesamte System zu erstellen. Diese Situation führt verständlicherweise zu verlängerten Build-Zeiten.

- Um lange Laufzeiten der Builds zu vermeiden, werden einige Abhängigkeitsinformationen ausgelassen, denn ansonsten würden die Entwickler-Builds unüberschaubare Wartezeiten in Anspruch nehmen. Dies kann aber dazu führen, dass gewisse notwendige Bereiche nicht zeitgerecht upgedated werden.

- Auch die Abhängigkeiten zwischen den Verzeichnissen werden häufig ausgelassen oder es ist zu schwierig diese auszudrücken. Aus diesem Grund werden die Makefiles weitaus öfter als notwendig in das Build geschrieben, um sicherzustellen, dass keine Information übersprungen wird.

- Die Ungenauigkeit bei den Abhängigkeiten bzw. der Mangel an berücksichtigten Abhängigkeiten kann dazu führen, dass das Produkt nicht einwandfrei erstellt werden kann und der Build-Prozess zusätzlich durch einen Menschen überwacht werden muss.

- Die bereits genannten Probleme führen meistens zu dem Punkt, an dem gewisse Projekte nicht mehr in der Lage sind, verschiedene Makes parallel zu implementieren, da das Build in diesem Fall fehlerhafte Prozesse ausführt.

Beim non-rekursiven Make können zwar auch kleinere Teile des Builds ausgeführt werden, aber da zuvor dennoch die gesamten Build-Beschreibungen gesammelt werden müssen, kann es wiederum zu einer Verlangsamung oder zu unerwünschten Nebeneffekten kommen (Neagu 2005).

3.3 Ant

Ant ist ein Java-basiertes Werkzeug, das seinen Dienst als Build-Tool leistet und weltweit häufige Anwendung findet. Der Begriff Ant ist ein Akronym für *Another Neat Tool*, welches ursprünglich für das Open Source Projekt *Tomcat* entwickelt worden ist. Die Entwicklung des Tools erfolgte als Teil des *Jakarta* Projekts, im Zuge dessen die erste Version von Ant durch James Duncan Davidson im Jahr 1999 erstellt wurde. Die erste eigenständige Version 1.1 von Ant wurde im Juli 2000 veröffentlicht und schließlich im November 2002 als Top-Level-Projekt der *Apache Software Foundation* deklariert (vgl. Popp 2006; Hatcher & Loughran 2004; Wikipedia, Ant).

In der Zeit vor der Entwicklung des Tools Ant war der Build-Prozess größtenteils plattformabhängig. Hüttermann definiert Ant als ein *Schweizer Taschenmesser*, denn es ist „sehr mächtig und erweiterbar und unterstützt den Entwickler dabei, viele einzelne Schritte zu automatisieren“ (2008, S. 128).

Zu den wichtigsten Neuerungen, die durch Ant eröffnet wurden, zählt die Plattformunabhängigkeit, welche durch den Einsatz der Java-Technologie und der verwendeten Skriptsprache bedingt ist. Durch diese Eigenschaft ist die Voraussetzung gegeben, dass der Build-Prozess somit auch portabel ist. Ein äußerst wichtiger Faktor zur Ermöglichung dieser Vorteile ist, dass die Ant-Skripte nicht mehr auf Shellskript-ähnlichen Kommandos, sondern auf XML basieren (vgl. ant.apache.org, Introduction; Hatcher & Loughran 2004).

Die wesentlichen Leistungen von Ant ergeben sich auf Grund der Nutzung der Programmiersprache XML und der Java-Technologie. So ist die Verwendung des Dateiformats XML selbst für einen Laien, nach einer kurzen Anwendungszeit, einfach zu bedienen. Zudem kommen durch Java bestimmte Erweiterungsmöglichkeiten hinzu, wie

die mögliche Implementierung spezieller Funktionen, die für den projektspezifischen Build-Prozess notwendig sind. Daraus folgt, dass innerhalb von Java Projekten der Build sehr schnell vollzogen werden kann, da „Java-Programme innerhalb derselben Java-Virtual Machine ausgeführt werden können wie Ant selbst“ (Hatcher & Loughran 2004, S. 35).

Obwohl Ant vorzugsweise bei Java-Entwicklungsprojekten zum Einsatz kommt, bleibt sein Einsatzspektrum nicht nur damit begrenzt, denn dieses Tool kann bei Projekten mit jeglichen Programmiersprachen und Technologien verwendet werden. Es werden jedoch auch Alternativen wie NAnt angeboten, die speziell bei .NET-Entwicklungen besonders empfohlen werden (Popp 2006).

3.3.1 Ant – Architektur

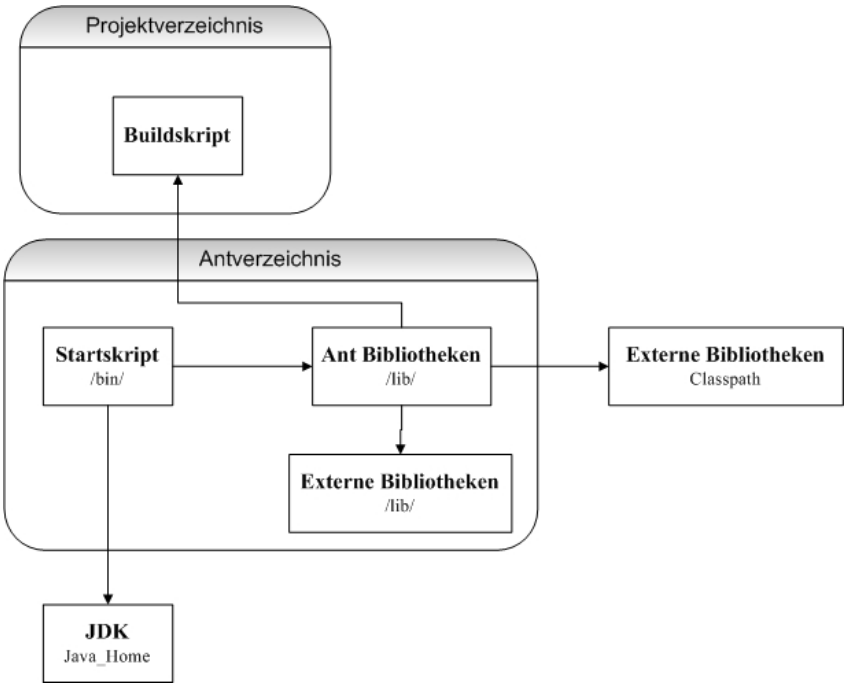


Abbildung 8: Ant-Architektur (nach Popp 2006, S. 75)

Grundsätzlich besteht Ant aus einer Reihe von Java-Bibliotheken und dem Startskript. Abb. 8 gibt einen Überblick über die wichtigsten Bestandteile einer Ant-Installation: Da Ant auf Java basiert, ist für die Ausführung ein installiertes *Java Development Kit* (JDK) notwendig. Das Startskript von Ant wird bei Windows über das Skript *ant.bat*, bei Unix mit *ant* aufgerufen und befindet sich im Installationsordner. Mit diesem Skript wird Ant in der *Java Virtual Machine* (JVM) ausgeführt. Nach dem Start wird seitens Ant das Build-Skript *build.xml* aufgerufen, wobei die Folgeschritte im Prozess, durch die festgelegten Build-Ziele bestimmt werden. Die für Ant erforderlichen Java-Bibliotheken, sowie externe

Bibliotheken können im Unterverzeichnis *lib* aufgefunden werden, wobei für externe Bibliotheken daneben die Möglichkeit besteht, dass diese über die %CLASSPATH%-Umgebungsvariable bereitgestellt werden (Popp 2006).

3.3.2 Build-Skript

Da Ant standardmäßig bei Build-Prozessen eingesetzt wird, werden dessen Skripte auch als Build-Dateien bezeichnet. Diese werden zuerst von Ant über die Datei *build.xml* aufgerufen und haben die Aufgabe Ant zu steuern. In Anlehnung an die Abb. 9 soll hier der hierarchische Aufbau der grundlegenden Elemente eines Build-Skripts erläutert werden:

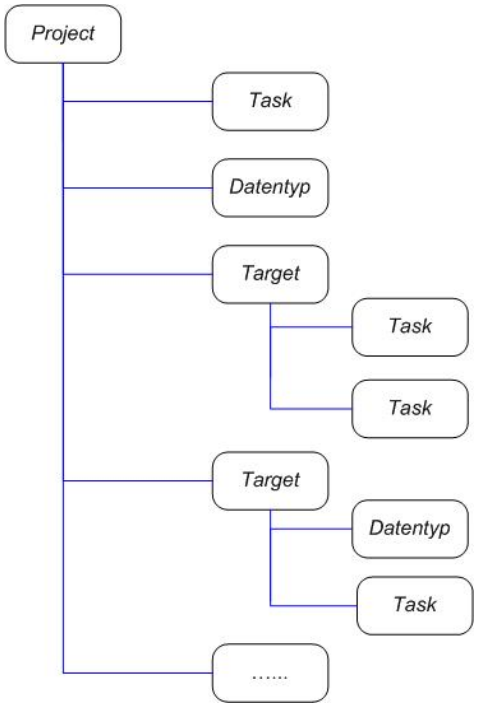


Abbildung 9: Struktur des Ant Build-Skripts

Zuallererst ist das Projekt das oberste Element, welches alle untergeordneten Elemente in sich aufnimmt und koordiniert. Daneben ist es bei größeren Projekten möglich, dass diese aus mehreren kleineren Projekten mit eigenen Buildfiles bestehen, wobei diese Unterprojekte über ein übergeordnetes Buildfile gesteuert werden können (Hatcher & Loughran 2004).

Als nächstes werden die Targets des Projekts definiert, sodass die Ziele für jeden einzelnen Schritt des Build-Skripts festgelegt werden. Diesbezüglich ist es wichtig zu beachten, dass

bei Ant, im Gegensatz zu Make, keine Dateiabhängigkeiten oder Umwandlungsregeln für Dateien existieren, sondern lediglich Targets untereinander in einer gewissen Abhängigkeitsbeziehung stehen. So werden diese Abhängigkeiten in Ant durch Zuhilfenahme des Attributes *depends* formuliert, womit die Targets nach der Reihenfolge ihrer Abhängigkeiten, und jeweils nur ein Mal, aufgerufen werden. Insofern werden die Verfügbarkeiten von Properties mit den Attributen *if* und *unless* überprüft. Das Attribut *if* verlangt, dass für die Ausführung des Targets ein Property existiert. *Unless* hingegen bewirkt die Ausführung des Targets, wenn das Property nicht existent ist. Innerhalb der Build-Datei müssen die einzelnen Targets durch einen eindeutigen Namen identifizierbar sein. Da aber jede einzelne Build-Datei ihre eigenen Targets besitzt, können dieselben Namensbezeichnungen problemlos auch in einer anderen Build-Datei verwendet werden (vgl. Matzke 2005; Hatcher & Loughran 2004; Holzner 2005).

Tasks gehören zu den Bestandteilen eines Targets und bezeichnen einen Teil des ausführbaren Codes. Es existiert eine große Menge an vorgefertigten Tasks, die sowohl von Ant, als auch von externen Bibliotheken zur Verfügung gestellt werden. Ein Task kann aus mehreren Attributen bestehen, wobei der Wert eines Attributes eventuell Informationen über das Property enthalten kann. Bevor der Task ausgeführt wird, werden diese Informationen ausgewertet. Ein wichtiger Vorteil entsteht dadurch, dass bei Ant zahlreiche vorgefertigte Tasks (*built-in-tasks, optional-tasks*), sowie selbstständig entwickelte neue Tasks, mit Leichtigkeit hinzugefügt werden können (vgl. Matzke 2005; Hatcher & Loughran 2004; Holzner 2005).

Nach der im Target festgelegten Reihenfolge ruft Ant die notwendigen Tasks auf, die selbst entscheiden können, ob sie ausgeführt werden oder nicht. Daraus ergibt sich der Vorteil, „[…] dass die Tasks mehr domänenspezifisches Wissen enthalten können als das Build-Werkzeug, wie beispielsweise die Ausführung einer Abhängigkeitsprüfung, bei der die Verzeichnishierarchie berücksichtigt wird, oder sogar die Lösung von Abhängigkeitsproblemen über ein Netzwerk: Die FTP- und HTTP-Tasks von Ant können die Abhängigkeitsprüfung verwenden, um das Herunter- oder Hinaufladen von Dateien zu steuern“ (Hatcher & Loughran 2004, S. 42). Daneben gibt es auch die Möglichkeit, dass Tasks außerhalb der Targets bereitgestellt und immer dann ausgeführt werden, wenn eine Build-Datei aufgerufen wird (Matzke 2005).

Properties können im Allgemeinen mit Konstanten herkömmlicher Programmiersprachen verglichen werden, denn sie sind Platzhalter für Werte, die, einmal festgelegt, nicht mehr abgeändert werden können. Der Geltungsbereich von Properties ist von globalem Umfang, d.h. sie können „zu jedem beliebigen Zeitpunkt der Ausführung einer Build-Datei erzeugt werden“ (Matzke 2005, S. 23). Nachdem der Wert eines Properties definiert wurde, kann dieser mittels $ (Name) an jeglicher Stelle im Build-Skript abgefragt werden (Popp 2006).

Der Einsatzbereich der Properties umfasst vor allem zwei Aufgabenbereiche. Zum Einen ist dies die Anpassung der Ablaufumgebung der Build-Datei an die reale Umgebung mittels geeigneter Pfadangaben. Zum Anderen wird der Ablauf innerhalb einer Build-Datei beeinflusst, indem das Property als Flag benutzt wird (Matzke 2005).

Wie in der Ant-Dokumentation (ant.apache.org, Property) beschrieben, gibt es grundsätzlich 6 Arten, um Properties zu definieren:

- Durch die gleichzeitige Bereitstellung der Attribute *name* und *value*.

- Durch die Breitstellung der Attribute *name* und *refid*.

- Indem festgelegt wird, dass das *file* Attribut gemeinsam mit dem Dateinamen der Property-Datei geladen wird. Diese Property-Datei hat jenes Format, welches zuvor durch die Datei in der *java.util.Properties* Klasse definiert wurde und muss dieselben Regeln befolgen, um non-ISO8859-1 Zeichen zu umgehen.

- Indem das *url* Attribut gemeinsam mit dem url festgelegt wird, über welches das Property geladen werden soll. Durch die Datei, die in der *java.util.Properties* Klasse verwendet wurde, wird definiert, welches Format die Datei haben muss, zu der diese url gelenkt wird.

- Indem bestimmt wird, dass das *resource* Attribut gemeinsam mit dem Resource-Namen der Property-Datei geladen wird. Eine Ressource ist eine Property-Datei im aktuellen *classpath* oder im spezifizierten classpath.

- Indem das *environment* Attribut mit einem Präfix versehen wird. Properties werden für jede Umgebungsvariable definiert, indem der gelieferte Name und ein Zeitraum für den Namen der Variable als Präfix vorgestellt wird.

Neben der Möglichkeit, Properties selbst zu definieren, können auch vordefinierte Properties herangezogen werden, die in großer Anzahl zur Verfügung stehen.

Abschließend soll noch kurz der Datentyp angesprochen werden, welches im speziellen Fall von Ant als komplexes Objekt, das Daten beinhaltet, definiert werden kann. Der Datentyp ist keineswegs passiv und hat somit die Funktion inne, Daten zu beschaffen. Auf diesen Daten werden jedoch keine Aktionen durch den Datentyp durchgeführt, sondern diese werden nur für die umgebenden Tasks bereitgestellt. (Matzke 2005)

3.4 Maven

Der erste Prototyp von Maven ist im August 2001 ins Leben gerufen worden. Der Anstoß zur Entwicklung von Maven kam während des *Jakarta Alexandria* Projekts und wurde später im *Turbine* Projekt weitergeführt. Maven wurde erstmals im Zuge des Turbine Projekts getestet und ausgeführt. Davor wurden die unterschiedlichen Subsysteme des Projekts mit Ant mittels verschiedener Builds gesteuert, wobei es dadurch zu einer mühevollen Instandhaltung mehrerer Builds kam, die generell dieselbe Funktionalität hatten (vgl. Massol et al. 2008; O'Brien et al. 2008; van Zyl 2005).

Der Ausgangspunkt für die Erstellung von Maven war jener, dass ein Modell für ein Projekt erschaffen werden sollte, in dem alle projektrelevanten Dokumente an demselben Ort auffindbar sind. Daneben war es Ziel, eine standardisierte Verzeichnisstruktur anzulegen, damit notwendige Informationen nicht in Bibliotheken, Quellen und Dokumentationen aufgestöbert werden müssen (van Zyl 2005).

Die erste Version von Maven 1.0 ist im Sommer 2004 von der *Apache Software Foundation* veröffentlicht worden. Die Version 2.0 folgte im Oktober 2005 und basiert auf einer kompletten Überarbeitung und konzeptionellen Änderungen der Vorgängerversion (Popp 2006).

Maven kann simplifizierter Weise als Build-Tool bezeichnet und auch als solches genutzt werden, aber diese Beschreibung erweist sich nicht als ausreichend, denn dieses Werkzeug vereint eine Reihe von weiteren Funktionen in sich. Der Begriff „*Project Management Framework*" (O'Brien et al. 2008) zur Benennung dieses Tools ist jedoch irreführend, da es an dieser Stelle einer konkreteren Definition Mavens bedarf.

Wie kann also Maven beschrieben werden? Maven besteht grundsätzlich aus einer Sammlung von Standards, einem Repository Format und ist im eigentlichen Sinne eine Software, die zur Verwaltung und Beschreibung eines Projekts herangezogen werden kann. So dient es zur Bereitstellung eines einheitlichen Lebenszyklus für das Building, Testen und das Deployment von Projektartefakten. Im Allgemeinen stellt Maven ein Framework zur Verfügung, das eine Reihe von Vorteilen mit sich bringt. Insofern erleichtert es die Wiederverwendung von geläufiger Build Logik, vereinfacht die Builds, die Dokumentation, Auslieferung, sowie den Deployment Prozess. Maven bietet ein umfassendes Model, welches bei allen Arten von Softwareprojekten verwendet werden kann. Es stellt zudem ein gemeinsames Layout zur Projektdokumentation bereit, das durch die Standardisierung der Verzeichnisstrukturen zustande kommt, sodass projektspezifische Abhängigkeiten mit Leichtigkeit abgefragt werden können. Da Maven eine gemeinsame Sprache bietet, ist es für Mitglieder eines Teams einfacher, das Projekt als Ganzes wahrzunehmen. Es ist möglich das Projekt auf einer höheren Abstraktionsebene zu betrachten, wodurch eine effizientere Kommunikation zwischen den Teammitgliedern

begünstigt wird, die sich somit wichtigeren Tätigkeiten auf der Anwendungsebene widmen können (vgl. Massol et al. 2008; O'Brien et al. 2008). Bezugnehmend auf die vorangehende Beschreibung lautet die passendere Bezeichnung für Maven daher *„Build Management Framework"* (Ulrich 2008).

3.4.1 Grundprinzipien von Maven

Es gibt vier Grundprinzipien, auf denen Maven beruht (Massol et al. 2008):

- Konvention über Konfiguration
- Deklarative Nutzung
- Wiederverwendung der Build Logik
- Konsequente Organisation von Abhängigkeiten

Im Folgenden sollen diese Prinzipien genauer diskutiert werden.

3.4.1.1 Konvention über Konfiguration

Der Ansatz Konvention über Konfiguration wird seit geraumer Zeit von der *Ruby on Rails* (ROR) Gesellschaft tradiert, allen voran von David Heinemeier Hansson, dem Gründer von ROR (Massol et al. 2008). In Rail wird die Konvention befolgt, dass „sinnvolle Standard-Einstellungen für das Zusammenspiel der einzelnen Komponenten einer Applikation gegeben sind" (Morsy & Otto 2008, S. 31). So macht man sich mit Rails die Flexibilität auf infrastruktureller Ebene zu Nutze, um auch Flexibilität auf der Applikationsebene zu gewinnen.

Maven bietet eine sinnvolle standardisierte Strategie für die geläufigsten Aufgaben im Build Prozess und erspart damit die mühevolle Auseinandersetzung mit banalen Details. Es gibt grundsätzliche Konventionen bei Maven, wie die Verwendung einer standardisierten Verzeichnisstruktur für Projektinhalte, die dadurch einheitlich gegliedert werden können. EntwicklerInnen steht es jedoch frei, bei Bedarf ihre eigene Verzeichnisstruktur zu erstellen und sich über jene von Maven hinwegzusetzen. Bei einer solchen Vorgehensweise muss jedoch in Kauf genommen werden, dass sich die Komplexität des POM steigert und der Build Prozess unüberschaubar wird (vgl. Ulrich 2008; Massol et al. 2008).

Eine weitere Konvention, die seitens Maven befolgt wird, ist die Anfertigung eines einzigen primären Outputs Artefakts pro Projekt. Dabei wird das Prinzip *Seperation of Concerns* (SoC) angewandt, um verschieden Anteile eines Problems, die eine

differenzierte Vorgehensweise verlangen, zu bestimmen und diese in mehrere eigenständige Projekte zu gliedern. So würde jedes *concern* als eigenes Projekt behandelt werden, dem ein einziges Output Artefakt zugeordnet wird. Ein solches Szenario ermöglicht, dass die Komplexität eines Projekts verringert werden kann, denn der separate Einblick und Zugriff auf nützliche Codes kann in großem Ausmaß die Faktoren der Wiederverwendbarkeit, Instandhaltbarkeit, Anpassungsfähigkeit und Erweiterbarkeit begünstigen (Massol et al. 2008; Ulrich 2008).

Daneben gibt es in Maven auch eine standardisierte Namenskonvention für Verzeichnisse und primäre Output Artefakte eines Projekts. Hier ist es wichtig, dass die Vergabe von Namen, unter Einbeziehung von wesentlichen Informationen, einem logischen Prinzip folgt, sodass die Bezeichnung einer Datei eindeutig auf die Art, die Version und den Standort des Artefakts schließen lässt. (Massol et al. 2008)

3.4.1.2 Deklarative Nutzung

Individuelle Projektstrukturen und -inhalte werden bei Maven im *Project Object Model (POM)* angegeben, dem ein deklarativer Ansatz zugrunde liegt.

- **POM**

Analog zum *build.xml* in Ant, besteht das Herzstück von Maven aus einer XML-Datei, dem *pom.xml*. Jedoch divergiert bei den genannten Tools nicht nur der Name der Dateien, sondern auch ihre Funktionsweise. So beruht das POM auf einem modelbasierten deklarativen Ansatz und enthält alle wichtigen Informationen, die zur Beschreibung eines Projekts („building the build“ [2]) notwendig sind (vgl. Massol et al. 2008; Redmond 2006; Smart 2005).

Dass Maven eine modelbasierte Software ist, bedeutet, dass es über ein Projektmodell verfügt, welches alle erforderlichen Metadaten eines Projekts enthält. Stehen im POM alle notwendigen Informationen zur Anfertigung des Produkts bereit, so übernimmt Maven automatisch die Umsetzung des Build-Prozesses (Popp 2006).

Abb. 10 zeigt den hierarchischen Aufbau der POMs, wobei als oberste Instanz ein Super POM existiert, von dem alle weiteren POMs abgeleitet werden. Dieses Super POM enthält alle wichtigen Standardinformationen, die Maven für die Ausführung des Build-Prozesses benötigt. Desweiteren kann ein benutzerdefiniertes Parent POM angelegt werden, dessen festgelegte spezifische Werte an alle weiteren POMs weitervererbt werden (vgl. Ulrich 2008; Massol et al. 2008; Redmond 2006)

[2] Massol et al. 2008, S. 18.

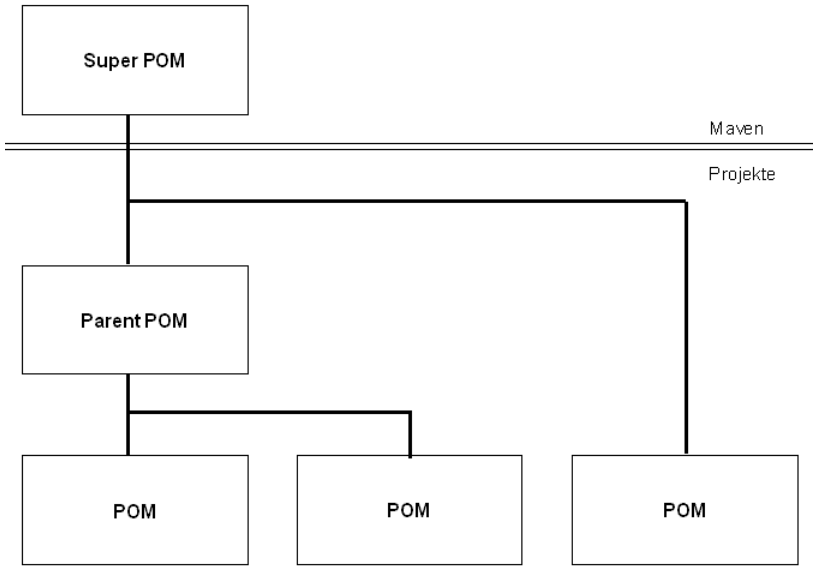


Abbildung 10: POMs (nach Vehns 2008)

Ein POM muss mindestens folgende Informationen enthalten: *project root, modelVersion, groupId, artifactId* und *version*. Dabei werden vom Super POM bzw. Parent POM bestimmte Einstellungen weitervererbt: *dependencies, developers and contributors, plugin list (including reports), plugin executions with matching ids, plugin configuration* und *resources* (maven.apache.org, POM).

In Abb. 11 wird die Grundstruktur eines POMs skizziert, das aus den Basiseinstellungen, Build-Einstellungen, Projektinformationen und Build Environment-Einstellungen besteht (Redmond 2006):

Im Abschnitt Basiseinstellungen werden die Abhängigkeiten eines Projekts von Artefakten anderer Projekte beschrieben. Insofern finden hier die automatische Definition von Abhängigkeitskoordinaten und die Weitergabe von festgelegtem Erbmaterial der Parent bzw. Super POMs an alle weiteren POMs statt. Daneben werden gegebenenfalls auch die Abhängigkeitsbeziehungen von und zwischen vorhandenen Submodulen angegeben.

Die Elemente, die in der Projektinformation angegeben werden, übernehmen lediglich bei der Dokumentation von Plugins eine wichtige Rolle. Insofern spielen diese Projektinformationen keine aktive Rolle in der Konfiguration, sondern können eher als Teile des Build-Prozesses angesehen werden, die aber nicht unbedingt für die Ausführung der Builds notwendig sind. Vielmehr geben sie allgemeine lesbare Informationen zum Projekt wieder, die zum Informationsaustausch und zur Kommunikation innerhalb des Projektteams dienlich sind.

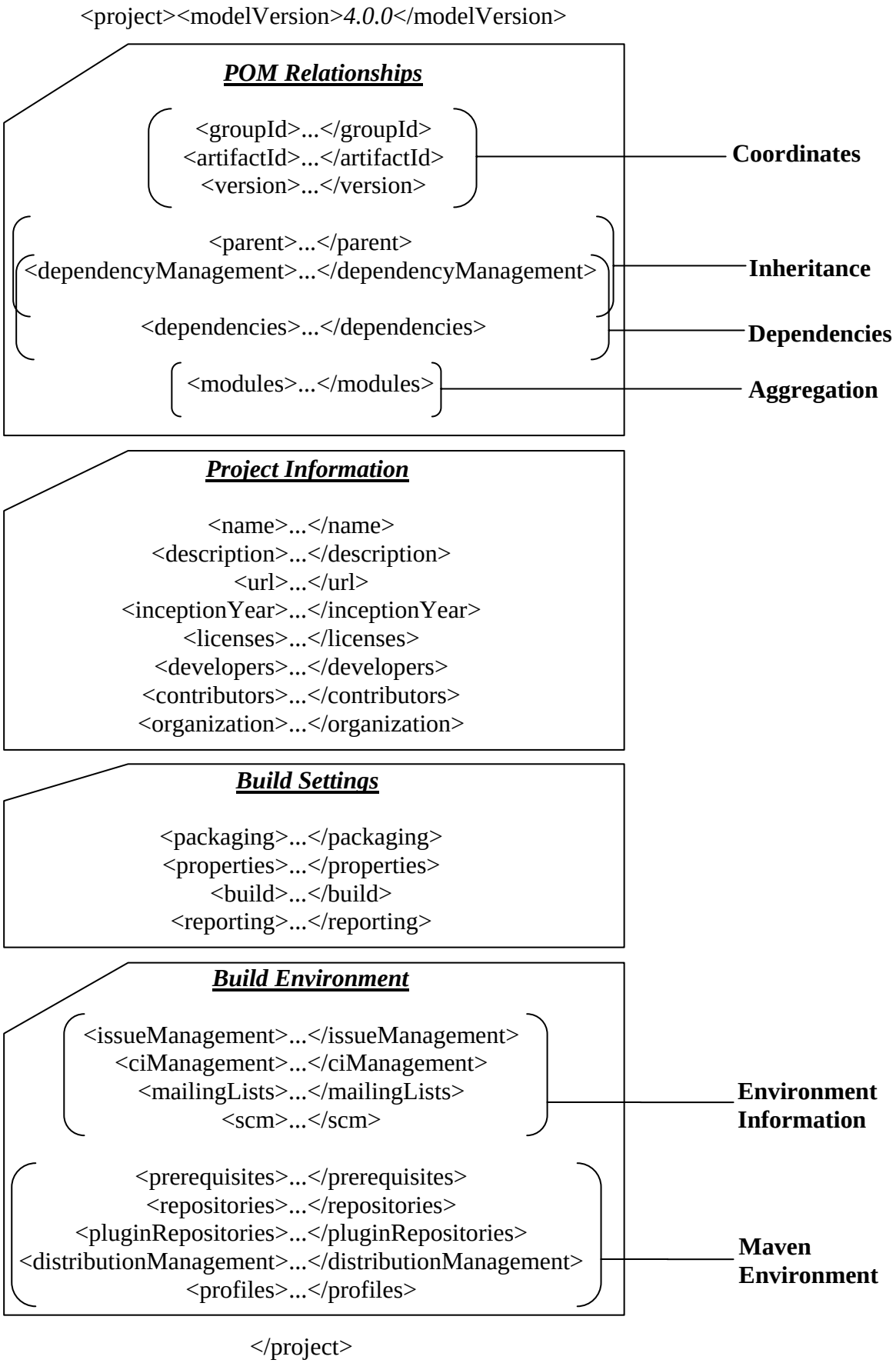


Abbildung 11: Grundstruktur eines POMs

Unter der Rubrik Build-Einstellungen kann der Verlauf des Standard-Builds von Maven individuell eingerichtet werden. D. h. hier können der Standort der Quellen und der Tests geändert, neue Plugins hinzugefügt, dem Build Lebenszyklus weitere Plugin-Ziele angefügt und die Parameter der Seitengenerierung angepasst werden.

Das Build-Environment enthält Profile und Einstellungen, die für die Anpassung an mögliche unterschiedliche Umgebungen gedacht sind. Im Build-Environment werden die Build Einstellungen der individuellen Umgebung angepasst und durch ein benutzerdefiniertes *settings.xml* in *~/.m2* ergänzt.

- **Build Lebenszyklus**

In allen Softwareprojekten wird im Build-Prozess ein ähnlicher Weg beschritten, wie z.B. die Vorbereitung, Kompilierung, Testung, Verpackung und Installation des Produkts. Diese Kette von Schritten wird in Maven als *Build Lebenszyklus* bezeichnet, der aus einer Abfolge von bestimmten Phasen besteht. In jeder dieser Phasen werden eine bzw. mehrere Handlungen durchgeführt oder relevante Ziele umgesetzt. Dem standardisierten Build Lebenszyklus entsprechend, werden diese einzelnen Phasen aufgerufen. Maven muss lediglich wissen, welche Phase(n), die von AnwenderInnen zuvor spezifiziert und festgelegt werden müssen, es auszuführen hat und übernimmt den Rest der Arbeit. Daneben ist bei Bedarf die Erweiterung des standardisierten Build Lebenszyklus durch zusätzliche Plugins möglich. So ist es problemlos zu bewerkstelligen, dem Build Lebenszyklus gewünschte Funktionalitäten hinzuzufügen, indem entweder bereitgestellte Plugins verwendet, oder ein benutzerdefiniertes Plugin erstellt und im *pom.xml* definiert wird (vgl. Ulrich 2008; Massol et al. 2008).

3.4.1.3 Wiederverwendung der Build Logik

Das Prinzip der Wiederverwendbarkeit wird bei Maven ermöglicht, indem die Build Logik in zusammenhängende Module, die sog. Plugins, eingebettet wird. Insofern ist Maven das Framework, in dem die Ausführung von Plugins in einer klar definierten Art und Weise koordiniert wird. Das wichtigste Grundprinzip bei Maven ist die Ausführung der einzelnen Schritte im Build Prozess via Plugins. D.h. Maven verfügt über einen standardisierten Lebenszyklus, der sich in mehrere Abschnitte gliedert, die jeweils von einem zugeordneten Plugin ausgeführt werden (vgl. Ulrich 2008; Massol et al. 2008; maven.apache.org, Plugin Development).

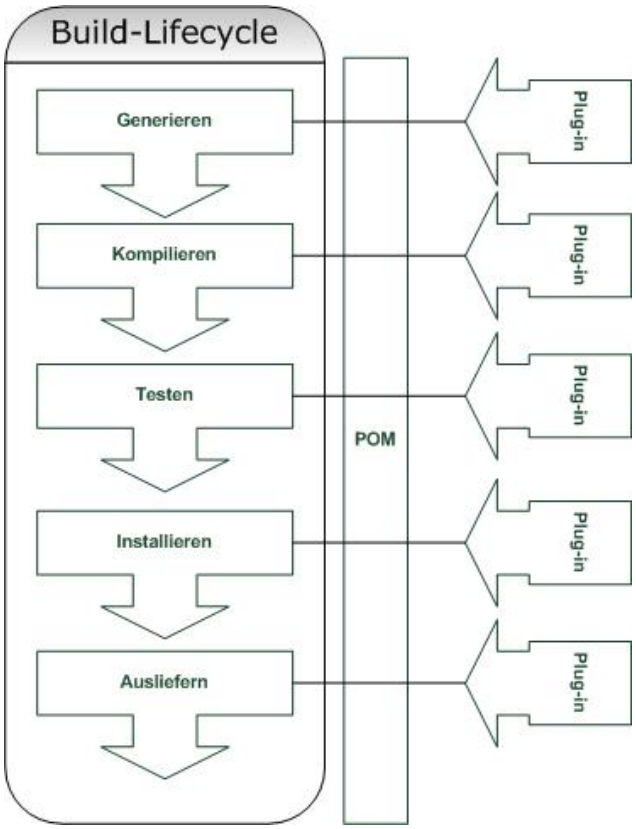


Abbildung 12: Standardisierter Maven-Lebenszyklus

In Abb. 12 ist ein solcher standardisierter Maven-Lebenszyklus dargestellt, wobei anzumerken ist, dass die Ausführung der Plugins über den Build Lebenszyklus nach deklarativem Ansatz gesteuert wird. Insofern werden auch Inputs des Project Object Models, insbesondere die Plugin Konfigurationen im POM herangezogen werden.

Der Faktor der Erweiterbarkeit durch spezielle Plugins, welche zudem die Integration verschiedener Technologien zulassen und vereinfachen, ist eines der wesentliche Vorteile, die uns Maven bietet. Nehmen wir als Beispiel ein .Net Projekt: um Ant für ein solches Projekt kompatibel zu machen, wird NAnt eingesetzt, dass auf einer kompletten Formulierung in der .Net Projektsprache basiert. Das Java-basierte Maven löst dieses Problem hingegen durch die Erweiterung mit dem Plugin NMaven, sodass Maven ohne neuerliche Implementierung in .Net Projekten verwendet werden kann (Ulrich 2008).

3.4.1.4 Konsequente Organisation von Abhängigkeiten

Um die Organisation und Auflösung der Abhängigkeiten erläutern zu können, ist es notwendig zu verstehen, wie Abhängigkeiten, Artefakte und Verzeichnisse in Maven überhaupt strukturiert sind und auf welche Weise sie arbeiten. Vorerst wird in Maven ein

Artefakt als spezifisches Stück einer Software definiert, welches im speziellen Fall von Java-basierter Software häufig eine JAR-Datei ist. Ein Java Artefakt kann jedoch auch als WAR-, SAR- oder EAR-Datei ausgegeben werden. Eine Abhängigkeit kann im Allgemeinen als ein Verweis auf ein spezifisches Artefakt, das in einem Verzeichnis liegt, definiert werden. Abhängigkeiten werden durch verschiedene Parameter zur einfacheren Identifizierung beschrieben, wie der *groupId*, *artifactId* und *version*. Maven benötigt diese Abhängigkeitskoordinaten, um erschließen zu können, in welchem Verzeichnis es nach der jeweiligen Abhängigkeit zu suchen hat. Kurz gesagt, werden die Abhängigkeiten in Maven nach deklarativen Ansatz formuliert (vgl. O'Brien et al. 2008; Massol et al. 2008; Popp 2006).

Das Management von Abhängigkeiten ist eines der bedeutendsten Stärken von Maven. Die genannten Abhängigkeitskoordinaten werden üblicherweise im POM gespeichert, die von dort aus in den internen logischen Abhängigkeitsmechanismus von Maven aufgenommen werden. Wird vom/von der AnwenderIn eine Abhängigkeit formuliert, so wird in allen Repositories nach jenen Artefakten gesucht, die den Anforderungen der Abhängigkeit entsprechen. Das aufgefundene Artefakt wird anschließend vom remote Repository in das lokale Repository kopiert und steht ab nun zur Verfügung. Maven verfügt über zwei Arten von Repositories: das lokale und das remote Repository. Prinzipiell werden Abhängigkeiten im lokalen Repository gesucht, sind sie jedoch nicht vorhanden, wird die Suche sodann in allen remote Repositories fortgesetzt, auf die Maven Zugriff hat (vgl. Popp 2006; Massol et al. 2008).

3.4.2 Architektur von Maven

Maven ist ein Java-basiertes Werkzeug, dessen Architektur sich aus einem kompakten *Kern* und einer beliebigen Anzahl von *Plugins* zusammensetzt. Bei der Installation von Maven findet zu allererst das Kopierverfahren des Kerns auf die Festplatte statt. Dieser Kern verfügt jedoch über keinerlei Funktionalität, kann jedoch als Grundgerüst angesehen werden, mit dem die Interpretation eines Projektmodells und die Festlegung der Build-Phasen möglich ist. Wie bereits bekannt, setzt sich ein Build-Prozess aus mehreren Schritten zusammen, die von einem Build-Ziel ausgeführt werden. Hier übernehmen Plugins nun die Aufgabe, Build-Ziele mit ähnlichen Aufgaben in sich zu vereinen. Nach der primären Installation von Maven befinden sich keinerlei Plugins auf dem lokalen System. Daher ist vor der ersten Ausführung des Build-Prozesses ein Zugriff auf das Internet notwendig, damit Maven mit der automatischen Nachinstallation der erforderlichen Plugins aus den verschiedenen Maven-Repositorys starten kann. Zunächst versucht der Maven-Kern die notwendigen Plugins im *lokalen Repository* unter *.m2\repository* aufzufinden. Sind an dieser Stelle keine Plugins vorhanden, so wird eine Verbindung zum *Plugin-Repository* unter der URL http://repo1.maven.org/maven2 hergestellt. Von dort werden die Plugins auf das lokale Repository kopiert, wo sie von nun

an für den Einsatz im Build-Prozess bereitstehen. Auf dieselbe Weise wird auch beim Aufsuchen von externen Bibliotheken vorgegangen, mit dem einzigen Unterschied, dass in diesem Fall eine Verbindung zum *Remote-Repository* hergestellt wird (Popp 2006). Die gesamte Architektur und geschilderte Funktionsweise von Maven kann der Abb. 13 entnommen werden.

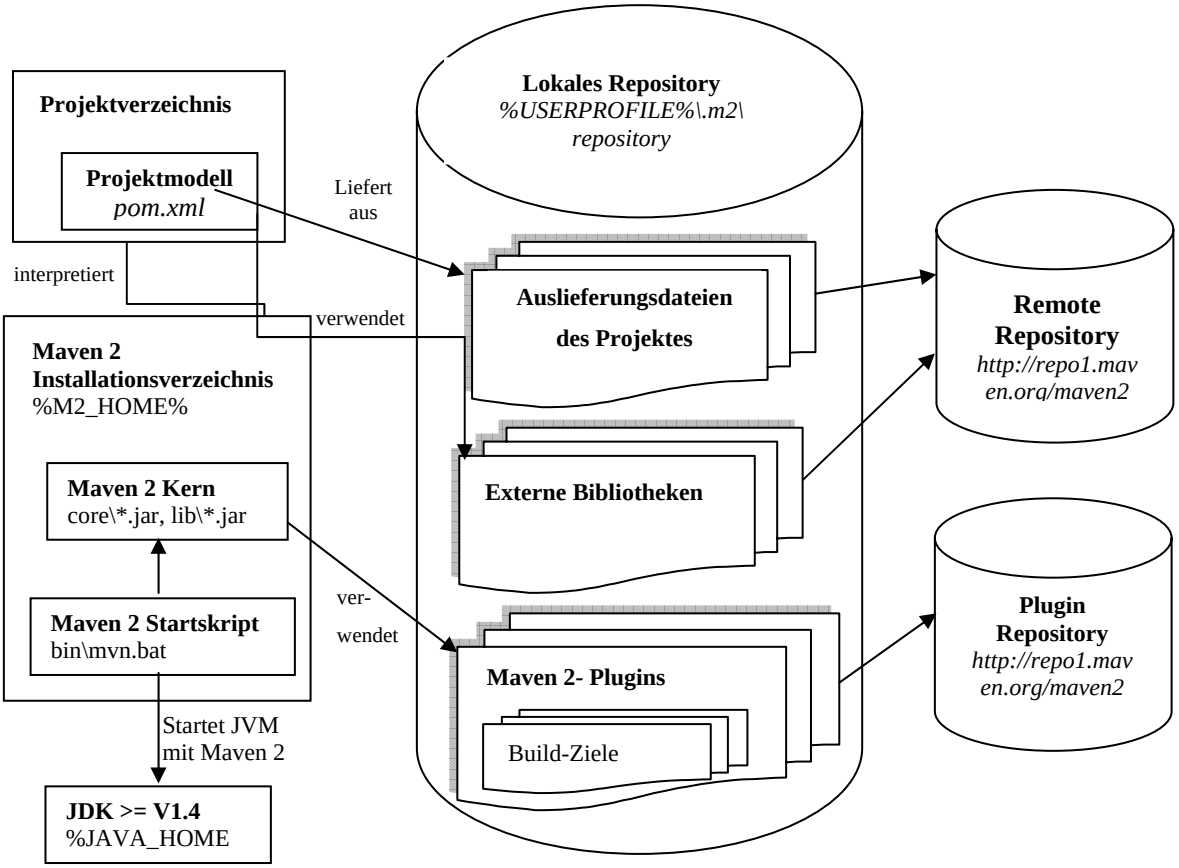


Abbildung 13: Maven Architektur (nach Popp 2006, S. 80)

4. Die Build-Prozess Phasen

Der gesamte Build-Prozess zerfällt in 5 zentrale Teilprozesse, die in Abb. 14 übersichtlich dargestellt sind. Dabei handelt es sich der Prozessreihenfolge nach um Source Control, Kompilierung, Test, Deployment, sowie die Dokumentation und Veröffentlichung der Software. In den folgenden Unterkapiteln werden diese einzelnen Teilprozesse in ausführlicher Form diskutiert.

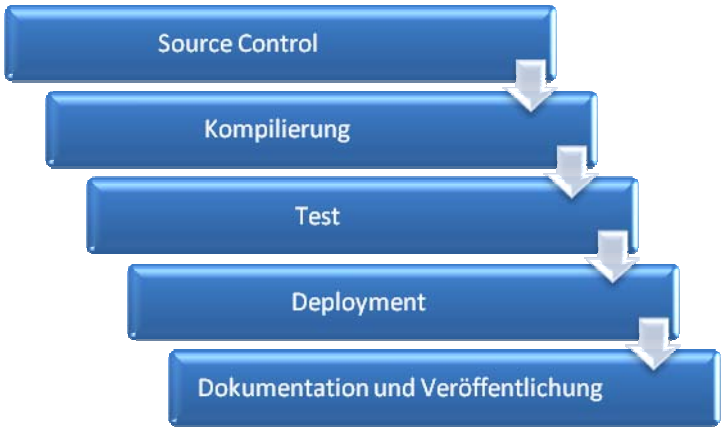


Abbildung 14: Die Build-Prozess Phasen

4.1 Source Control

Source Control ist eines der wichtigsten Verfahren, das für das Build-Management benötigt wird. Neben der geläufigen Bezeichnung als *Source Control* gibt es eine Reihe von weiteren Benennungen, wie *Version Control System (VCS), Source Code Management* und *Revision Control* (Wikipedia, Revision Control). Der Quellcode stellt das Basiselement der Softwareentwicklung dar und ist von besonderer Bedeutung für das Softwareentwicklungsteam. Mittels Source Control kann der Quellcode gespeichert und archiviert werden, sodass der frühere Zustand des Quellcodes jederzeit aufgerufen und wiederhergestellt werden kann. Außerdem eröffnet sich durch Source Control die Möglichkeit, dass EntwicklerInnen zur gleichen Zeit an verschiedenen Aufgaben arbeiten können, ohne sich gegenseitig im Weg zu stehen. Der von den einzelnen EntwicklerInnen verzeichnete Arbeitsfortschritt kann schließlich im Nachhinein zu einem Ganzen zusammengefügt (sog. *merging*) werden (Sink 2004). In den folgenden Unterkapiteln werden neben den Konfigurationselementen und der Verzeichnisstruktur, das Repository mitsamt dem Check-out und Check-In Verfahren zu behandeln sein.

4.1.1 Konfigurationselemente

Konfigurationselemente sind atomare Objekte, die alle Bestandteile eines Projekts bilden und zusammengefasst das Produkt ergeben. Atomare Objekte sind als Einheiten zu verstehen, die im jeweiligen Projektzusammenhang keiner weiteren sinnvollen Unterteilung unterzogen werden können. Beispielsweise kann ein Quellcode als eigenständiges Konfigurationselement definiert werden, wobei durch projektspezifische Anforderungen auch Komponenten des Quellcodes als Konfigurationselemente fungieren können. Wichtig ist in jeglicher Situation das Resultat, bei dem die Konfigurationselemente das Produkt bilden und dessen vollständige Auslieferung gewährleisten müssen (vgl. Popp 2006; Glinz 2005).

Nach ANSI/IEEE Standard 1042-1987 wird ein Konfigurationselement folgendermaßen definiert: „A software configuration item (SCI) is an aggregation of software designated for configuration management and is treated as a single entity in the SCM process."

Nachdem die Konfigurationselemente einmal ausgewählt worden sind, sollten diese einer sinnvollen Strukturierung unterzogen werden. Als Beispiele für Konfigurationselemente kommen in Frage:

- Quellcode
- Build-Skripte
- Testdaten

4.1.2 Verzeichnisstruktur

Durch die zahlreichen Inputs und Outputs, die innerhalb eines Build-Prozesses benötigt werden, ergibt sich eine wichtige Fragestellung: Woher können erforderliche Inputs für den Build-Prozess entnommen werden und was sind die geeigneten Auswahlkriterien hierfür?

Auf die erforderlichen Konfigurationselemente und Ressourcen kann über das Repository zugegriffen werden, wobei hier unabhängig von der NutzerInnenanzahl sehr große Datenmengen problemlos verwaltet werden können. Zur Herstellung einer sinnvollen Struktur der Konfigurationselemente ist die Festlegung einer Hierarchie der Verzeichnisse eine wesentliche Voraussetzung. Zur Erstellung dieser Verzeichnishierarchie gibt es zahlreiche Möglichkeiten, wobei es primär wichtig ist, dass die in den Verzeichnissen abgelegten Dateien einen temporären Charakter haben. Die Festlegung der Verzeichnishierarchie kann durch bestimmte Faktoren, wie Projektorganisation, Softwarearchitektur und herangezogene Technologien (CVS, Subversion, Maven, etc.) beeinflusst werden (Popp 2006).

- **Projektorganisation**

Conway spricht von einer Projektorganisation, bei der sich die Struktur des Systems analog zur Organisation des Projekts entwickelt. Er definiert dieses System wie folgt (1968):

"Any system of consequence is structured from smaller *subsystems* which are interconnected. A description of a system, if it is to describe what goes on inside that system, must describe the system's connections to the outside world, and it must delineate each of the subsystems and how they are interconnected. Dropping down one level, we can say the same for each of the subsystems, viewing it as a system. This reduction in scope can continue until we are down to a system which is simple enough to be understood without further subdivision."

Besonders bei großen Projekten, die über räumliche Distanzen getrennte Teilteams beherbergen, sollte die Gestaltung der Projektstruktur dementsprechend angepasst und in Teilprojekte gegliedert werden. Abhängig von dieser Projektorganisation müssen somit die einzelnen Teilprojekte bei der Verzeichnishierarchie als Grundlagen miteinbezogen werden. In Abb. 15 ist eine solche Projektstruktur mit Teilprojekten beispielhaft dargestellt, wobei der Vorteil dieser Strukturierung darin liegt, dass jedes Teilteam nur am jeweils vorgesehenen Ast des Projekts arbeitet. Da die einzelnen Teams jedoch keine Einsicht in andere Teilbereiche haben und einen voneinander unabhängigen Arbeitsfortschritt zurücklegen, können eventuelle Probleme daraus resultieren, da alle Komponenten und Subsysteme grundsätzlich zueinander passen und sich ergänzen sollten. Wird dieser Aspekt vernachlässigt, so kann die gesamte Architektur des Projekts darunter leiden (Popp 2006).

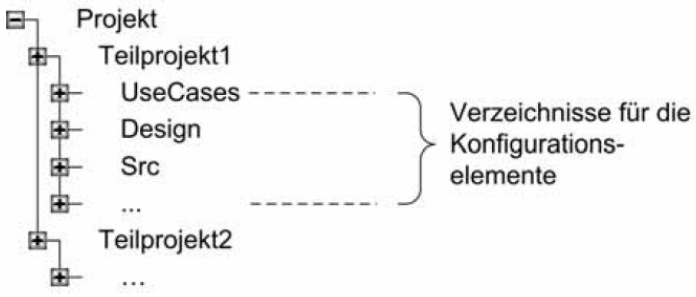


Abbildung 15: Projektstruktur mit Teilprojekten (aus Popp 2006, S. 32)

➢ Softwarearchitektur

Die Software-Architektur kann im Allgemeinen als ein Gerüst definiert werden, das die Beschreibung von Elementen, aus denen Systeme aufgebaut sind und die Interaktion zwischen diesen Elementen enthält. Desweiteren beinhaltet es Muster, welche die Zusammensetzung der genannten Elemente steuern und auch Änderungen im Bezug auf diese Muster festhalten. Es ist möglich, ein System durch die Sammlung von Komponenten und deren Interaktionen zu beschreiben (Shaw und Garlan 1996, S. 1).

Ausgehend von dieser Definition, ergibt sich der Schluss, dass Subsysteme oder Komponenten, welche sich aus Elementen mit inhaltlichen Gemeinsamkeiten zusammensetzen, die Basis für die Hierarchie der Verzeichnisstruktur bilden sollten. Zur strukturellen Gestaltung eines Projekts gibt es verschiedenste Möglichkeiten, die vor allem aus der Notwendigkeit hervorgegangen sind, um mit der wachsenden Komplexität von großen Projekten umgehen zu können. Die vorherige Analyse und der Entwurf von Komponenten und Subsystemen stellen die Grundlage für eine sinnvolle Gliederung und ein funktionierendes Zusammenspiel dieser Komponenten dar (vgl. Zuser et al. 2004; Popp 2006).

In Abb. 16 ist eine wohlgegliederte Projektstruktur zu sehen, in der Subsysteme untergebracht sind. Diese Subsysteme bzw. Komponenten sind unter der Ebene der Konfigurationselemente eingegliedert. In bestimmten Fällen kann es auch notwendig sein einen übergeordneten Ordner für eine Build-Phase anzulegen, dem alle relevanten Konfigurationselemente hinzugefügt werden.

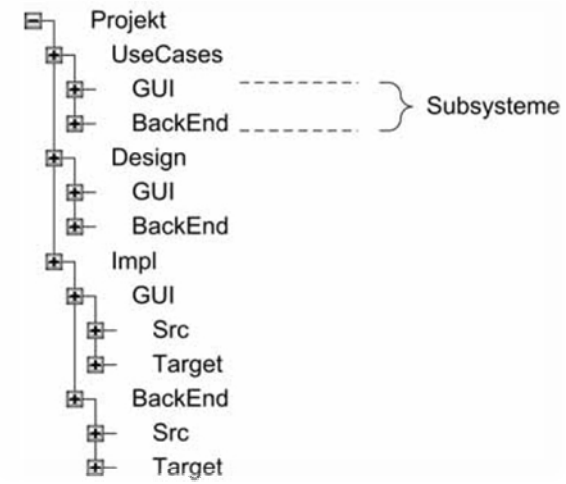


Abbildung 16: Projektstruktur mit Subsystemen (aus Popp 2006, S. 33)

- **Herangezogene Technologien**

Im Zuge der Bestimmung der Verzeichnisstruktur ist es notwendig, Überlegungen zu den verwendeten Technologien anzustellen und deren Anforderungen zu berücksichtigen (Popp 2006). Am Beispiel von Subversion werden Tags und Branches in Form von Verzeichnissen umgesetzt. Als weiteres Beispiel wäre zu nennen, dass in CVS und Subversion ein Verzeichnis für Konfigurationsparameter notwendig ist, das zumeist als *trunk* bezeichnet wird (Collins-Sussman et al. 2008). Bei Maven müssen die Verzeichnisse mindestens aus den Abhängigkeitskoordinaten *groupId*, *artifactId* und *version* [3] bestehen, wobei die groupId mittels Punktabtrennung in weitere Hierarchiestufen unterteilt werden kann.

In Abb. 17 ist die Verzeichnisstruktur des Maven-Repository ersichtlich, wobei eine beispielhafte Gliederung der Hierarchiestufen durch die zugehörigen Abhängigkeitskoordinaten zu sehen ist.

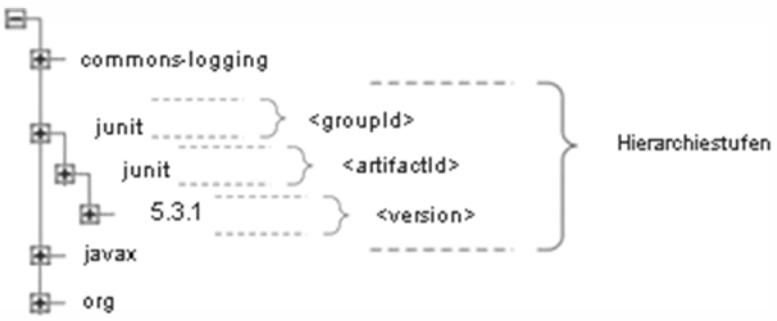


Abbildung 17: Maven-Repository

4.1.3 Repository

Das Repository dient als Ablage, in dem alle Konfigurationselemente gespeichert werden, wobei ihr Dateisystem über eine Verzeichnisstruktur verfügt. Dabei werden Artefakte und Abhängigkeiten unterschiedlicher Typen in Form eines Dateisystembaums dargestellt. Eines der bedeutendsten Vorteile der Ablage stellt die Dateisicherheit dar. Zudem können dadurch unerwünschte Zugriffe verhindert, die Erreichung der Dateien erleichtert und gleichzeitig Änderungen und Erweiterungen durch verschiedene EntwicklerInnen ermöglicht werden. Repositories können grundsätzlich lokal oder remote sein, wobei Beide auf gleiche Weise strukturiert sind (vgl. Popp 2006; Hüttermann 2008; Richardson & Gwaltney 2006).

[3] Siehe Kapitel 3.4.1.4 zur Erläuterung der Abhängigkeitskoordinaten groupId, artifactId und version.

Außerdem enthält das Repository einen internen Versionierungsmechanismus, welcher als Versionskontrolle bezeichnet wird. Eine zentrale Rolle spielt die Versionierung bei der Verwaltung von Konfigurationselementen. Der wesentliche Nutzen der Versionierung ist, dass unerwünschte Änderungen und Erweiterungen nicht ausgeführt werden und gelöschte Daten oder Ordner zu einem späteren Zeitpunkt noch verfügbar sind. Daneben wird eine Übersicht zum Projektstatus zur Verfügung gestellt, indem angegeben wird, welche EntwicklerInnen zu welchem Zeitpunkt Änderungen bzw. Erweiterungen an Dateien vorgenommen haben. Im Vordergrund steht außerdem, dass somit jeglicher Stand der Projektentwicklung jederzeit eingesehen werden kann, sodass die Voraussetzungen für die Wiederholbarkeit und Reproduzierbarkeit des Prozesses gegeben sind (vgl. Popp 2006; Clark 2006).

Obwohl die optimalste Variante vorsieht, dass für jedes Projekt ein eigenes Repository vorhanden ist, kommen daneben andere Vorgehensweisen zum Einsatz. Hierbei handelt es sich um die Nutzung eines einzigen Repository für mehrere Projekte, wobei sich in diesem Fall aber die Notwendigkeit ergibt, dass die Struktur dieses Repository gut organisiert und den jeweiligen Projekten angepasst wird (vgl. Versteegen et al. 2001; Popp 2006).

Externe Bibliotheken werden auf Grund ihrer Größe oftmals nicht als Konfigurationselemente definiert und deshalb außerhalb des Repository archiviert. Zu den auftretenden Verwaltungsproblemen bietet Maven einen Lösungsansatz, indem das Maven-Repository zum Einsatz kommt und die festgelegten externen Bibliotheken aus dem Maven-Repository automatisch lädt und zur Erstellung des Produktes verwendet (Popp 2006).

Die Hersteller von Maven geben einen kurzen Überblick zu den unterschiedlichen Funktionen und Vorteilen, die im Vergleich zu CVS und weiteren Versionskontrollsystemen erfüllt werden. Diesbezüglich wird geschildert, dass Maven vergleichsweise weniger Speicherplatz verbraucht und den Check-Out Prozess des Projekts wesentlich beschleunigt. Daneben ist bei Maven auch keine zusätzliche Versionierung erforderlich, da die Dateinamen mit Leichtigkeit auf die jeweilige Version schließen lassen (maven.apache.org, Repositories).

Neben den genannten Vorteilen, sollte hier auch der wichtigste Nachteil von Mavens Repository-Konzept angesprochen werden. Die Kernfunktionalität von Maven liegt *außen*, d.h. sie wird nicht durch die Installationsdatei, sondern durch die nachträglich implementierten Plugins gewährleistet, die aus dem *Standard-Plugin-Repostiory* geladen werden. Da dieses Repository zu einem unbestimmten Zeitpunkt geändert werden kann, wird somit die Wiederholbarkeit des Build-Prozesses gefährdet. Maven bietet jedoch die Möglichkeit an, das Standard-Repository durch ein eigenes Repository zu ersetzen, womit dieses Problem aus dem Weg geräumt werden kann (Popp 2006).

➢ Check-out und Check-in

Dieses Konzept ermöglicht, dass EntwicklerInnen Dateien nicht im Repository selbst, sondern auf ihrem eigenen Workspace als Arbeitskopie betrachten und bearbeiten können. Hierzu werden Kopien der notwendigen Dateien aus dem Repository geholt und isoliert behandelt. Der Vorgang der Lieferung einer Arbeitskopie aus dem Repository und dessen Speicherung im Workspace wird als Check-out bezeichnet. Der Gegenpart dazu ist das Check-in (auch *Commit* genannt), bei dem die geänderten bzw. erweiterten Dateien wiederum in das Repository eingelesen werden. Während des Check-ins werden die geänderten Dateien einer Überprüfung unterzogen und anschließend eine statische Version des Produkts erzeugt. Bei einer weiteren Behandlung der Dateien muss wiederum der ganze Check-out/Check-in Prozess von Neuem gestartet werden. Wichtig ist dabei, dass die Verzeichnishierarchie des Repository mit dem des Workspace übereinstimmt (vgl. Popp 2006; Collins-Sussman et al. 2008). Die visuelle Darstellung des beschriebenen Check-in/Check-out Verfahrens kann der Abb. 18 entnommen werden.

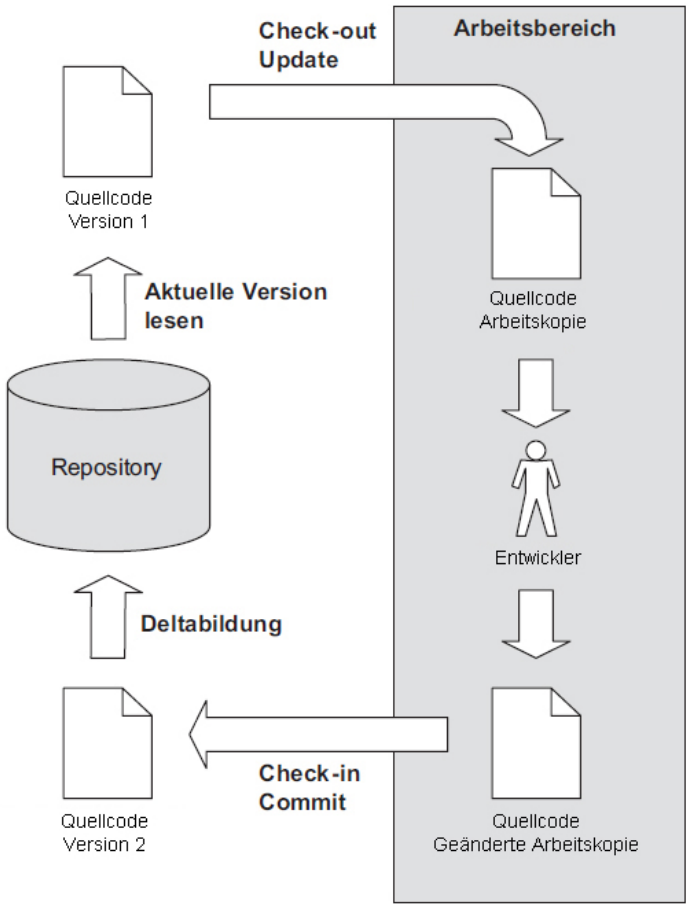


Abbildung 18: Check-out und Check-in einer Datei (aus Popp 2006, S. 38)

Wenn mehrere EntwicklerInnen gleichzeitig dieselbe Datei abrufen und bearbeiten wollen, kann der dabei aufkommende Konflikt durch zwei Modelle gelöst werden (vgl. Popp 2006; Collins-Sussman et al. 2008):

- Lock-Modify-Unlock: Wird der Check-out einer Datei vollzogen, so wird diese Datei bis auf Weiteres für andere Zugriffe gesperrt. Diese Sperre wird erst dann aufgehoben, wenn die Datei mittels Check-in wieder in das Repository eingelesen wird. Der Nachteil, der sich dadurch ergibt ist jener, dass die gleichzeitige Behandlung einer Datei durch mehrere EntwicklerInnen verhindert wird und die entstehenden Wartezeiten zu großem Zeitverlust führen. Der Vorteil dieses Modells ist aber, dass durch diese Vorgehensweise mögliche Konflikte umgangen werden und somit das Risiko einer erforderlichen Konfliktbewältigung verfällt.

- Copy-Modify-Merge: Dieses Modell ermöglicht das gleichzeitige Arbeiten mehrerer EntwicklerInnen an ein und derselben Datei ohne jegliche Sperre. Während des Check-ins schaltet sich ein Informations- bzw. Kontrollmechanismus ein, wodurch der/die jeweilige EntwicklerIn über eventuelle, parallel stattfindende Änderungs- und Erweiterungsversuche an der Datei in Kenntnis gesetzt wird. Anschließend wird die Option angeboten, die unterschiedlichen Änderungen und Weiterentwicklungen automatisch oder manuell zusammenzufügen. Der Prozess der Zusammenfügung wird als *Merge* bezeichnet. Treten nun durch die gleichzeitige Bearbeitung der Datei Konflikte auf, so müssen diese durch bestimmte Lösungsansätze behoben werden. Hält man sich vor Augen, dass dieses Modell die gleichzeitige Behandlung einer Datei ermöglicht, so ergibt sich daraus ein geringerer Zeitverlust als beim Lock-Modify-Unlock Modell. Dieser Ansatz wird speziell in jenen Situationen bevorzugt, bei denen mehrere, räumlich voneinander getrennte EntwicklerInnen an demselben Projekt arbeiten.

➢ Baseline

Baselines (Bezugskonfigurationen) können als Schnappschüsse bezeichnet werden, die zu wichtigen Meilensteinen des Projekts aufgenommen werden. Besonders in Situationen, in denen eine intensive Arbeit mit dem Repository erforderlich ist, kann es notwendig und vorteilhaft sein, den aktuellen Zustand des Repository via Baselines festzuhalten (Popp 2006). Anders ausgedrückt ist eine Baseline die geplante Momentaufnahme eines sehr wichtigen Ereignisses im Repository, also ein „(Zwischen-) Ergebnis des Systems“ (Balzert 1997). Mit Hilfe dieser Bezugskonfigurationen kann die Eigenschaft der Reproduzierbarkeit erfüllt und somit problemlos ein früherer Originalzustand des Repository wiederhergestellt werden.

- **Tags**

Ein Tag (Bezeichner) ist eine Art Querschnitt durch das Repository und hält zu einem bestimmten Zeitpunkt die gültigen Versionen aller Dateien im Repository fest. Die Bezeichnung für einen solchen Tag kann frei gewählt werden, womit es leichter wird, zu einem späteren Zeitpunkt, durch die Angabe des jeweiligen Tag-Namens, den damaligen Zustand des Repository wieder abzurufen. Eine Baseline ist aus technischer Sicht dasselbe wie ein Tag, unterscheidet sich im Praktischen jedoch dadurch, dass sie vergleichsweise eine längere Vorbereitungszeit verlangt und insbesondere zu jenem Zeitpunkt durchgeführt werden sollte, bei dem das System bereits einen gesicherten Status erreicht hat (Popp 2006).

4.2 Kompilierung

In dieser Phase des Build-Prozesses findet die Kompilierung, d.h. die Erzeugung des Produkts aus den Quellenelementen statt. Die Absicht besteht dabei darin, den Quellcode in ausführbaren Maschinencode umzuwandeln, jedoch bedeutet das nicht, dass im Anschluss daran das jeweilige Produkt bereits auslieferfähig ist, da es lediglich in seinen Einzelteilen vorliegt. „Ein Compiler erzeugt aus dem Quellcode nach festen Regeln ausführbaren Code und muss einigen anderen Code aus Bibliotheken hinzufügen oder Schnittstellen zum Betriebssystem herstellen. Er muss also verschiedene Quellteile zusammenfügen“ (Gerlich & Gerlich 2005, S. 60). Einen wichtigen Dienst leistet der Compiler zusätzlich dadurch, dass er den/die BenutzerIn umgehend über Fehler im Quellcode informiert (Aho et al. 1999).

Bei der Übersetzung von Quellcode in ausführbaren Code kommt das Analyse-Synthese-Modell zum Einsatz. Der Analyse-Teil übernimmt dabei die Zerlegung des Quellcodes in seine einzelnen Bestandteile, transformiert diese in eine Zwischensprache und liefert somit eine Zwischendarstellung der abstrakten Baumstruktur des Quellcodes. Zu nennen sind hier drei Teile, aus denen die Analyse besteht: lexikalische-, syntaktische- und semantische Analyse. Bei der lexikalischen Analyse findet die Erkennung (*scanning*) von Grundsymbolen, u.a. Bezeichnern, Wortsymbolen und Literalen, statt. Im Zuge der syntaktischen Analyse (*parsing*) wird die syntaktische Struktur des Quellcodes ermittelt. Hierzu wird der Quellcode zu grammatikalischen Sätzen zusammengefasst, durch die es möglich ist, hierarchische Abhängigkeiten zu erfassen. Während der semantischen Analyse wird sichergestellt, dass alle Bestandteile des Quellcodes sinnvoll zusammenpassen. Dabei wird der Quellcode auf semantische Fehler überprüft und es werden Typ-Informationen gesammelt, die mittels Typenüberprüfungen erhalten werden (vgl. Aho et al. 1999; Kastens 1990; Penner 1994).

Nach Abschluss der drei Analyse-Teile findet die Erzeugung des Zwischencodes statt, sodass über diese Schnittstelle die Überleitung zum Synthese-Teil beginnt. Es wird eine Transformation der Ergebnisse aus dem Analyse-Teil durchgeführt, damit diese in einer für die Synthese geeigneten Form vorliegen. „Die Elemente der Zwischensprache sind Tupel (Operator, Attribute, Operanden), die Zusammen eine Sequenz von Bäumen oder Graphen bilden. Sie können nach verschiedenen Techniken implementiert werden“ (Kastens 1990, S. 32).

Der Synthese-Teil übernimmt sodann die Übersetzung des Zwischencodes in die jeweilige Maschinensprache. Die Synthese lässt sich wiederum in mehrere Teilbereiche gliedern, darunter die Code-Auswahl, die Assemblierung und die Optimierung des Codes. Die Code-Auswahl ist eine besonders wichtige Phase der Synthese, der eine Entwurfsphase der Code-Erzeugung vorangeht. Dabei werden für Operanden (Baumknoten) der Zwischensprache mehrere Code-Sequenzen als mögliche Übersetzungen in die Maschinensprache entworfen. Im Zuge der Code-Auswahl wird schließlich die geeignetste Code-Sequenz bestimmt, mit der jeder einzelne Baumknoten übersetzt werden soll. Als Ergebnis der Code-Auswahl wird somit ein abstrakter Maschinencode, d.h. eine Folge von Maschinenbefehlen als Tupel (Operanden) ausgeliefert. Dieser abstrakte Maschinencode wird anschließend bei der Assemblierung in das Befehlsformat (Bitmuster) der Zielmaschine codiert und als Code-Datei ausgegeben. Diese Datei enthält auch zusätzliche Informationen über das Binden, Referenz-Objekte, Testhilfen und die Struktur der Code-Segmente. Die Assemblierung kann auch mittels Zuhilfenahme von entsprechenden Programmen durchgeführt werden. Beim Optimierungsverfahren geht es darum, die Qualität des erzeugten Maschinencodes zu verbessern, um die Erstellung von praktisch brauchbaren Übersetzern zu fördern. So findet die Betrachtung und Analyse der Baumstruktur der Zwischensprache in einem größeren Kontext statt, sodass diese noch besser transformiert werden kann. Die Optimierung kann dabei auf der Ebene der Zwischensprache oder aber auf jener des erzeugten Maschinencodes (sog. Nachoptimierung) stattfinden (Kastens 1990).

Die Kompilierung ist als eine Phase aufzufassen, in der zur Produkterstellung das Aufrufen des Compilers und des Linkers, sowie weiteren Hilfsprogrammen, notwendig ist. Wie auch der Abb. 19 entnommen werden kann, gehören dazu folgende unterstützende Programme: Präprozessoren, Assembler und Lader oder Binder.

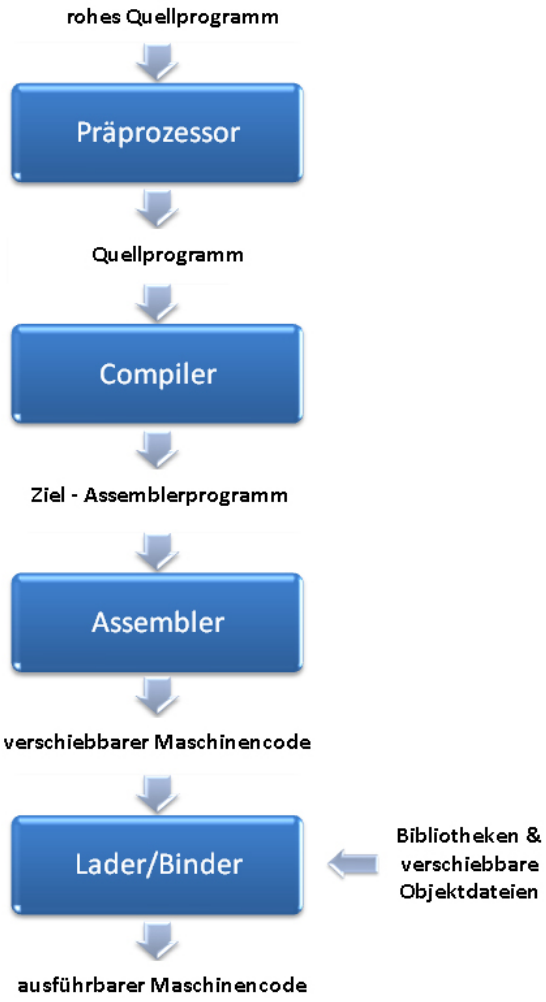


Abbildung 19: Ein sprachverarbeitendes System (nach Aho et al. 1999, S. 5)

Der Präprozessor ist im eigentlichen Sinne ein Programm, welches den Quellcode für den Compiler vorbereitet. Er übernimmt dabei bestimmte Funktionen, wie beispielsweise die Makro-Bearbeitung, die Einkopierung von Dateien und die Spracherweiterung. In der Programmiersprache C findet die Makro-Bearbeitung über Makroprozessoren statt, die denselben Dienst wie Präprozessoren leisten. Der Makroprozessor übernimmt die Aufgabe, eine Zeichenfolge innerhalb des Textes durch eine andere Zeichenfolge zu ersetzen. Bei der Einkopierung werden indes bestimmte Dateien in den Programmtext einkopiert, was im Falle einer C-Datei über den *include*-Befehl abgewickelt wird. Bei der Spracherweiterung können, mittels eingebauten Makros, einer Sprache neue Möglichkeiten hinzugefügt werden. Die Eingabe für den Compiler kann dabei von einem oder mehreren Präprozessor(en) (Aho et al. 1999), oder aber, wie am Beispiel von Java, von gar keinem erzeugt werden. Bei Java werden nämlich die benötigten Informationen vom Compiler selbst über Softwareschnittstellen von Klassen direkt aus den Klassendateien gelesen (Ullenboom 2008).

Bei der Assemblierung findet die Erzeugung von verschiebbarem Maschinencode aus dem Quellcode statt, wobei der vom Assembler ausgegebene Code eine sehr leicht verständliche Version des Maschinencodes darstellt. Im Normalfall erzeugt der Compiler den Assemblercode und gibt diesen an den Assembler zur Verarbeitung weiter. In gewissen Situationen fällt der Assembler jedoch vollständig weg, da manche Compiler alle Aufgaben des Assemblers selbst erledigen und den verschiebbaren Maschinencode direkt an den Binder bzw. Lader übergeben (Aho et al. 1999).

Lader und Binder (od. Bindelader) werden unter dem Begriff Linker zusammengefasst. Für das Laden und Binden wird somit ein einziges Programm ausgeführt, mit dem die einzelnen Programmmodule zu einem ausführbaren Programm zusammengefügt werden. Die Aufgabe des Laders besteht darin, die verschiebbaren Adressen (Befehle und Daten) innerhalb des verschiebbaren Maschinencodes zu verändern und an der richtigen Stelle im Speicher abzulegen. Der Binder nimmt anschließend die Zusammenfügung der verschiedenen Dateien (externe Bibliotheksdateien, verschiebbare Objektdateien, etc.) im verschiebbaren Maschinencode zu einem Programm vor. „Diese Dateien können aus mehreren verschiedenen Übersetzungen stammen oder vom System bereitgestellte Bibliotheksdateien sein. Bibliotheksdateien bieten jeweils eine Sammlung von Routinen an, die allen Programmen zur Verfügung stehen“ (Aho et al. 1999, S. 23).

4.2.1 Programmiersprachen

Es gibt eine Vielzahl an Programmiersprachen, wobei unter diesen die Sprachen C, C++, Java und Microsoft .NET zu den Meistverwendeten zählen. Diese Sprachen verlangen oftmals die Verwendung von bestimmten geeigneten Build-Tools und können in unterschiedlichem Maße auch Einfluss auf den Build-Prozess ausüben, da dieser maßgeblich von der jeweils verwendeten Programmiersprache abhängig ist (Lee 2008).

C ist eine prozedurale Hochsprache, die von Dennis Ritchie in den frühen 70er Jahren in den *Bell Laboratories* für die Plattform Unix entwickelt wurde. Sie wird vermehrt in der systemtechnischen Anwendungsprogrammierung verwendet. C++ stellt eine objektorientierte Erweiterung von C dar und unterstützt daher objektorientierte Entwicklung. C++ enthält konkrete und abstrakte Klassen und ermöglicht die Weitergabe von Mehrfachvererbungen. Auch C++ ist eine Errungenschaft der Bell Laboratories und wurde zu Beginn der 80er Jahre seitens Bjarne Stroustrup entwickelt. Die Veröffentlichung erfolgte schließlich im Jahr 1989. C und C++ gelten als kompilierte Sprachen, die relativ hohe Performanceleistungen erbringen können. Der kompilierte C oder C++ Code arbeitet zwar sehr rasch, aber um von seiner hohen Geschwindigkeits- und Performanceleistung profitieren zu können, ist eine gänzliche Optimierung der Anwendungen notwendig. Dies kann wiederum die erforderliche Kompilierungszeit in deutlichem Maße steigern. Traditionellerweise wird für die Programmiersprachen C und C++ das Build-Tool Make

eingesetzt, zu dem es aber auch verschiedene Alternativen wie Jam, SCons, clearmake und ElectricAccelerator gibt (vgl. Klima & Selberherr 2003; Lee 2008).

Im Jahre 1995 hat die Firma Sun die plattformunabhängige und objektorientierte Programmiersprache Java herausgegeben, deren Syntax jener von C und C++ ähnlich ist. Javas Vorgänger war die Sprache *Oak*, die im Zuge des Green Projekts im Jahr 1991 seitens Bill Joy, James Gosling und Mike Sheridan entwickelt wurde. Java qualifiziert sich durch zahlreiche Einsatzmöglichkeiten in verschiedenen Web-Anwendungen und durch seine vielfältigen Java-Applets, die bei Netzwerk Computern eingesetzt werden können (Zuser et al. 2004). Die zuvor angesprochene Ähnlichkeit zwischen C, C++ und Java besteht darin, dass es sich bei Java auch um eine kompilierte Sprache handelt, die jedoch keinen ausführbaren Code ausgibt. Bei Java wird der Quellcode zuerst in Bytecode kompiliert, der mit Unterstützung einer *Java Virtual Machine* (JVM) ausgeführt wird. JVM kann auf fast allen Betriebssystemen implementiert werden und zeichnet sich deshalb durch seine Plattformunabhängigkeit aus. Trotz dieser Fähigkeit können Java-Anwendungen dennoch nicht mit derselben hohen Performanceleistung ausgeführt werden wie jene von C und C++. Da es sich bei Java um eine Open Source verwandte Anwendung handelt, gibt es eine große Menge an wiederverwendbaren Komponenten-Bibliotheken, Entwicklungs-Frameworks und Server-Plattformen. Auf Grund der Wiederverwendbarkeit dieser Ressourcen verkürzt sich der Java Entwicklungslebenszyklus, sodass Builds rascher ausgeführt werden können, da in diesem Fall weniger Code entwickelt bzw. kompiliert werden muss. Auch wenn die Kompilierung einen relativ kurzen Zeitraum in Anspruch nimmt, kann, durch die Notwendigkeit von bestimmten Konfigurationen für run-time container, die Deployment-Phase und somit die totale Build-Zeit langwierig ausfallen. Das Build-Tool, welches typischerweise für Java-Applikationen eingesetzt wird, ist Apache Ant. Als alternatives Build-Werkzeug zu Ant wird für Java-basierte Applikationen das Tool Apache Maven angeboten (Lee 2008).

Die .NET-Technologie wird von Microsoft vertrieben und ist nicht an eine bestimmte Programmiersprache gebunden. Es ist möglich, eine Vielzahl an Programmiersprachen auf der .NET-Plattform anzuwenden, wobei hierunter C#.C# zu den meistverwendeten Sprachen gehört. Bei .NET begegnen wir der Besonderheit, dass hier Programme nicht in Maschinencode kompiliert, sondern in die Intermediate Language (IL) übersetzt werden. Erst aus der IL wird dann schließlich der eigentliche Code erzeugt. Zur Umsetzung dieses Verfahrens ist deshalb die Installierung von .NET-Runtime erforderlich, das einen ähnlichen Dienst wie die Java Virtual Machine leistet (Zuser et al. 2004). So resultieren für den Build-Prozess mit .NET ähnliche Konsequenzen wie bei Java. Diese können jedoch mehr oder minder abgeschwächt werden, da Microsoft von einer kontrollierteren Entwicklungsumgebung umgeben ist. Das speziell für .NET entwickelte und eingesetzte Tool heißt MS-Build. Ein alternatives Angebot hierzu wurde von der *Apache Foundation* als Open Source Tool unter dem Namen NAnt herausgegeben (Lee 2008).

4.3 Unit Test

Bei der Kompilierung von Quellcode ist es eine voraussetzende Bedingung, dass der Syntax des Codes auf Korrektheit überprüft wird, damit er überhaupt fehlerfrei kompiliert werden kann. Damit wird jedoch nicht garantiert, dass der kompilierte Code den Anforderungen des Projekts entspricht, sodass nur wenig brauchbare Informationen zur Software ausgeliefert werden.

Daher ist es notwendig einen Test durchzuführen, um festzustellen, ob die Software bestimmte Anforderungen erfüllt bzw. ob die Anwendung fehlerfrei ausgeführt werden kann. Um darzustellen, was im Allgemeinen unter dem Testen einer Software verstanden wird, soll auf folgende Definition verwiesen werden (Spillner & Linz 2004, S. 8): „Unter dem Testen von Software wird jede (im Allgemeinen stichprobenartige) Ausführung eines Testobjekts verstanden, die der Überprüfung des Testobjekts dient. Die Randbedingungen für die Ausführung des Tests müssen festgelegt sein. Der Vergleich zwischen Soll- und Ist-Verhalten des Testobjekts dient zur Bestimmung, ob das Testobjekt die geforderten Eigenschaften erfüllt."

Nach ANSI/IEEE Standard 610.12-1990 ist Test: „[…] the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component."

Diese Definition zeigt, dass Tests nicht unbedingt für das vollständige System zugeschnitten sein müssen, sondern auch Teilkomponenten einer unfertigen Software getestet werden können. Dieser Aspekt ist vor allem deshalb wichtig, da innerhalb des Build-Prozesses nach der Kompilierung kein fertiges Endprodukt, sondern lediglich Komponenten vorliegen, die einen geeigneten Komponententest, auch Modul- oder Unit-Test genannt, erfordern.

Ein Unit-Test wird seitens EntwicklerInnen in einer Laborumgebung ausgeführt und dient zur Testung des kleinsten spezifizierten oder nicht spezifizierten Moduls in einem System (Koomen & Pol 1999). Es handelt sich um einen technisch orientierten Test, der auf Basis von *in* und *out* Parametern arbeitet (Runeson 2006). Bei der zu testenden Unit kann es sich um eine „isolierte Klasse, eine durch Assoziationen verbundene Kette von benachbarten Klassen" (Vigenschow 2005, S. 22) und einzelnen oder mehreren Komponenten einer Software (Whittaker 2000) handeln.

Da es beim Unit-Test vorrangig darum geht, das Design und die Funktionalität des Quellcodes zu prüfen, kommen insbesondere zwei Verfahren in Frage: das *White-Box* und das *Grey-Box Testing*. Beim White-Box Test wird die innere Struktur eines Testobjekts auf Vollständigkeit und Korrektheit geprüft, d.h. der Programmcode analysiert und festgestellt, ob dieser richtig funktioniert. Eine weitere Bezeichnung für den White-Box Test lautet

Strukturtest (Franz 2008). Der Grey-Box Test ist eine Mischung aus dem White-Box und dem *Black-Box Test*, der die Funktionalitäten beider Tests in sich vereint. Beim Black-Box Test wird überprüft, ob die Software den gewünschten Anforderungen entspricht, d.h. ob alles richtig programmiert wurde. Somit treffen beim Grey-Box Test sowohl die Überprüfung der Anforderungen, als auch der inneren Strukturen zusammen, weshalb diese Art des Tests gerne bei komponentenbasierten Strukturen zum Einsatz kommt (Vigenschow 2005).

4.3.1 Unit-Test Ebenen

Der Unit-Test findet auf drei verschiedenen Ebenen statt, die in Abb. 20 übersichtlich dargestellt sind. Diese Ebenen werden als Klassentest, Ketten-/Klassenintegrationstest und Komponententest bezeichnet. Im Zuge des Build-Prozesses wird verlangt, dass auf all diesen Ebenen Unit-Tests durchgeführt werden. An diesem Punkt ist es wichtig zu klären, welche AkteurInnen die Ausführung der jeweiligen Tests übernehmen. Das ausschlaggebende dabei ist, dass für EntwicklerInnen die kleinste Software-Einheit als Klasse, für TesterInnen hingegen als Komponente definiert wird (Vigenschow 2005).

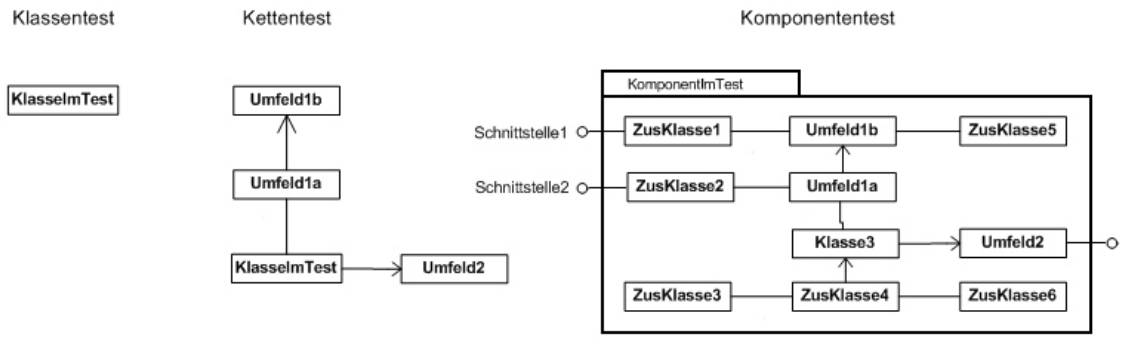


Abbildung 20: Klassen-, Ketten- und Komponenten-Test (aus Vigenschow 2005, S. 23)

Aus diesem Grund fällt die Durchführung der Klassentests in den Aufgabenbereich der EntwicklerInnen, was aber nicht bedeuten muss, dass diese mit der aufwändigen Erstellung aller Testfälle betraut sind. Folglich kann eine Entlastung der EntwicklerInnen erreicht werden, wenn Testfälle für den Klassentest vom Testteam zur Verfügung gestellt werden oder ein geeignetes Testframework zur Durchführung und Dokumentation der Klassentests eingesetzt wird (Franz 2008). An dieser Stelle ist explizit darauf hinzuweisen, dass nicht nur die Funktionalität isolierter Klassen getestet werden sollte, sondern auch genügend Zeit für einen sog. Ketten- bzw. Klassenintegrationstest einzuräumen ist.

Obwohl dem Kettentest im Vergleich zum Klassentest ein höherer Stellenwert beizumessen ist, kommt es dennoch vor, dass bei Zeitmangel gänzlich auf ihn verzichtet wird. Diese Vorgehensweise ist vor allem deshalb nicht ratsam, da erst durch den Klassenintegrationstest sichergestellt wird, ob/dass die Kommunikation und das Zusammenspiel benachbarter Klassen entlang der Assoziationsketten fehlerfrei funktioniert (Vigenschow 2005). Wichtig ist in jedem Fall, dass EntwicklerInnen die Ergebnisse des Klassen- bzw. Klassenintegrationstests sorgfältig dokumentieren, sodass dieser gegebenenfalls auch wiederholbar ist, da diese Testnachweise für das Testteam eine unverzichtbare Voraussetzung zur Durchführung des Komponententests sind (Franz 2008).

Der meiste Aufwand ist letztlich beim Komponententest (Modultest) zu erwarten, da sich eine einzelne Komponente aus mehreren Klassen und/oder Klassenketten zusammensetzen kann und sich die Komplexität des Tests somit erheblich steigert. Da innerhalb der zu testenden Komponente mehrere Schnittstellen und Verzweigungen zwischen den Klassen bzw. Klassenketten vorhanden sein können, gilt es deren Beziehungen, Kommunikation und Interaktion auf Fehleranfälligkeit zu testen, was verständlicherweise einen größeren Aufwand an Zeit bedeutet (Vigenschow 2005).

4.3.2 Mock-Objekte

Nach der Definition von Koomen und Pol findet der Unit-Test bekanntlich in einer Laborumgebung statt, womit im Speziellen darauf hingewiesen wird, dass dieser lokal und isoliert von Fremdsystemen, wie Netzwerken, Datenbanken oder noch nicht funktionsfähigen Teilen, durchgeführt wird (1999). Wenn das Testobjekt aber von Fremdsystemen oder nicht funktionsfähigen Teilen abhängig ist, müssen diese mittels Simulation in den Test miteinbezogen werden. Hier kommen nun Stellvertreterobjekte, die sog. Mock-Objekte, zum Einsatz. „Ein Mock-Objekt ersetzt während des Testens das entsprechende reale Objekt“ (Hunt & Thomas 2004, S. 75).

Insgesamt gibt es drei verschiedene Arten von Stellvertreterobjekten: *Stub*, *Dummy* und *Mock*s. Ein Stub ist ein einfacher Platzhalter, der dazu dient, den Platz einer geplanten, aber noch nicht umgesetzten Funktionalität einzunehmen. Im Vergleich dazu ist Dummy ein Objekt, welches eine echte Implementierung für Testzwecke möglichst einfach und oft stark eingeschränkt simulieren kann. Mock-Objekte hingegen verfügen über zusätzliche Funktionalität und sind die intelligentesten Stellvertreterobjekte dieser Reihe. Dadurch dass sie testspezifischen Code enthalten, ist es möglich, das Testobjekt von innen zu testen (Link et al. 2005).

Um ein Mock-Objekt erstellen zu können, muss zuerst ein Interface angelegt werden (siehe Abb. 21), über welches das Verhalten der zu simulierenden Fremdkomponente festgelegt wird. Dieses Interface wird dann sowohl von der fremden Komponente, als auch vom

Mock-Objekt implementiert, welches durch diese Implementierung das Verhalten der Fremdkomponente annimmt (Hüttermann 2008).

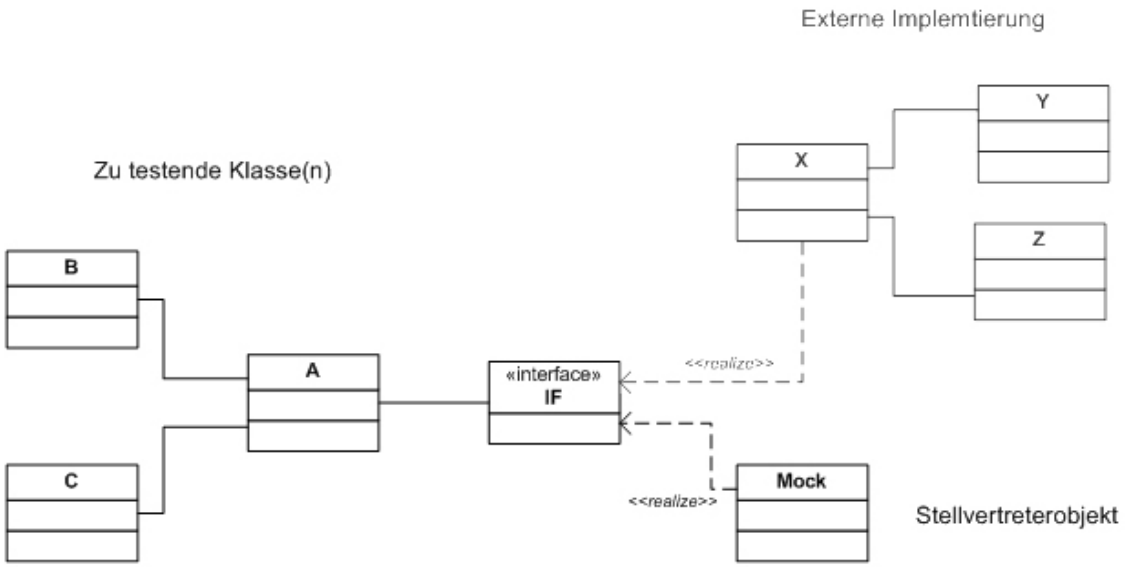


Abbildung 21: Realisierung des Stellvertreterobjekts (aus Vigenschow 2005, S. 181)

Durch den Einsatz von Mock-Objekten wird gewährleistet, dass der Unit-Test so weit wie möglich lokal und isoliert durchgeführt werden kann. Daneben entsteht der Vorteil, dass Unit-Tests auf strukturierte und rasche Art und Weise durchgeführt werden können und auch Rückkopplungen jederzeit möglich sind. Ist eine Komponente von einer anderen, nicht existenten Komponente abhängig, so kann mittels Simulation dennoch mit der Entwicklung und dem Test fortgefahren werden (Hüttermann 2008). Mit Hilfe von Mock-Objekten ist es auch einfacher eventuelle Grenzfälle zu simulieren (Vigenschow 2005).

4.3.3 Automatisierung des Unit-Tests

Als integrativer Bestandteil des Build-Prozesses ist bei einem automatisierten Unit-Test besonders zu beachten, dass zu keinem Zeitpunkt der Testphase ein manueller Eingriff vorgenommen werden darf. Die Automatisierung des Unit-Tests wird primär durch das Vorhandensein und dem Einsatz von Mock-Objekten unterstützt, wobei durch sie automatisierte Tests schon frühzeitig im Entwicklungszyklus eingesetzt und auftretende Fehler besser eingegrenzt werden können.

Daher ist es auch gut möglich, den Unit-Test als begleitenden Test auszuführen, wodurch die Produktivität der Testphase auf einem relativ hohen und konstanten Niveau gehalten werden kann (Hunt & Thomas 2004). Dieser konstante Produktivitätszustand beim begleitenden Testen ist im linken Teil der Abb. 22 ersichtlich, wobei rechts die Auswirkungen einer einzigen Testphase auf die Produktivität am Verlauf des Graphen abgelesen werden kann.

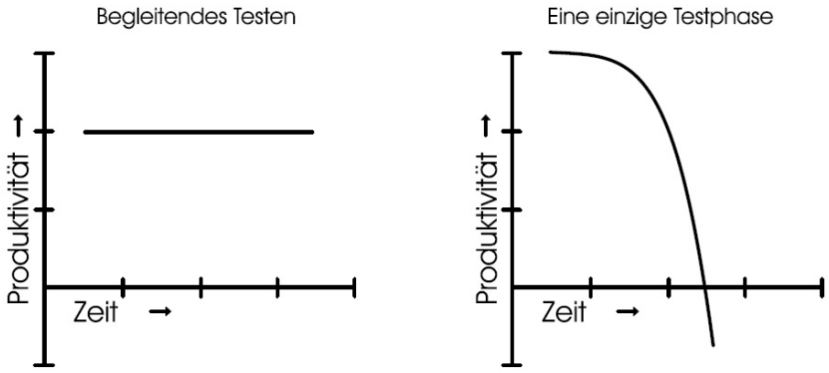


Abbildung 22: Begleitendes Testen (aus Hunt & Thomas 2004, S. 10)

Als Bestandteil des automatisierten Build-Prozess bietet der Unit-Test außerdem die Möglichkeit, das Design und die Architektur der Software schon ab einem frühen Zeitpunkt fortlaufend zu verbessern (Hunt & Thomas 2004). Zusätzlich kann der Unit-Test auch als Performancetest zum Einsatz kommen, sodass man zu einem bestimmten Zeitpunkt eine Rückmeldung, beispielsweise zum Zustand der Datenbank, bekommt (Hüttermann 2008).

4.4 Deployment

Lee vertritt die Ansicht, dass der Build-Prozess die grundlegende Voraussetzung für die Umsetzung eines kompletten Lebenszyklus-Managements von Anwendungen (*Application Lifecycle Management*) ist. Er bevorzugt es, den Build-Prozess als eigentlichen Faktor anzusehen, der es überhaupt erst möglich macht, dass eine Software ausgeliefert werden kann (2008).

Software Deployment ist eine Post-Produktionsaktivität, die zur Bereitstellung eines Softwarestücks für oder von KundInnen [4] dient. Software Deployment kann als Prozess definiert werden, der aus einer Reihe von Aktivitäten besteht und zwischen der Erfassung

[4] Darunter können sich sowohl EndkundInnen als auch jegliche AkteurInnen innerhalb eines Software-Lebenszyklus befinden.

(*Acquisition*) und der Ausführung (*Execution*) einer Software stattfindet. Zu diesen Aktivitäten gehören die Erstellung des Installationspakets am Ende des Entwicklungszyklus, die Konfiguration, die Installation auf der Zielplattform und die endgültige Aktivierung der Software. Hinzu kommen noch Aktivitäten, die nach der Installation stattfinden, wie z. B.: Beobachtung, Deaktivierung, Update, Re-Konfiguration, Adaptation, *Redeploying* und *Undeploying* der Software (Dearle 2007). Abhängig vom jeweiligen Softwaresystem kann die Vorgehensweise beim Deployment zwar variieren und daher nicht einheitlich präzise beschrieben werden, aber es ist dennoch möglich die wichtigsten Kernbereiche des Prozesses generell festzuhalten.

4.4.1 Installationspaket

Ein Installationspaket besteht aus einer Bündelung von Build Outputs, welche die Bedingungen der Vollständigkeit (Lee 2008), Nachvollziehbarkeit und Installierbarkeit (Steg 2004) erfüllen müssen, damit das Produkt korrekt ausgeliefert werden kann.

Abb. 23 zeigt, aus welchen Teilen sich ein solches Installationspaket (sog. *Deployment Unit* oder *Distributionsdatei*) zusammensetzt und gibt Aufschluss zu den internen hierarchischen Beziehungen.

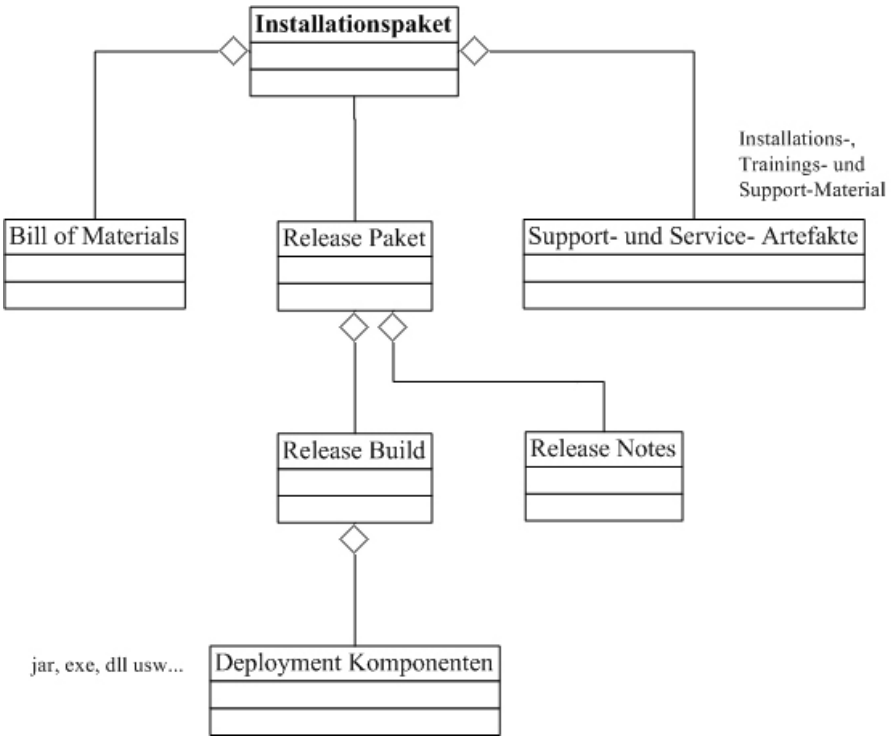


Abbildung 23: Installationspaket (nach Lee 2008, S. 116)

Ein *Installationspaket* besteht aus einer Reihe von Artefakten, die sich auf verschiedenen Ebenen befinden. Auf der untersten Ebene befinden sich die *Deployment Komponenten*, welche nach der Ausführung des Kompilierungsprozesses produziert werden. Deployment Komponenten sind für gewöhnlich die Build-Bibliotheken (z. B.: .jar, .dll Dateien) oder aber ausführbare Dateien. Eine Ebene höher steht das *Release Build*, welches im Allgemeinen aus der Sammlung aller bestehenden Deployment Komponenten hervorgeht. Auf derselben Ebene befinden sich die *Release Notes*, welche gemeinsam mit dem Release Build in einem *Release Paket* gebündelt werden können, damit ein einfacherer Installations- und Deployment-Prozess gewährleistet werden kann. Wie die genaue Ausprägung des Release Pakets aussieht, ist vorrangig von der Art des gewünschten Produkts und der Zielumgebung abhängig. Zu beachten ist, dass das Release Paket auf jeden Fall alles Notwendige enthalten sollte, damit es erfolgreich eingesetzt werden kann. Gemeinsam mit dem Release Paket befinden sich auf dieser Ebene auch die *Bill of Materials* und *Support- und Service-Artefakte* (Anwenderdokumentation). Die Anwenderdokumentation setzt sich aus Installationsanleitungen, Schulungs- und Unterstützungsmaterialien zusammen. Auf oberster Ebene begegnet uns das Installationspaket selbst, welches alle bisher genannten unteren Instanzen in sich aufnimmt und zu einem Ganzen vereint (Lee 2008).

Bei den *Bill of Materials* (sog. Stückliste) handelt es sich um eine exakte Auflistung aller Bestandteile, die zur Erstellung eines Produkts benötigt werden. Eine Liste der verwendeten Materialien und Komponenten sollte stets angefertigt und dem Installationspaket hinzugefügt werden (Wikipedia, Bill of Materials).

Um nun ein Installationspaket zusammenstellen zu können, ist es vorerst notwendig die Installationsparameter hinzuzufügen. Durch diese Parameter werden Anweisungen zum Bau eines Installationspakets angegeben, die mit Hilfe von Installationsskripten ausgeführt werden können, wobei sich diese Anweisungen entweder im Release-Skript befinden können oder aber ein separates Skript zur Auslieferung des Installationspakets erstellt werden kann (Clark 2006). Anschließend wird eine *Baseline* erstellt, welche die Installationsskripte zusammen mit den ausführbaren Programmen festhält, die als *Golden Copy/2* bezeichnet wird. Die einzelnen Installationspakete werden dann in das Repository übernommen und in einer weiteren Baseline (*Silver Copy*) festgehalten, die als Schnittstelle zur Verteilung des Softwarepakets dient (Steg 2004). In Abb. 24 werden diese Schritte zur Erstellung des Installationspakets zusammengefasst.

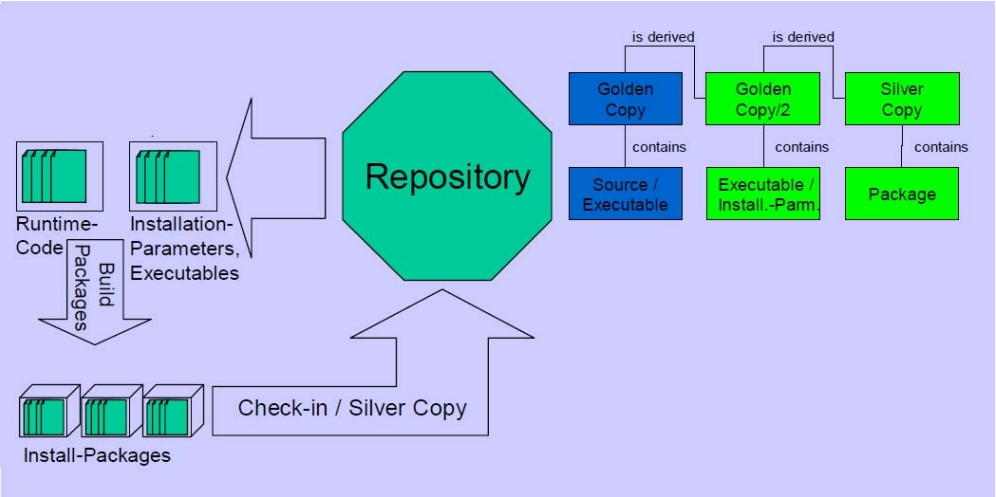


Abbildung 24: Erstellen des Installationspakets (nach Steg 2004, S. 28)

- **Konfiguration**

Die Konfiguration der Software ist ein wichtiger Aspekt innerhalb des gesamten Deployment Prozesses, da sie gewährleistet, dass die Anwendung auf verschiedenen Zielumgebungen ausgeführt werden kann. Konfiguration bedeutet insofern, dass Änderungen an den Parametern zur funktionsfähigen Ausführung der Software in der jeweiligen Zielumgebung durchgeführt werden. Die genannten Parameter befinden sich für gewöhnlich in einer eigenen Property- oder Konfigurationsdatei innerhalb des Installationspakets. Bei der Implementierung der Anwendung sollte daher darauf geachtet werden, dass diese konfigurierbar ist. Somit wird es möglich, dass die Parameter in einer Datei verändert werden können, während die Anwendung diesen Vorgang beobachtet und sich bei einer allfälligen Änderung automatisch re-initialisiert. Wichtig ist in jeglicher Situation, sich im Vorhinein Gedanken zur Struktur der Anwendung zu machen und eine Strategie zu bestimmen, wie Konfigurationen zu handhaben sind und schnellstmöglich implementiert werden können (Lee 2008).

4.4.2 Installation

Die Installation erfordert die Übertragung der Software zum Kunden/zur Kundin und die vorbereitende Konfiguration für die Aktivierung der Software (Dearle 2007). Es gibt zwei Mechanismen, bezeichnet als *Push* und *Pull*, die zum Transfer der Anwendung in die Zielumgebung zum Einsatz kommen. Beim Push-Modell übernimmt das Source Control System den aktiven Part und schiebt das Anwendungspaket auf das Zielsystem. Das Pull-Modell arbeitet in gegensätzlicher Funktionsweise, indem sich das Zielsystem aktiv um die Übertragung des Installationspakets kümmert, dieses anfordert und vom Source Control System auf das lokale System kopiert (Mücke & Ochotta 2006). Das Pull-Verfahren findet

meist auf Benutzerseite Anwendung, indem die Installation von den BenutzerInnen selbst ausgeführt wird, beispielsweise über eine CD oder über den Download von der Anbieter-Webseite. Der Push Mechanismus kommt indes generell bei Installationen über das Netzwerk zum Einsatz.

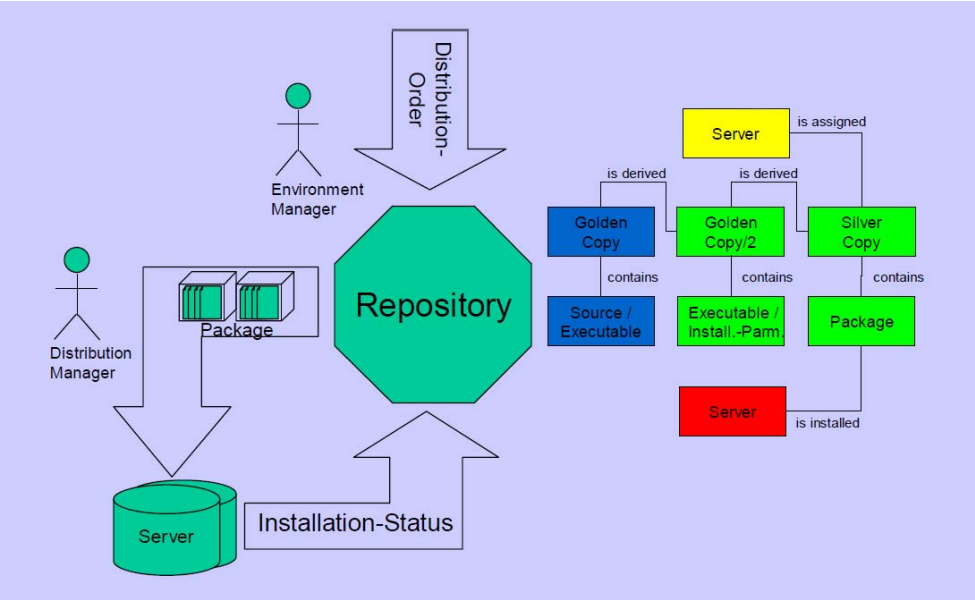


Abbildung 25: Installation (aus Steg 2004, S. 29)

In Abb. 25 ist ein Installations- und Verteilungsprozess mitsamt Angabe der beteiligten AkteurInnen dargestellt. Die Anweisung zur Verteilung des Softwareprodukts erteilt der/die Environment-ManagerIn und bestimmt dabei in enger Zusammenarbeit mit dem/der EntwicklerIn einen bzw. mehrere Ziel-Server, an die das Installationspaket verteilt werden soll. Anschließend kommt der/die Distributions-ManagerIn ins Spiel und führt die Installation auf den vorbestimmten Ziel-Servern gemäß den Installationsanweisungen aus. Zuletzt sollte der Status der Installation festgehalten und im Repository gespeichert werden, sodass alle EntwicklerInnen diesen jederzeit einsehen können. In manchen Fällen kann der Posten des/der Environment-ManagerIn und des/der Distribution-ManagerIn in einer Person vereint werden, da deren Aufgabenbereiche eng miteinander in Verbindung stehen (Steg 2004).

➢ Post-Installationsaktivitäten

Nach der Aktivierung der Software, d.h. dass die Ausführung der Anwendung gestartet oder ein genauer Zeitpunkt zur Ausführung bestimmt worden ist, können eine Reihe von weiteren Aktivitäten durchgeführt werden. Dazu zählt die Deaktivierung, die zumeist für die Adaption oder Re-Konfiguration der Software notwendig sein kann, damit bestimmte

Teile der Software außer Betrieb gesetzt werden können. Eine weitere Aktivität ist das Update, bei dem Teile der installierten Software, meist bedingt durch die Veröffentlichung einer neuen Version, geändert werden müssen. Das Update kann als eine Form der Installation angesehen werden, bei der jedoch die vorherige Deaktivierung der installierten Software notwendig sein kann. Nach abgeschlossener Re-Konfiguration kann die Software wieder aktiviert werden. Die Adaptation ist eine Aktivität, bei der die installierte Software so verändert wird, dass sie an Änderungen in der Systemumgebung angepasst wird. Zuletzt ist die Aktivität zu erwähnen, bei der die Software vom Rechner, auf dem sie installiert war, entfernt wird. Dieser Vorgang wird als Undeployment bezeichnet, besser bekannt unter dem Begriff Deinstallation (Dearle 2007).

4.4.3 Automatisierung des Deployment Prozesses

Da sich der Deployment Prozess auf mehreren Ebenen abspielen kann, - auf der HerstellerInnenseite (*Producer Deployment*), Unternehmensseite (*Enterprise Deployment*) und BenutzerInnenseite (*User Deployment*) - fallen zahlreiche Aktivitäten an, die sich auf viele unterschiedliche AkteurInnen beziehen (Mücke & Ochotta 2006). Trotz der großen Komplexität des gesamten Deployment Prozesses, ist es wichtig, dass dennoch dessen Wiederholbarkeit gewährleistet wird. An diesem Punkt wäre es sinnvoll eine Automatisierung der Deployment Prozesse, die zuvor standardisiert wurden, durchzuführen. Beispielsweise kann mittels Standardisierung der Installationsanweisungen der Bau des Installationspakets automatisiert werden. Durch Änderung der Installationsparameter kann man unterschiedliche Applikationen, die auf dieselbe Art installiert werden können, denselben Prozess durchlaufen lassen. “Wichtig ist, dass Parameter-Änderungen für den automatischen Paket-Bau ebenso versioniert werden wie die Softwareprodukte“ (Steg 2004, S. 27). Wenn zu regelmäßigen, nahe beieinanderliegenden Zeitpunkten Releases erscheinen, ist es nicht sinnvoll den Prozess jedes Mal manuell auszuführen.

So ist auch das Push-Modell ein Beispiel für einen automatisierten Vorgang, bei dem insbesondere die aktive Kundenbeteiligung am Installationsprozess minimiert werden kann. Der Push-Mechanismus bietet den Vorteil, dass BenutzerInnen keine manuelle Installation durchführen und sich nicht darum kümmern müssen, die passende Version für ihre Plattform zu finden, da die Software unabhängig von Plattformgrenzen automatisch verteilt werden kann. Hinzu kommt die bequeme Funktion, dass in regelmäßigen Abständen Upgrades automatisch gesucht und installiert werden.

4.5 Dokumentation

Die in Kapitel 2.3.3 angesprochene Eigenschaft des automatisierten Build-Prozesses, dass Builds unter anderem informativ sein sollten, wird erst durch den Dokumentationsprozess erfüllt. Die Dokumentationsphase findet nicht erst am Ende des Build-Prozesses statt, sondern ist eine Aktivität, die parallel zu den einzelnen Schritten des Build-Lebenszyklus abläuft.

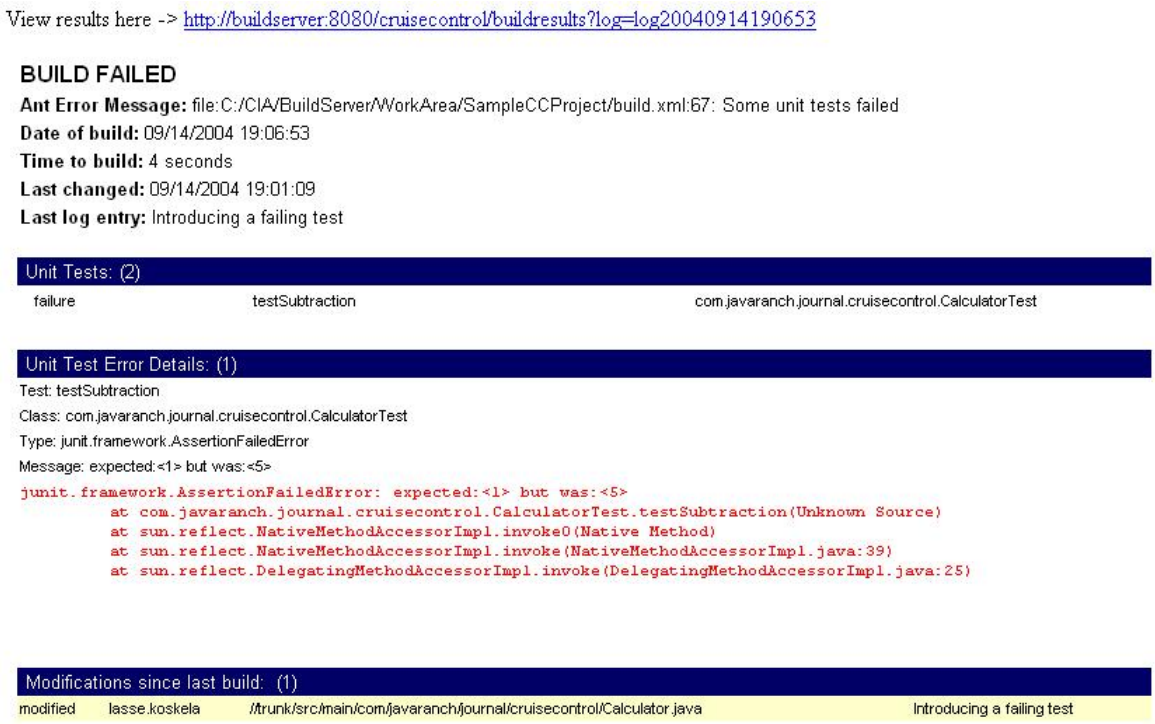
Die resultierenden Dokumentationsergebnisse zu den einzelnen Schritten werden in einer *Log-Datei* gespeichert, für die im Repository wiederum ein zugehöriger Tag angelegt werden kann. Die kontinuierliche Benachrichtigung bzw. gesammelte Veröffentlichung in Form einer Protokolldatei kann via E-Mail, SMS, eXtreme Geräte oder über eine Projekthomepage erfolgen (Popp 2006).

Die Dokumentation genießt einen wichtigen Stellenwertwert als Teil des Build-Prozesses, da dieser uns über den aktuellen Gesundheitszustand des Produkts zu unterrichten vermag. Somit können potenzielle Fehlerquellen bzw. aufgetretene Defekte und Probleme sehr rasch identifiziert und dokumentiert werden. ProjektmitarbeiterInnen werden umgehend und frühzeitig auf Defizite und Fehler am Produkt aufmerksam gemacht, da durch die Dokumentation die Ursachen von Fehlern leichter aufzudecken sind, und relativ einfach festgestellt werden kann, ob es eventuell fehlende Dateistücke gibt (Clark 2006). Der Nutzen, der auf der Ebene der Projektteammitglieder entsteht ist jener, dass mit Hilfe der Dokumentation die Kommunikation zwischen diesen, über geografische bzw. räumliche Distanzen hinweg, hergestellt und aufrechterhalten werden kann. Damit wird sichergestellt, dass unabhängig von einem bestimmten Standort, jedes Teammitglied jederzeit über den aktuellen Projektstand informiert ist und Einsicht in die ausgegebene Dokumentation nehmen kann, sodass der Informationsfluss während des gesamten Build-Prozesses nicht unterbrochen wird.

4.5.1 Veröffentlichung der Dokumentation

Ein Dokumentationsprozess hält den Build-Prozess zwar Schritt für Schritt fest und gibt alle entdeckten Fehler bekannt, er ist für sich selbst aber nicht analysefähig und wird erst durch das hinzukommen des menschlichen Faktors aussagekräftig. Diesbezüglich spielen vor allem EntwicklerInnen eine bedeutende Rolle in der Analyse des ausgegebenen Dokumentationsberichts, sowie bei anschließenden Maßnahmen zur Behebung von Fehlern. Daher ist die Veröffentlichung der Dokumentationsergebnisse für mitwirkende Teammitglieder eine äußerst wichtige und empfehlenswerte Vorgehensweise, die nicht vernachlässigt werden sollte. Besonders bei größeren Projekten, die auf Grund von möglichen räumlichen Distanzen zwischen den ProjektmitarbeiterInnen eine übersichtliche Organisation verlangen, ist es vorauszusetzen, dass die Informationserlangung und der Informationsaustausch zum Produkt einwandfrei funktionieren. Die Veröffentlichung kann

indes über verschiedene Informations- und Kommunikationskanäle erfolgen. Unter den meist verwendeten Möglichkeiten, die auch gleichzeitig zum Einsatz kommen können, befinden sich E-Mails, Projekthomepages, RSS, SMS, eXtreme Geräte und sogar eine experimentelle Herangehensweise unter Verwendung einer Lawa-Lampe [5].


View results here -> http://buildserver:8080/cruisecontrol/buildresults?log=log20040914190653

BUILD FAILED

Ant Error Message: file:C:/CIA/BuildServer/WorkArea/SampleCCProject/build.xml:67: Some unit tests failed
Date of build: 09/14/2004 19:06:53
Time to build: 4 seconds
Last changed: 09/14/2004 19:01:09
Last log entry: Introducing a failing test

Unit Tests: (2)

failure	testSubtraction	com.javaranch.journal.cruisecontrol.CalculatorTest

Unit Test Error Details: (1)

Test: testSubtraction
Class: com.javaranch.journal.cruisecontrol.CalculatorTest
Type: junit.framework.AssertionFailedError
Message: expected:<1> but was:<5>

```
junit.framework.AssertionFailedError: expected:<1> but was:<5>
        at com.javaranch.journal.cruisecontrol.CalculatorTest.testSubtraction(Unknown Source)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

Modifications since last build: (1)

modified	lasse.koskela	/trunk/src/main/com/javaranch/journal/cruisecontrol/Calculator.java	Introducing a failing test

Abbildung 26: E-Mail bei einem Build-Fehler (aus Lasse 2004)

Die Übermittlung von Build-Ergebnissen per E-Mail sollte einfachen Prinzipien folgen, wie etwa Benachrichtigungen im Falle eines fehlgeschlagenen Builds oder, wenn dieser wieder funktionsfähig ist. Es gibt aber auch eine Vielzahl anderer Möglichkeiten, die Benachrichtigungsweise und den Empfänger zu bestimmen (Clark 2006). Die Abb. 26 zeigt ein Beispiel für eine E-Mail, die den Bericht zu einem Build-Fehler enthält.

Auch die Benachrichtigung via SMS ist eine bequeme und einfache Form, um direkt über Build-Ergebnisse informiert zu werden. Gegenwärtig bieten zahlreiche Mobilfunkanbieter E-Mail Adressen an, über die das Senden bzw. Empfangen von E-Mails als SMS möglich ist. Die Nachrichten können wahlweise auch an einen Pager verschickt werden. Eine weitere Möglichkeit ist die Übertragung von Informationen via RSS-Feeds direkt auf den eigenen Desktop. Dafür ist lediglich ein RSS-Programm notwendig, dem mitgeteilt werden muss, wo die erforderlichen Dokumentationsdateien auffindbar sind. Der Rest geschieht

[5] Sog. „Bubble, Bubble, Build‘s in trouble“ - Versuch, bei dem zwei Lawa-Lampen (eine mit roten, eine mit grünen Bläschen) als Indikatoren für funktionierende (grüne Lampe leuchtet) oder fehlgeschlagene (rote Lampe leuchtet) Builds dienen. (pragmaticautomation.com)

automatisch, indem bei jeder Aktualisierung an der Datei eine Benachrichtigung über das Vorhandensein neuer Informationen aufscheint. RSS-Feeds sind XML-Dateien auf einem Server. Daher ist es möglich, dass XML-Build-Protokolle mittels XSLT in ein RSS-Build-Feed transformiert werden. Die meisten eXtremen Geräte bieten eine Benachrichtigungsform, die nur eine bestimmte Anzahl von MitarbeiterInnen in einem gemeinsamen Arbeitsraum erreichen kann. Als Beispiel dazu wäre ein Überwachungsmonitor zu nennen, der den aktuellen Zustand des Dokumentationsprozesses anzeigt, sodass die Teammitglieder jederzeit die Angaben am Monitor ablesen können (Clark 2006). Einige Build-Tools, darunter Ant und Maven, unterstützen die automatisierte Dokumentation und Veröffentlichung der Build-Resultate und bieten auch verschiedenste Konfigurationsmöglichkeiten an.

Eine der meistverwendeten Informations- und Benachrichtigungsform neben den E-Mails ist die Veröffentlichung der Build-Rückmeldungen auf einer Projektwebseite. Ein großer Vorteil dieser Vorgehensweise ist, dass historische Aufzeichnungen zu allen stattgefundenen Builds an ein und derselben Quelle aufgefunden werden können. In Abb. 27 ist eine beispielhafte Projektwebseite mit der Build-Geschichte (sog. *Build-History*) zu sehen, wobei im Hauptfenster das Protokoll zum aktuellen Build angezeigt und in der linken Leiste die Build-History der zuletzt durchgeführten Builds aufgelistet wird, die durch einen Mausklick in detaillierter Form im Hauptfenster eingesehen werden können.

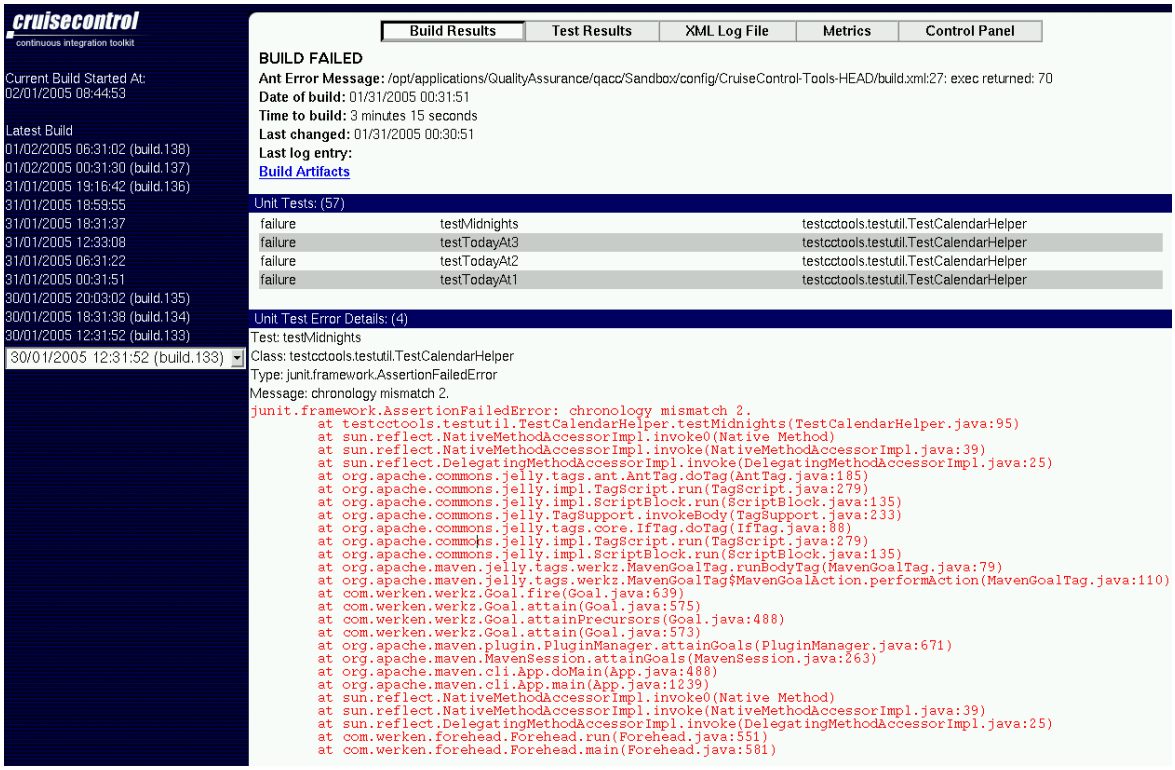


Abbildung 27: Web-Seite mit Build-Geschichte (aus Meyer 2005)

5. Build-Varianten

In diesem Kapitel werden die verschiedenen Varianten des Build-Prozesses aufgegriffen, die den speziellen Anforderungen einzelner Projektphasen angepasst sind. Insofern werden die Entwickler-Builds (*Private System Builds*), die Integrations-Builds und die Produktions-Builds (*Release-Builds*) anzusprechen sein. Die genannten Build-Varianten unterscheiden sich dadurch, dass sie zur Umsetzung unterschiedlicher Anforderungen in unterschiedlicher Häufigkeit zum Einsatz kommen (vgl. Popp 2006; Berczuk et al. 2003). Der Build-Rhythmus, d.h. die Häufigkeit mit der Builds ausgeführt werden, ist bereits in Kapitel 2.6 behandelt worden, sodass in den nachstehenden Kapiteln Aufschluss zu den Unterschieden, der Beschaffenheit und den Anforderungen der verschiedenen Build-Varianten gegeben werden soll.

5.1 Entwickler-Build

Der Entwickler-Build (auch *Private System Build* genannt) wird speziell von den einzelnen EntwicklerInnen auf ihrer isolierten und lokalen Arbeitsumgebung (*Private Workspace*) ausgeführt, um ihren individuellen Arbeitsfortschritt bzw. Änderungen am Quellcode auf Konsistenz und Korrektheit zu überprüfen. Mit der Ausführung des Entwickler-Builds soll sichergestellt werden, dass die am Quellcode getätigten Änderungen nicht zum Zusammenbruch des Systems führen. Zu diesem Zweck ist es notwendig vor dem Check-in der Änderungen in das Repository, d.h. bevor diese für den Rest des Entwicklungsteams sichtbar werden, deren Kompatibilität mit der aktuellen stabilen Version des ausführbaren Quellcodes zu überprüfen (Berczuk et al. 2003).

Bevor der Entwickler-Build ausgeführt wird, müssen einige vorbereitende Maßnahmen getroffen werden. Nachdem der/die EntwicklerIn eine Arbeitskopie des erforderlichen Quellcodes per Check-out aus dem Repository abgerufen und die beabsichtigten Änderungen bzw. Erweiterung durchgeführt hat, muss er/sie ein Update des Private Workspace durchführen. Hierbei wird die jeweilige lokale Arbeitsumgebung auf den neuesten Stand gebracht, indem die aktuellsten stabilen Versionen des Quellcodes, die von anderen EntwicklerInnen in das Repository eingelesen wurden, hinzugefügt werden. Berczuk und Appleton empfehlen zudem, dass vor dem Update des Private Workspace, eine *Private Version* des geänderten Quellcodes angelegt werden sollte, damit nach der Aktualisierung bei Bedarf der vorherige Zustand der lokalen Arbeitsumgebung wiederhergestellt werden kann (2002). Nach dem Update kann nun der Entwickler-Build ausgeführt werden, bei dem es sich entweder um einen inkrementellen oder auch um einen vollständigen Build handeln kann. Daneben sind auch Tests durchzuführen, um sicherzugehen, dass die vollzogenen Änderungen gemeinsam mit dem restlichen Quellcode und den zusammengehörigen Komponenten korrekt ausgeführt werden können (Berczuk et

al. 2003). Als letzter Schritt wird der geänderte und korrekt ausführbare Quellcode per Check-in wieder in das Repository eingelesen, sodass es für den Rest des Entwicklungsteams sichtbar wird.

Der Entwickler-Build kann in der lokalen Arbeitsumgebung des Entwicklers/der Entwicklerin ein Mal oder auch mehrmals im Zuge der Durchführung von Änderungen am Quellcode ausgeführt werden. Wichtig ist dabei, dass dieser auf jeden Fall ausgeführt wird, bevor die Änderungen per Check-in wieder in das Repository hinzugefügt werden. Der Entwickler-Build hilft nicht nur den Quellcode fehlerfrei und konsistent zu halten, sondern ermöglicht auch die Integrierung von Erweiterungen und Änderungen in regelmäßigen sowie dicht frequentierten Abständen (Berczuk et al. 2003).

Auch wenn die isolierte Betrachtung und Arbeit in der lokalen Arbeitsumgebung für EntwicklerInnen vorteilhaft sein kann, ist es dennoch unverzichtbar das der geänderte Quellcode mit dem Rest des Systems zusammenpasst. Aus diesem Grund gibt es einige Anforderungen, die der Entwickler-Build zu erfüllen hat. Zuallererst ist anzumerken, dass der Entwickler-Build „einfach zu bedienen und vor allem schnell“ (Popp 2006, S. 192) ausführbar sein sollte. Desweiteren sollte er soweit wie möglich dem Integrations- und Release-Build ähnlich sein. Es können bei Bedarf diesbezüglich jedoch manche Schritte übersprungen werden, wie beispielsweise die Erstellung des Installationspakets. Abgesehen davon sollten aber zumindest derselbe Compiler, dieselben Versionen externer Komponenten und dieselbe Verzeichnisstruktur, wie bei den anderen beiden Build-Varianten, verwendet werden. Daneben ist es zu beachten, dass der Entwickler-Build alle notwendigen Abhängigkeiten und auch alle Komponenten beinhaltet, die von den Änderungen betroffen sind. Welche Komponenten beim Entwickler-Build miteinbezogen werden sollten, können mit Hilfe einer guten Projektarchitektur leichter bestimmt werden (Bercuk et al. 2003b). Hunt und Thomas empfehlen zudem, dass alle EntwicklerInnen im Projekt zumindest dieselben Build-Tools zur Kompilierung benutzen und der Kompilierungsprozess am Besten automatisiert werden sollte (2003).

Wie zuvor erwähnt, kann der Entwickler-Build sowohl inkrementell, als auch vollständig durchgeführt werden. Es gibt zwar einige wichtige Aspekte die für einen vollständigen Build sprechen, so z.B. dass sich dadurch die Wahrscheinlichkeit minimiert, dass gewisse Abhängigkeiten übersprungen werden, jedoch ist es dennoch nicht außer Acht zu lassen, dass ein vollständiger Build wesentlich mehr Zeit in Anspruch nimmt als ein inkrementeller Build. Wenn also alle Abhängigkeiten berücksichtigt und korrekt gehandhabt werden, steht der fehlerfreien Ausführung inkrementeller Builds nichts im Wege, wobei diese Form insofern von den meisten EntwicklerInnen bevorzugt wird (Berczuk & Appleton 2002).

5.2 Integrations-Build

Der Integrations-Build dient der Integration aller unlängst vollzogenen Änderungen der EntwicklerInnen in ihrer Gesamtheit, sodass eine einzelne konsistente und lauffähige Version des Produktes erstellt wird. Die spezielle Aufgabe des Integrations-Builds ist die zentrale Koordinierung und Synchronisierung der voranschreitenden Quellcodegenerierung in kleinen Einheiten und regelmäßigen Zeitabständen, sodass es letztendlich nicht zu einem *Big Bang Effekt* mit negativen Überraschungen kommt. Der Integrations-Build kann als inkrementeller, oder aber als vollständiger Build ausgeführt werden, wobei es sich vorzugsweise um einen vollständigen Build handeln sollte, der in dichten Zeitintervallen ausgeführt wird. Grundsätzlich ist auch eine Kombination des inkrementellen und des vollständigen Builds zu bestimmten Szenarien möglich. Der Integrations-Build sollte zumindest nach Fertigstellung von Änderungen am Quellcode (als *ereignisgesteuerter Build* (z.B. nach dem Check-in) ausgeführt werden, kann aber auch ein Mal pro Woche, täglich (als *Daily/Nightly Build*) oder mehrmals am Tag (*Continuous Integration*) stattfinden (vgl. Popp 2006; Lee 2008; Berczuk et al. 2003).

Im Kapitel zum Entwickler-Build wurde bereits angesprochen, dass Änderungen am Quellcode in der lokalen Arbeitsumgebung des Entwicklers/der Entwicklerin durchgeführt werden. Diese unabhängig voneinander getätigten Änderungen müssen jedoch zueinander passen und zu einem gemeinsamen Ganzen integrierbar sein. Meist ist nach dem Entwickler-Build dennoch nicht auszuschließen, dass der Quellcode fehlerfrei ist. Auch parallele Änderungen am Quellcode können noch nach der Hinzufügung in die *Mainline* Inkonsistenzen verursachen (Berczuk & Appleton 2002). Als Mainline bezeichnet man die zentrale Codeline des Projekts, die als Hauptpfad für die Entwicklung dient. Sie stellt die Basis für alle Verzeigungen dar, die als Subbranches bezeichnet werden und nach der Fertigstellung wieder in die zentrale Codeline integriert werden (Vance 1998).

Eine gute Kommunikation zwischen den Mitgliedern des Entwicklungsteams ist ein wichtiger Aspekt, um Integrationsproblemen vorzubeugen und das gesamte System fehlerfrei und funktionsfähig zu halten. Aus diesem Grund ist es notwendig, einen vollständigen und sauberen Integrations-Builds durchzuführen, bei dem alle Änderungen mitsamt allen Anhängigkeiten in den Prozess miteinbezogen werden. Je früher und dicht frequentierter der Integrations-Build zum Einsatz kommt, umso leichter können Fehler und Inkonsistenzen entdeckt und anschließend behoben werden. Dies ist nicht nur zeitsparend, sondern verringert zudem das Risiko eines Build-Zusammenbruchs, der den Entwicklungsfortschritt des gesamten Teams verlangsamen würde. Daher ist es wichtig Inkonsistenzen und Fehler auf rasche und automatisierte Weise zu ermitteln. Der Integrations-Build Prozess sollte im Allgemeinen bestimmte Anforderungen erfüllen. Dazu gehört, dass dieser Reproduzierbar und dem Release-Build so ähnlich wie möglich sein sollte. Außerdem sollte der Prozess vollständig automatisiert und ein Benachrichtigungs- oder Protokollierungsmechanismus eingesetzt werden, um Inkonsistenzen und Fehler

schneller identifizieren zu können. Ermittelte Fehler werden sodann dem/der Build-ManagerIn und auch demjenigen/derjenigen EntwicklerIn gemeldet, von dem/der die jeweilige fehlerhafte Änderung hinzugefügt wurde (Berczuk & Appleton 2002).

Grundsätzlich kann beim Integrationsprozess eine Differenzierung in *Software Integration* und *System Integration* vorgenommen werden, deren Unterschiedlichkeiten hier kurz angesprochen werden sollen. Bei der *Software Integration* [6] werden kleinere Integrationsschritte vollzogen, sodass bereits bei kleinsten Erweiterungen bzw. Änderungen am Quellcode ein Integrations-Build ausgeführt wird, damit EntwicklerInnen eventuelle Fehler im Quellcode augenblicklich beheben können. Bei der *System Integration* hingegen wird die gesamte Betriebssystemumgebung, die sich aus Subsystemen (z.B. Komponenten, externe Systeme, etc.) zusammensetzt, zu einem einzigen System integriert. Dabei werden die Subsysteme mit einem System Integrations-Build zusammengefügt, um sicherzustellen, dass sie gemeinsam ein funktionsfähiges System bilden (Lee 2008).

Nach Popp sollten die Zeitintervalle, in denen der Integrations-Build ausgeführt wird, prinzipiell der jeweiligen Projektphase angepasst werden (2006). Es gibt jedoch verschiedenste Integrations-Modelle, welche die Ausführung des Integrations-Builds zu unterschiedlichen Anlässen und Zeitintervallen verlangen. Bei diesen Modellen handelt es sich um die Meilenstein Integration, Staged Integration, Komponenten Integration und Kontinuierliche Integration (Lee 2008).

➢ Meilenstein Integration

Wie auch dem Namen entnommen werden kann, wird bei dieser Variante der Integrations-Build zu einem festgelegten spezifischen Meilenstein ausgeführt. Dabei werden die Software Komponenten in isolierten Arbeitsumgebungen entwickelt und nach der Erreichung des Meilensteins zusammen, mittels eines System Integration Builds, in die Mainline integriert. Da während der Entwicklung von Komponenten zwischenzeitlich keine Integrations-Builds ausgeführt werden, ergibt sich dadurch ein hohes Risiko für Fehleranfälligkeiten und Inkonsistenzen, die sich erst im Zuge des System-Integration Builds herausstellen. Auf Grund dieser Umstände wird dieses Integrationsmodell auch als *big-bang Integration* bezeichnet, deren Einsatz möglichst vermieden werden sollte (Lee 2008). In Abb.31 ist die gemeinsame Integration zweier Komponenten (A und B) zu einem bestimmten Meilenstein in die Mainline dargestellt.

[6] Diese Form der Integration wird auch als *Develop-Build-Fix Modell* bezeichnet.

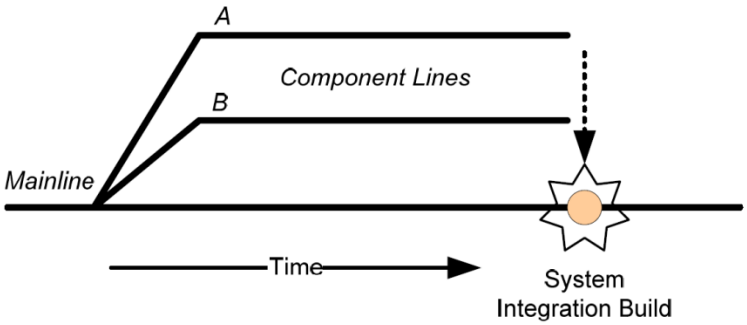


Abbildung 28: Meilenstein Integration (aus Lee 2008, S. 45)

➢ Staged Integration

Bei diesem Integrationsmodell wird die Entwicklung auf der Mainline fortgesetzt und der Entwicklungsfortschritt kontinuierlich integriert, während parallel dazu auf der Ebene von Subprojekten und deren *Subproject Lines*, eine weitere Entwicklung stattfindet. Beispielsweise kann diese Form der Integration nach der Herausgabe eines Release notwendig sein, bei dem einige Fehler festgestellt wurden und zu beheben sind, während zugleich die Entwicklung für das nächste Release fortgesetzt werden muss. Zu diesem Zweck wird ein kurzlebiges Branch erstellt, das zugleich als Subprojekt aufgefasst werden kann, an dem die notwendigen Änderungen und Problembehebungen durchgeführt und kontinuierlich integriert werden. Ist die Entwicklung am Subproject Line fertiggestellt worden, so wird das Subprojekt mit einem System-Integration Build wiederum der Mainline hinzugefügt (Lee 2008). Abb. 32 zeigt die parallele Entwicklung auf der Ebene der Mainline und der Subproject Lines, sowie deren Integration.

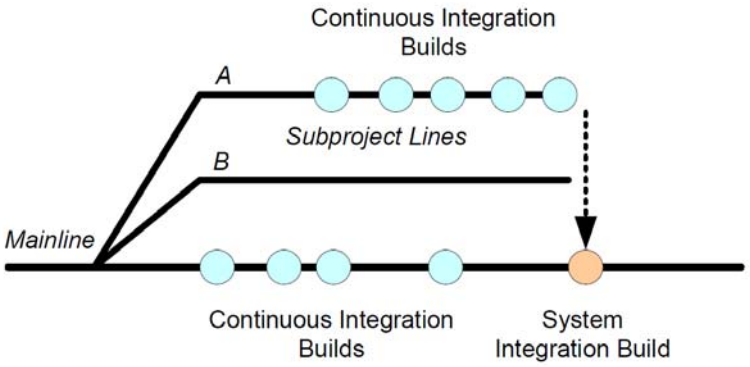


Abbildung 29: Staged Integration (aus Lee 2008, S. 47)

➢ Komponenten Integration

Besonders umfangreiche Systeme bestehen aus mehreren Komponenten, die oftmals wie eigenständige Projekte behandelt und voneinander unabhängig entwickelt werden. Da es jedoch wichtig ist, auch mögliche Zwischenbeziehungen unter diesen Komponenten zu berücksichtigen, ist an dieser Stelle die Ausführung der Komponenten Integration von besonderer Bedeutung. Um mögliche Komplikationen zu vermeiden, sollte die Komponenten Integration in dicht frequentierten Zeitabständen stattfinden, auch wenn die Funktionalität der Komponente nur teilweise fertiggestellt wurde. Andernfalls hätte die Komponenten Integration keinen Unterschied zur Meilenstein Integration. Für die isolierte Entwicklung der Komponenten werden eigene *Component Lines* verwendet, auf denen auch regelmäßig kontinuierliche Integrations-Build ausgeführt werden, bevor die verschiedenen Komponenten endgültig über das System Integration Build in die Mainline integriert werden (Lee 2008). In Abb. 33 ist eine solche Komponenten Integration zu sehen, bei der die Komponenten A und B isoliert entwickelt werden. Nach Fertigstellung bestimmter Funktionalitäten für die jeweilige Komponente wird eine Baseline (A2 bzw. B1) erstellt wird. Bei einer frühzeitigen Fertigstellung einer bestimmten Funktionalität wird die jeweilige Baseline (siehe B1) in das System Integriert und die System-Baseline SYS1 erstellt. Diese System Baseline dient nun als Grundlage für weitere Integrationen, sodass sie bei der Integrierung der Baseline A2 herangezogen und nach einem erneuten System Integrations-Build als SYS2 festgehalten wird.

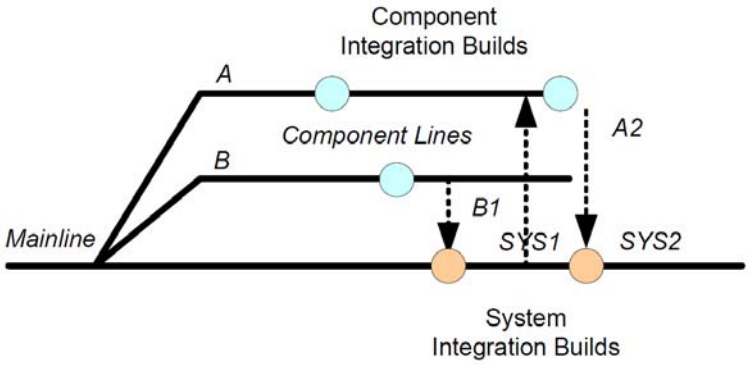


Abbildung 30: Komponenten Integration (aus Lee 2008, S. 48)

➢ Kontinuierliche Integration

Fowler definiert die kontinuierliche Integration (*Continuous Integration*) folgendermaßen (2006): „Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.“

Nach dem kontinuierlichen Integrationsmodell wird das gesamte System mehrmals täglich einem automatisierten Integrations-Build unterzogen. Dabei wird jede kleinste Änderung, die per Check-in dem Repository hinzugefügt wurde, direkt in die Mainline integriert (Lee 2008). Die einzelnen Schritte der kontinuierlichen Integration beginnen, nachdem der/die EntwicklerIn seiner/ihre Änderungen am Quellcode im Repository abgelegt hat. Sodann wird im Repository in regelmäßigen Zeitabständen (z.B.: alle 10 oder 30 Minuten), nach neu hinzugefügten Änderungen gesucht. Gibt es solche, dann wird die Ausführung des Build-Skripts zur Integrierung dieser Änderungen in das System veranlasst. Am Ende des Integrations-Builds, findet die automatisierte Generierung von Feedback, welches anschließend an die EntwicklerInnen geschickt wird, statt. Danach wird derselbe Vorgang kontinuierlich wiederholt, indem immer wieder die Suche nach neuen Änderungen im Repository beginnt. Der eigentliche Gedanke hinter dem kontinuierlichen Integrationsmodell ist es, Probleme und Fehler im System frühzeitig entdecken und beheben zu können. Zu diesem Zweck sollte die kontinuierliche Integration so früh wie möglich in das Projekt implementiert werden. Die kontinuierliche Integration reduziert nicht nur das Risiko großen Integrationsproblemen zu begegnen, sondern hilft im Allgemeinen auch die Integrationszeiten zu reduzieren, das System fehlerfrei und gesund zu halten und die Qualität der Software zu bewahren (Berczuk & Appleton 2002). In Abb. 34 ist ein beispielhafter Integrationsprozess entlang der Mainline mit kontinuierlichen Integrations-Builds zu sehen.

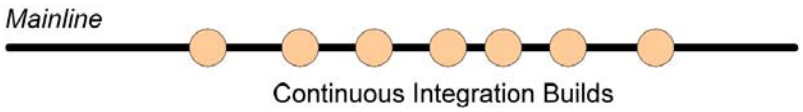


Abbildung 31: Kontinuierliche Integration (aus Lee 2008, S. 46)

Ein wichtiger Aspekt ist, dass die Build-Infrastruktur in jedem Fall den Bedingungen des Continuous Integration Modells angepasst werden sollte, sodass die Laufzeit des Integrations-Builds nicht den geplanten Zeitintervall für die Suche nach neuen Änderungen überschreitet (siehe Kapitel 2.3.4 zur Skalierbarkeit und 2.4.1 zur Build-Infrastruktur).

5.3 Release-Build

In seiner technischen Grundstruktur ist ein Release-Build (Produktions-Build) dasselbe wie ein Integrations-Build, unterscheidet sich jedoch auf Grund der Eigenschaft, dass die Build-Outputs für Installations- und Verteilungszwecke paketiert werden. „Wird eine Version des Softwareproduktes an die Endanwender ausgeliefert, bezeichnen wir diese als Release“ (Popp 2006, S. 53). Das Ergebnis des Release-Builds ist somit das Installationspaket [7], welches für die KundInnen des Entwicklungsteams angefertigt wird. Da nicht alle Tage ein Release der Software herausgegeben und der Zeitpunkt vorher vom Entwicklungsteam gemeinsam mit den KundInnen bestimmt wird, ist es verständlich, dass der Release-Build wesentlich seltener ausgeführt wird als der Integrations-Build (Berczuk et al. 2003).

Ein wichtiger Aspekt vor der Erstellung und Ausführung des Release-Builds ist die Bestimmung, welche Artefakte dem Release-Paket hinzugefügt werden sollen. Bei der Erstellung des Release-Builds ist es zu berücksichtigen, dass dieselben Input-Dateien und Build-Skripte wie beim Integrations-Build verwendet werden sollten. Zudem ist es wichtig den Release-Build in einer kontrollierten bzw. isolierten Umgebung auszuführen, damit sich das erzeugte Produkt zurückverfolgen lässt und auch korrekt und prüffähig ist. Bevor der Release-Build ausgeführt wird, sollte diese Umgebung einem Update unterzogen und notwendige Fehlerbehebungen durchgeführt werden. Die Build Outputs, welche nach der Ausführung des Release-Builds ausgegeben werden, sind meist langlebig, sodass sie bei späteren Build- bzw. Deployment-Prozessen wiederum als Inputartefakte verwendet werden können. Vergleichsweise sind die Outputs der Integrations-Builds sehr kurzlebig, da sie bei der Ausführung des nächsten Integrations-Builds überschrieben werden (Lee 2006).

Nachdem das Release-Build das Produkt erzeugt hat, wird dieses mit einer Version (sog. Release-Nummer) gekennzeichnet. Eine solche Versionierung wird beim Integrations-Build beispielsweise nicht vorgenommen, da die Ergebnisse dieses Builds lediglich zur Kenntnisnahme seitens Mitglieder des Entwicklungsteams gedacht sind. Beim Integrations-Build wird außerdem der Schritt zur Erstellung des Installationspakets Verteilung an KundInnen übersprungen. Im Gegensatz dazu muss das, an KundInnen ausgelieferte, Produkt eine Release-Nummer (Versionsnummer) zur eindeutigen Kennzeichnung enthalten. Daneben werden im Zuge des Release-Builds automatisch zusätzliche Informations- und Dokumentationsmaterialien generiert (z.B. Bill of Materials, AnwenderInnen-Handbuch, etc.) und dem Paket hinzugefügt (Berczuk et al. 2003).

Der Inhalt des Release kann, den Erwartungen der KundInnen entsprechend, etwas unterschiedlich sein, aber es gibt gewisse Bestandteile, die ein Release unbedingt enthalten

[7] Siehe Kapitel 4.4.1 *Installationspaket*.

sollte. Dazu gehört die Vergabe eines eindeutigen Namens und einer Release-Nummer, welche sich typischerweise aus der Major- und der Minor-Release-Nummer zusammensetzt. Das Release sollte zudem erst dann herausgegeben werden, wenn alle Funktionalitäten, die von KundInnen angefordert wurden, fertiggestellt worden sind. Das Release sollte auf jeden Fall vollständig sein, d.h. alle Dokumentationen und Dateien enthalten, die zur Ausführung der Software notwendig sind. Falls die Installation des Release mehrere Schritte erfordert, sollten diese mit Hilfe eines einzigen Installationsskripts oder Hilfsprogramms ausführbar sein (Clark 2006).

Wie zuvor erwähnt, erhält jedes Release zur eindeutigen Kennzeichnung eine spezifische Release-Nummer. Diese Nummer ist für gewöhnlich dreigliedrig und setzt sich aus der Major-, Minor- und Patch-Nummer zusammen. Die *Major-Release-Nummer* wird nur dann geändert, wenn signifikante Änderungen am Produkt vorgenommen wurden, wie beispielsweise eine vollständige Überarbeitung der Benutzeroberfläche oder bei Änderung des verwendeten Dateiformats. Weitere Gründe können auch die Hinzufügung neuer Subsysteme und Komponenten sein. Die Erhöhung der Major-Release-Nummer vermittelt den AnwenderInnen, dass größere Veränderungen am Produkt vorgenommen worden sind. Das *Minor-Release* (Wartungs-Release) erscheint hingegen in regelmäßigeren Abständen, denn hier findet hauptsächlich die Behebung von Fehlern statt. Im Zuge eines Minor-Release werden funktionale Erweiterungen nur dann integriert, wenn damit keine größeren Auswirkungen auf das System zu erwarten sind. Eine *Patch-Nummer* wird dem Release normalerweise äußerst selten hinzugefügt und ist eigentlich für Ausnahmefälle gedacht, bei denen in kürzester Zeit die Behebung kritischer Fehler am Produkt für KundInnen notwendig ist. Patches dienen lediglich der Fehlerbehebung und dürfen keinerlei funktionale Erweiterungen enthalten. Bei der Planung von Releases sollte ein angemessener zeitlicher Abstand zwischen den Major- und Minor-Releases gewählt werden. Besonders bei der Einplanung der Zeitintervalle zwischen den einzelnen Minor-Releases sollte darauf geachtet werden, dass genügend Zeit für die Erstellung neuer Funktionalitäten und Behebung von Fehlern eingeräumt wird. Ist der zeitliche Abstand zu kurz, werden EntwicklerInnen unter Druck geraten, die notwendigen Änderungen zeitgerecht fertigzustellen. Bei zu langen Release-Abständen würden sich wiederum die Test- und Vorbereitungsphasen ausdehnen, sodass die Erstellung *paralleler Entwicklungspfade* (sog. *Branching*) notwendig wird. Das Gegenstück zum parallelen Entwicklungspfad ist der *lineare Entwicklungspfad*. Vor der Ausführung des Release-Builds müssen Vorbereitungen zur Erstellung der Baseline getroffen werden, die beim linearen Entwicklungspfad mit einem *Code-Freeze* eingeleitet werden. Bevor dieser in Kraft tritt, müssen alle Änderungen an den Dateien abgeschlossen und das Repository aktualisiert worden sein, wobei der aktuelle Zustand mit einem Tag festgehalten wird. Während dem Code-Freeze dürfen Änderungen an den Dateien nur unter kontrollierten Bedingungen vorgenommen werden, wobei eine vorherige Festlegung von erlaubten und unerlaubten Änderungen ratsam ist. Beispielsweise werden in diesem Fall Fehlerbehebungen erwünscht, jedoch die Implementierung neuer Funktionalitäten gänzlich

zu vermeiden sein. Das erzeugte Release wird letztendlich mit einer Baseline im Repository festgehalten, sodass anschließend der Code-Freeze wieder aufgehoben werden kann (Popp 2006). Als Alternative zum linearen Entwicklungspfad kann das Branching-Modell (paralleler/verzweigter Entwicklungspfad) eingesetzt werden, um Problemen, wie dem hohen Zeitverbrauch und dem Verbot zur Erstellung neuer Funktionalitäten, die mit dem Code-Freeze verbunden sind, auszuweichen.

5.3.1 Branches

Das Branching ermöglicht die parallele Entwicklung eines neuen Systems und unterstützt die Vorbereitung mehrerer Releases, wobei jedes Branch eine eindeutige Kennzeichnung erhält. Sobald die Vorbereitungen für das Release beginnen, wird der Entwicklungspfad (*Codeline*) in zwei separate Branches aufgeteilt, wovon der Hauptentwicklungspfad als *Trunk*, und der andere Pfad als *Release-Branch* bezeichnet wird. Während auf dem Trunk die Entwicklung wie gewohnt fortgesetzt wird, findet parallel dazu am Release-Branch die Vorbereitung für das Release statt. Zum Zeitpunkt der Erstellung des Release-Branch werden alle aktuellen Dateien des Hauptentwicklungspfades dupliziert, sodass der Release Branch die Kopien all dieser Dateien enthält. D.h. an der Verzeigungsstelle enthalten beide Pfade noch dieselben Dateiversionen, die sich dann im Laufe der Zeit auseinander entwickeln. Am Ende des Release-Branches werden die beiden Entwicklungspfade wiederum zusammengefügt, sodass die Änderungen und Bug-Fixes (Fehlerbehebungen) im Release-Branch in den Hauptentwicklungspfad integriert (*Merging*) werden. Ein bedeutender Vorteil ist, dass auch nach der Herausgabe des Release der Release-Branch weiterhin verfügbar bleibt, sodass dieser für spätere Fehlerbehebungen nochmals erstellt werden kann. Beim Release-Branch sind dieselben Einschränkungen wie beim linearen Entwicklungspfad, mitsamt dem Code-Freeze, einzuhalten. Daher findet in diesem Zusammenhang keine echte Parallelentwicklung statt, denn diese würde eine parallele und uneingeschränkte Pflege und Weiterentwicklung mehrerer Releases verlangen, sodass auch den Release-Branches neue Funktionalitäten hinzugefügt werden können (vgl. Clark 2006; Popp 2006).

Zur Umsetzung der parallelen Entwicklung gibt es verschiedene Branching-Modelle, die nach dem Kriterium ausgewählt werden sollten, ob sie für die jeweilige Projektumgebung geeignet sind (Walrad & Strom 2002). Zu diesen Branching-Modellen gehören: *Branch-by-Release, Branch-by-Purpose, Branch-by-Task, Branch-by-Change-Request,* sowie *Personal-Activity-Branch*. Die Eigenheiten der aufgezählten Branching-Modelle sollen an dieser Stelle etwas näher beleuchtet werden.

➢ Branch-by-Release

Nach diesem Modell wird ein Branch erst dann angelegt, wenn eine neue Version des Produkts erstellt werden soll. In diesem Sinn wird zum Auslieferungszeitpunkt eines Release ein neuer Branch für die Entwicklung des nächsten Release erstellt. Der Grundgedanke bei diesem Modell ist, dass der neue Release-Branch als Baseline für die Fortsetzung des Entwicklungsprozesses dient, während der Vorgänger-Release als letzte stabile und konsistente Version des Produktes festgehalten wird, der seine gesamte Funktionalität dem Folge-Release weitervererbt hat. Auf diese Weise besteht stets die Möglichkeit, bei der Notwendigkeit von Bug-Fixes, zum älteren Release zurückzukehren und notwendige Änderungen durchzuführen, ohne den aktuellen Entwicklungsprozess damit zu belasten. Als Kehrseite dieses Modells zeigen sich jedoch einige Schwächen, sodass das Branch-by-Release Modell nicht als besonders entwicklerInnenfreundlich beurteilt werden kann. Da sich die Anzahl der verfügbaren Branches mit jedem neuen Release erhöht, kann es bei größeren Projekten mitunter problematisch sein, den Überblick über die Release-Branches zu behalten und Änderungen jeweils am richtigen Pfad durchzuführen. Bedingt durch den Code-Freeze, der während der Vorbereitungsphase des aktuellen Releases herrscht, ist, abgesehen von eingeplanten Funktionalitäten, keine Hinzufügung von Änderungen bzw. Erweiterungen am Code erlaubt. Diese können erst nach der Auslieferung des aktuellen Release in dessen Entwicklungspfad aufgenommen werden. Ein weiterer nachteilhafter Aspekt entsteht dadurch, dass Bug-Fixes, die an einer älteren Release-Version durchgeführt wurden, jedem einzelnen Folge-Release integriert werden müssen, bis sie letztendlich auch dem aktuellen Release-Branch hinzugefügt werden können (vgl. Berczuk & Appleton 2002; Walrad & Strom 2002; Ruminer 2006). Die Abb. Zeigt den typischen Verlauf der Entwicklung beim Branch-by-Release Modell.

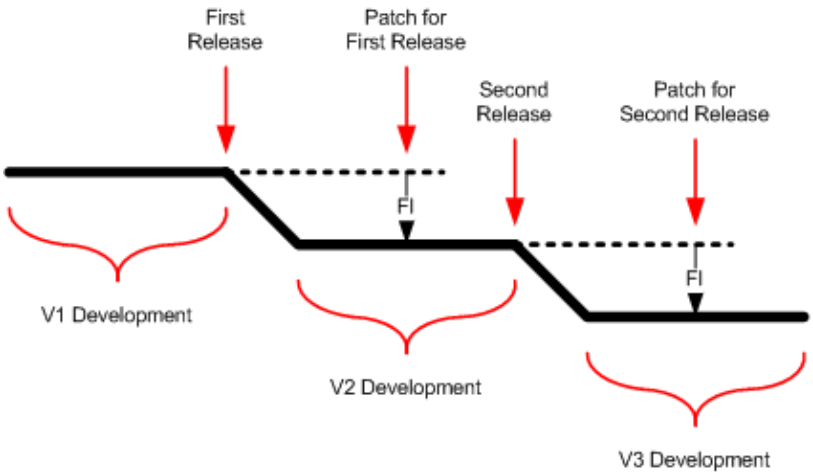


Abbildung 32: Branch-by-Release Modell (aus Denny 2006)

➢ Branch-by-Purpose

Mit dem Branch-by-Purpose Modell (sog. *Mainline Development*) können einige Nachteile, die beim Branch-by-Release Modell entstehen, umgangen werden. Die Entscheidung, einen Branch anzulegen, ist dabei abhängig von einer bestimmten Absicht (Purpose), die zu erfüllen ist. Sobald die Vorbereitungen für ein Release entlang der Mainline beinahe fertiggestellt sind und der Code einen stabilen Status erreicht hat, wird eine, extra für dieses Release bestimmte, Verzweigung der Mainline als QA- (Quality Assurance) Branch angelegt. Dieser separate Branch ist für unterstützende Entwicklungsmaßnahmen zum aktuellen Release gedacht und dient nicht wie beim Branch-by-Release Modell als Entwicklungspfad für das nächste Release. Somit kann die Entwicklung von Funktionalitäten und die Behebung von Fehlern auf der Mainline konsequent fortgesetzt werden, während die Releases einen eigenen Branch erhalten. Der Vorteil dabei ist, dass die Entwicklung auf der Mainline nicht durch einen Code-Freeze unterbrochen werden muss und die parallele Entwicklung an mehreren Subprojekten möglich ist (vgl. Berczuk & Appleton 2002; Walrad & Strom 2002; Ruminer 2006). In Abb. 36 ist das geschilderte Branch-by-Purpose Modell übersichtlich dargestellt.

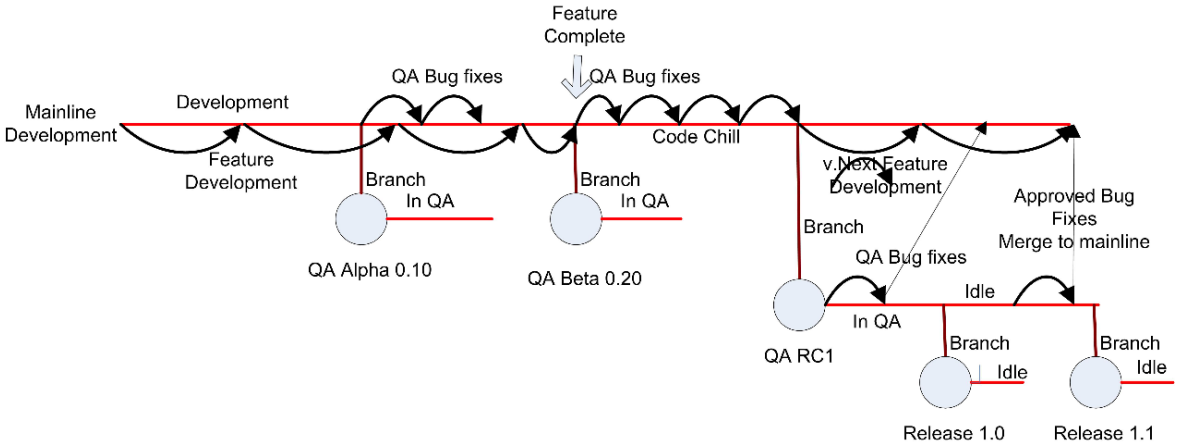


Abbildung 33: Branch-by-Purpose Modell (aus Ruminer 2006)

➢ Branch-by-Task

Bei diesem Modell wird ein eigener Activity-Branch angelegt, auf dem alle Tasks, die Auswirkungen auf den aktuellen Hauptentwicklungspfad hätten, umgesetzt werden. Jeder Task wird per Branching auf einem separaten Pfad durchgeführt und nach seiner Fertigstellung dem Activity-Branch hinzugefügt. Wenn alle vorgesehenen Tasks abgeschlossen und die Aktivität in ihrer Gesamtheit stabil und funktionsfähig ist, wird der Activity-Branch schließlich in den Hauptentwicklungspfad integriert. Alternativ dazu, kann ein Activity-Branch auch nur für umfangreichere Tasks (Major-Tasks) erstellt und

verwendet werden, während kleinere Tasks (Minor-Tasks) ohne Branching direkt am Activity-Branch vollzogen werden (Appleton et al. 1998).

- **Branch-by-Change-Request**

Dieses Modell (unter anderem als *CR-Branch, Change-Set Branch* oder *Change-Package Branch* bezeichnet) kommt zum Einsatz, wenn Änderungswünsche (Change-Requests) der KundInnen eintreffen. Hierzu wird ein eigener Branch zur isolierten Durchführung der gewünschten Änderungen angelegt, auf dem jeweils nur eine einzelne Funktionalität umgesetzt bzw. eine einzige Änderung (z.B. Fehlerbehebung oder Erweiterung) an der Funktionalität vorgenommen wird (Appleton et al. 1998).

- **Personal-Activity-Branch**

Dieses Modell wird unter Anderem auch als *Single-User Task Branch* bezeichnet, wobei der Branch zur Durchführung eines Tasks erstellt wird, für den der/die alleinige InhaberIn (EntwicklerIn) des Tasks zuständig ist. Durch die Verwendung eines speziellen Sperrmechanismus kann der Personal-Activity-Branch zugleich als *Private Branch* dienen, bei dem niemand anderes als der/die EigentümerIn des Branches diesen verwenden darf (Appleton et al. 1998).

6. Developmentmethoden

Mit den verschiedenen Developmentmethoden kommt ein weiterer Aspekt hinzu, der sich auf den Build-Prozess auswirkt, denn jede einzelne dieser Methoden verlangt, dass Builds bestimmte Anforderungen erfüllen. Prinzipiell sollten sich EntwicklerInnen nicht nach dem Build-Prozess richten müssen, sondern vielmehr ein Build-Prozess gewählt werden, der dem Arbeitsstil des Entwicklungsteams angepasst ist. (vgl. Lee 2008; Berczuk et al. 2003)

Unter Kapitel 4.2.1 ist im Allgemeinen bereits der Einfluss der gewählten Programmiersprache auf den Verlauf des Build-Prozesses dargestellt worden. In diesem Kapitel werden nun drei der meistverwendeten Developmentmethoden genauer betrachtet: darunter Agile Development, Open Source Development und Traditional Enterprise Development.

6.1 Agile Development

„Agile Entwicklung ist eine Philosophie, ein Metamodell, auf das verschiedene Ausprägungen aufbauen [….]“ (Hüttermann 2008, S. 3). Als Ausprägungen kommen dabei verschiedenste agile Entwicklungsmethoden in Frage, wie beispielsweise eXtreme Programming, Dynamic System Development Method (DSDM) und Scrum. Alle genannten Methoden vereint eine gemeinsame Grundlage, das Agile Manifesto. Dieses Manifesto ist im Zuge eines Treffens der *Agile Alliance* in der Zeit vom 11. bis 13. Februar 2001 in Chicago zusammengestellt worden. Die Agile Alliance setzt sich aus 17 Mitgliedern zusammen, die als qualifizierte Persönlichkeiten der Softwareentwicklungswelt bzw. Vertreter einzelner agiler Methoden bekannt sind. Während des Zweitagesmeetings konnten sich die Teilnehmer auf gemeinsame Grundprinzipien der agilen Entwicklung einigen, die schließlich in das schriftliche *Manifesto for Agile Software Development* eingeflossen sind. (Highsmith 2001) Die fundamentalen Prinzipien aus diesem Manifesto können folgendermaßen resümiert werden (agilemanifesto.org):

1. Individuals and interactions over processes and tools.
2. Working software over comprehensive documentation.
3. Customer collaboration over contract negotiation.
4. Responding to change over following a plan.

Eine konkretere Ausführung und Beschreibung dieser Kernaussagen ist in den 12 Prinzipien [8] des Manifests enthalten, auf die hier kurz eingegangen werden soll (Hüttermann 2008):

„Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
Das allererste und höchste Prinzip des agilen Manifests ist die Kundenzufriedenheit. Um diese Zufriedenheit gewährleisten zu können, ist es erforderlich, dass EntwicklerInnen ihren KundInnen bereits frühzeitig und kontinuierlich die Software bereitstellen und ausliefern. Dadurch können KundInnen den aktuellen Stand der Entwicklung nachverfolgen, mit der Software arbeiten, diese analysieren und bei Unzufriedenheit oder Änderungswünschen rechtzeitig intervenieren. Somit wird den KundInnen eine aktive Rolle innerhalb des Softwareentwicklungszyklus beigemessen.

„Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."
Innerhalb eines traditionellen Entwicklungszyklus findet jede einzelne Phase, von der Analyse bis zur Auslieferung, gleich im Anschluss an die Vorangehende statt, wobei in der Zwischenzeit die funktionsfähige Software bis zur Fertigstellung nicht an KundInnen ausgeliefert wird. Da die Anforderungen an das Produkt in der anfänglichen Analysephase bestimmt werden und der gesamte Softwareentwicklungszyklus einen langen Zeitraum in Anspruch nimmt, ist es zu erwarten, dass sich in der Zwischenzeit die Marktbedingungen ständig ändern. Als Konsequenz wird die Konkurrenzfähigkeit des ausgelieferten Produkts und in weiterer Folge jene der KundInnen in Mitleidenschaft gezogen. In der agilen Entwicklung wird ein anderer Ansatz bevorzugt, wonach zur Berücksichtigung der Anforderungen und Bedingungen am Markt und Verbesserung der allgemeinen Konkurrenzfähigkeit, zu jeder, auch noch so späten Phase des Entwicklungszyklus Änderungs- und Anpassungswünsche der KundInnen entgegenzunehmen und zu berücksichtigen sind.

„Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale"
In agilen Projekten wird iterativ gearbeitet, wobei eine Iteration als ein einzelner abgeschlossener Entwicklungszyklus zu verstehen ist, in dem alle Schritte des üblichen Softwareentwicklungszyklus durchlaufen werden. Das Team konzentriert sich dabei immer nur auf bestimmte Funktionalitäten, die zu einem späteren Zeitpunkt weiter ausgebaut werden können.
Falls eine Funktionalität nicht in die aktuelle Iteration aufgenommen wird, kann sie als *backlog* gespeichert werden, sodass sie für eine spätere Iteration zur Verfügung steht. (Lee 2008) In Abb. 28 wird die Priorisierung der gewünschten Funktionalitäten und

[8] Für die 12 Prinzipien siehe agilemanifesto.org, Principles.

Kundenanfragen und die davon abhängige Implementierung in die aktuelle Iteration dargestellt.

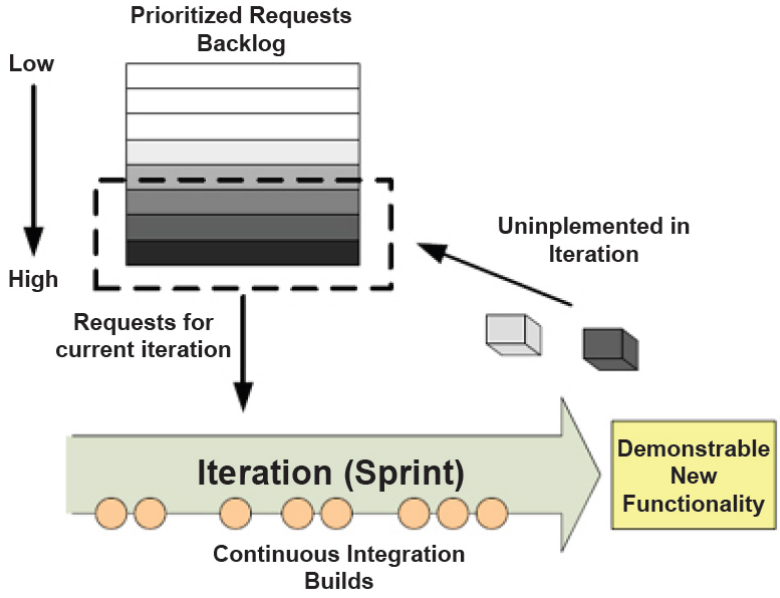


Abbildung 34: Example Agile Method (aus Lee 2008, S. 85)

Wichtig ist, dass die einzelnen funktionsfähigen Softwareteile in regelmäßigen Zeitabständen an KundInnen überreicht werden, wobei die Präferenz stets bei kürzeren Zeitintervallen liegen sollte.

„Business people and developers must work together daily throughout the project.“
Die vorangehenden Prinzipien haben schon verdeutlicht, dass auch KundInnen eine wichtige Rolle innerhalb des agilen Entwicklungsprozesses zukommt. Dieses Prinzip unterstreicht nochmals die Notwendigkeit der Zusammenarbeit zwischen EntwicklerInnen und KundInnen durch das ganze Projekt hindurch. Dadurch wird eine Synergie hergestellt, welche die Kommunikation zwischen beiden Parteien verbessert, sodass Missverständnisse leichter aus dem Weg geräumt werden können. Diese Bedingungen ermöglichen, dass sich KundInnen vermehrt auf die Anforderungen und die Verbesserung der gewünschten Software konzentrieren können.

„Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.“
Unter jedem Projektdach werden verschiedene Individuen mit unterschiedlichen Charakteren, Fähigkeiten, Eigenheiten und kreativen Zugängen zusammentreffen. Deshalb gilt es, diese Eigenschaften in das Projekt einfließen zu lassen und der individuellen Natur der MitarbeiterInnen Respekt zu gebühren. Auf dieser Basis sind die

ProjektmitarbeiterInnen mit der notwendigen Umgebung und Unterstützung zu versorgen, sodass sie ihre Verpflichtungen mit voller Motivation erfüllen können.

„The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."
Der bevorzugte Kommunikationsweg zum Informations- und Gedankenaustausch innerhalb des Entwicklungsteams soll in Form von face-to-face Gesprächen erfolgen. Diese Art der direkten und effizienten Kommunikation kann der Entstehung von Missverständnissen vorbeugen, sodass das Vertrauen zwischen den einzelnen Teammitgliedern gestärkt wird. Die Informationsbeschaffung aus erster Quelle ist stets zu bevorzugen, da durch mangelnde Kommunikation und schlechtem Informationsfluss bedingte Missverständnisse die Motivation des gesamten Entwicklungsteams erschöpfen können und deren Beseitigung zu unnötigem Zeitverlust führt.

„Working software is the primary measure of progress."
In der agilen Entwicklung wird die funktionsfähige Software als wichtigster Indikator für den Fortschritt des Projekts angesehen. Die Dokumentation der Ergebnisse wird in den Hintergrund gedrängt und spielt eine zweitrangige Rolle, da KundInnen schon seit Anbeginn des Projekts mit funktionsfähiger Software beliefert werden und ständig mit dem Entwicklungsteam in Kontakt stehen, sodass sie stets auf dem neuesten Stand der Entwicklung sind.

„Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely."
Agile Prozesse eröffnen die Möglichkeit der kontinuierlichen Entwicklung, denn jede neue Funktionalität wird gleich im Anschluss an die Entwicklung getestet, implementiert und ausgeliefert. Diese Vorgehensweise begünstigt die Bewahrung eines konstanten Arbeitstempos, den ganzen Entwicklungszyklus hindurch. Da regelmäßig Rückmeldungen zur Software erhalten werden, minimiert sich die Wahrscheinlichkeit, dass zu einer späten Phase des Entwicklungsprozesses negative Überraschungen zum Vorschein kommen, die zu einem enormen Zeitaufwand führen würden.

„Continuous attention to technical excellence and good design enhances agility."
Die Qualität des Designs ist ein bedeutender Aspekt, um die Agilität des Projekts aufrechterhalten zu können. Das Design sollte daher als kontinuierliche Aktivität (*Refactoring*) ständig optimiert werden, die das ganze Projekt hindurch stattfindet. Daher sind EntwicklerInnen dazu verpflichtet, sauberen Code auf höchster Qualität zu schreiben und Defekte am Design augenblicklich zu beheben. Die hohe Qualität des Designs ist somit der Schlüssel zu hoher Geschwindigkeit und fördert die Flexibilität, sowie Langlebigkeit der Software.

„Simplicity - the art of maximizing the amount of work not done - is essential."
Dieses Prinzip betont, dass die Einfachheit des gesamten Entwicklungsprozesses essentiell ist und mit minimalem Aufwand optimale Ergebnisse erzielt werden können. Um diese Strategie effektiv umsetzen zu können, ist es notwendig, sich auf aktuelle Probleme in der Entwicklung zu konzentrieren und sich deren Lösung zu widmen, anstatt sich mit möglichen zukünftigen Problemen zu beschäftigen. Daher sollten gegenwärtige Anforderungen zuerst erfüllt werden, um die besten Ergebnisse herausholen zu können.

„The best architectures, requirements, and designs emerge from self-organizing teams."
Agile Teams sind selbstorganisiert, sodass Aufgaben und Verantwortungen untereinander aufgeteilt werden. Mittels offenen Teamgesprächen, enger Zusammenarbeit und kontinuierlichem Wissensaustausch, können Design und Architektur der Software optimiert und Anforderungen bestmöglich erfüllt werden. Eine solche Arbeitsatmosphäre fördert die Qualität der zwischenmenschlichen Beziehungen, sodass sich Teammitglieder gegenseitig respektieren und unterstützen können.

„At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."
Agile Teams sollten in regelmäßigen Zeitabständen ihre Organisationsstrategie und Arbeitsmethoden überdenken und aktualisieren, da in ihrem Umfeld ständig Änderungen stattfinden und somit die Anpassung an diese Bedingungen erforderlich ist. Um die Agilität des Projekts zu bewahren und effektiver arbeiten zu können, ist kontinuierliches Lernen und Beobachten eine wichtige Voraussetzung.

6.1.1 Agile Development und Build-Prozess

In der vorangehenden Beschreibung der 12 Prinzipien des agilen Manifests wurde ausführlich und einleuchtend dargestellt, dass Agilität auf einer intensiven Zusammenarbeit zwischen EntwicklerInnen und KundInnen beruht. Nach Berczuk, Appleton und Konieczka müssen KundInnen von der Anfangsphase bis zum Ende des Projekts regelmäßig mit lauffähigen Softwareteilen versorgt werden, wodurch für EntwicklerInnen die Notwendigkeit entsteht, sich frühzeitig, bereits in der Anfangsphase des Projekts, mit der Erstellung von Build Skripten und eines Build-Prozesses zu befassen (2003).

Der Build Prozess spielt eine besonders bedeutende Rolle in der agilen Entwicklung, denn er vollbringt die Überwachung des Prozesses, sodass Fehler bereits frühzeitig entdeckt und augenblicklich behoben werden können, damit kein großer Zeitverlust entsteht. Das Konzept der Agilität fordert die Ausführung des Build-Prozesses in regelmäßigen und kurzgehaltenen Zeitabständen, wobei es insofern auch wünschenswert ist, die Build-Zeit möglichst kurz zu halten. Nach Lee sollte die durchschnittliche Build-Zeit unter 10

Minuten liegen (2008). Hinsichtlich der Reduzierung des Zeitaufwands werden von Berczuk, Appleton und Konieczka bestimmte Maßnahmen vorgeschlagen (2003):

- Durch Zuhilfenahme von mehr Hardware [9] können langandauernde Builds verkürzt werden.
- Zwischenzeitliche Targets können im Repository abgelegt werden, um die Build-Zeit zu reduzieren.
- Bei der Auswahl des Compilers und Linkers für das Projekt sollte die mögliche Dauer der Compiler- und Linker-Zeiten berücksichtigt werden, sodass die zeitsparendste Alternative gewählt wird.
- Das Projekt kann in mehrere unabhängige und build-fähige Module aufgespalten werden, die *run-time dependencies* erzeugen.

In der agilen Entwicklung ist neben dem Zeitaspekt auch der Build-Architektur Aufmerksamkeit zu schenken, da sie die Qualität des Build-Prozesses mitbestimmt und notwendigerweise die Reproduzierbarkeit und Testfähigkeit von kleineren Einheiten, sowie die Ausführung von schnellen bzw. parallelen Builds unterstützen sollte (Berczuk et al. 2003).

Die Automatisierung des Build-Prozesses kann am Besten auf der Grundlage von ausführbarem Wissen stattfinden und sollte dokumentiertem Wissen vorgezogen werden, denn die Beschreibung des Build-Prozesses mittels Skripten ist besser verständlich und reproduzierbar. D.h. an dieser Stelle ist es wichtig eine *doppelte Dokumentation* des Prozesses zu vermeiden, der ohnehin unbrauchbar und zeitraubend wäre. *Doppelte Dokumentation*en begegnen uns in manuellen Prozessen, denen oftmals zusätzliche schriftliche Berichte hinzugefügt werden, die aber in den meisten Fällen rasch veralten können. Folglich wird es für den/die Build-ManagerIn zu einer schwierigen, wenn nicht gar unmöglichen Aufgabe, auf Grundlage dieser veralteten Dokumentationen den Build-Prozess korrekt auszuführen. Bei automatisierten Build-Prozessen wird dieses Problem automatisch umgangen, da bei der Erstellung eines installierbaren Release die Dokumentation des Prozesses mit ausgeliefert wird, sodass ein zusätzliches schriftliches Dokumentationsverfahren nicht von Nöten ist (Berczuk et al. 2003). In der agilen Entwicklung sollte der Build-Prozess zudem die kollektive Erreichbarkeit der Build Skripte sicherstellen, sodass alle Mitwirkenden im Entwicklungsteam Zugang zum Code haben und gegebenenfalls auch Änderungen an diesem durchführen können. Außerdem sollte jede/jeder MitarbeiterIn die Möglichkeit besitzen, den Build auszuführen bzw. diesen für sich ausführen zu lassen (Lee 2008). Zusammenfassend kann festgehalten werden, dass agile Projekte einen konsistenten, automatisierten, reproduzierbaren, schnellen und wiederholbaren Build-Prozess verlangen.

[9] Vgl. Skalierbarkeit unter Kapitel 2.3.4

6.2 Open Source Development

Der Begriff *Open Source* wurde im Jahr 1998 von den Mitgliedern der Open Source Initiative (OSI), allen voran Raymond, eingeführt. Das Konzept der Open Source Entwicklung lehnt sich jedoch stark an jene der *Free Software* [10] an und stimmt mit dieser in seinen fundamentalen Prinzipien weitgehend überein, unterscheidet sich von ihr aber auf Grund ideologischer Sichtweisen. Die neue Namensgebung ist vor allem aus marketingstrategischen Überlegungen hervorgegangen, um Open Source Software besser vermarkten zu können und sich vom Begriff *free* loszulösen, der in vielen Kreisen mit einem kostenlosen Angebot assoziiert wird, jedoch im eigentlichen Sinn für bestimmte Freiheiten in der Entwicklung steht (vgl. Wikipedia, Open Source; Grassmuck 2004). So war es zwar gewünscht, dass der neue Name auf jene Freiheiten, die den UserInnen bzw. EntwicklerInnen eingeräumt werden, hinweist, zugleich aber nicht mit kostenloser Software in Verbindung gebracht wird.

Das Konzept, welches der Open Source Entwicklung zugrunde liegt, ist der freie Zugang zum Quelltext der Software. D.h. sie ist quelloffen und kann somit von jedem/jeder UserIn eingesehen, modifiziert und verbreitet werden. In der *Open Source Definition* (OSD) werden charakteristische Merkmale festgehalten, die erfüllt werden müssen, damit eine Software als Open Source gelten kann (opensource.org, The Open Source Definition):

- Free Redistribution: Niemand darf in seinem Recht eingeschränkt werden die Software als Teil eines Softwarepaketes, in Kombination mit anderen Programmen, zu verschenken oder zu verkaufen. Im Falle eines Verkaufs dürfen keine Lizenz- oder sonstigen Gebühren verlangt werden.

- Source Code: Das Programm muss den Quellcode bzw. eine kompilierte Form des Codes beinhalten. Wird der Quellcode nicht mitgeliefert, so muss eine Möglichkeit angeboten werden, diesen als gebührenfreien Download aus dem Internet zu beschaffen. Unverständlich geschriebener Code und Zwischenformen des Codes, die von einem Präprozessor oder Konverter erzeugt wurden, sind nicht gestattet.

- Derived Works: Veränderungen und Derivate an der Software müssen erlaubt sein. Außerdem muss die abgeleitete Software unter denselben Lizenzbestimmungen weitergegeben werden können, wie die ursprüngliche Software.

- Integrity of The Author's Source Code: Die Möglichkeit, den Quellcode in veränderter Form weiterzugeben, darf nur dann eingeschränkt werden, wenn zusammen mit dem Quellcode sog. *Patch files* weitergegeben werden dürfen, die den Programmcode bei der Kompilierung verändern. Die Weitergabe von Software, die durch Abänderung des

[10] Der Begriff und das Konzept von *Free Software* ist seitens Stallmann begründet worden.

Quellcodes entstanden ist, muss erlaubt sein. Es kann lediglich verlangt werden, dass abgeleitete Programme einen anderen Namen bzw. eine andere Versionsnummer als die Ausgangssoftware tragen.

- No Discrimination Against Persons or Groups: Bei der Weitergabe der Lizenz dürfen keinerlei Personen bzw. Gruppen diskriminiert oder benachteiligt werden.

- No Discrimination Against Fields of Endeavor: Das Einsatzfeld der Software darf nicht eingeschränkt werden, sodass sie in jedem denkbaren Bereich eingesetzt werden kann.

- Distribution of License: Alle Personen, welche die Software bekommen, müssen zugleich auch alle Rechte am Programm erhalten, ohne sich eine Lizenz erwerben zu müssen.

- License Must Not Be Specific to a Product: Die Rechte an einem Programm dürfen nicht auf ein bestimmtes Produktpaket beschränkt sein. Wenn ein Programm aus dem Produktpaket entnommen und anschließend benutzt bzw. weitergegeben wird, sollen jene NutzerInnen, die dieses Programm erhalten haben, über dieselben Rechte verfügen, die in Verbindung mit dem originalen Produktpaket gewährt wurden.

- License Must Not Restrict Other Software: Die Lizenz darf die Weitergabe der Software gemeinsam mit anderer Software nicht einschränken und auch nicht verlangen, dass andere mitgelieferte Programme quelloffen sind.

- License Must Be Technology-Neutral: In den Lizenzregelungen dürfen keinerlei Beschränkungen bezüglich der Verwendung individueller Technologien oder einem bestimmten Interface-Stil enthalten sein.

Die Bestimmungen zur Open Source Software werden in den OSD klar dargestellt, sodass an dieser Stelle die wichtigsten Unterschiede zu proprietären Software nochmals unterstreicht werden sollen. Open Source Software kennzeichnet sich durch seine Quelloffenheit, proprietäre Software hingegen benutzt ein Blackbox-System, was bedeutet, dass der Quellcode in einer unverwertbaren vorkompilierten Form ausgeliefert wird. Der Quelltext wird somit in binärer Form vermarktet und die Software im Allgemeinen mittels Urheberrechten, Patenten, Markenschutz und Kopierschutzmaßnahmen vor Lektüre, Weitergabe und Veränderungen geschützt. Dies Widerspricht natürlich vollkommen dem Konzept der Open Source Initiative. Der Grundgedanke hinter diesen proprietären Maßnahmen ist der Schutz von geistigem Eigentum, bedeutet für EntwicklerInnen und SystemadministratorInnen jedoch eine massive Einschränkung in ihrer Arbeitsumgebung. Der Aspekt, der somit nicht berücksichtigt bzw. vernachlässigt wird, ist, dass Software- und Hardwarebedingungen einem ständigen Wandel unterworfen sind, somit rasch veralten

und anpassungswürdig werden. In diesem Fall ist es für die Qualitätserhaltung der Software sinnvoller, AnwenderInnen den Einblick in den Quellcode der Software zu gestatten, sodass sie Veränderungen durchführen oder zumindest auf wünschenswerte Änderungen hinweisen können (Grassmuck 2004). Ein kurzer Einblick in die internen organisatorischen Verhältnisse soll hier Klarheit über die Vorgehensweise bei der Open Source Entwicklung verschaffen. Die Organisation bei Open Source Projekten glieder sich in drei Ebenen: das Core Team, die MaintainerInnen und die Community.

Das Core-Team bildet dabei das zentrale Steuerungsgremium des Projekts und besteht meist aus Personen, die am längsten an der Entwicklung arbeiten oder am aktivsten sind. Die wichtigsten Entscheidungen hinsichtlich der Softwareentwicklung und des Designs fallen in Ihren Aufgabenbereich. Anzumerken ist jedoch, dass Entscheidungen dennoch auf der Grundlage von Mehrheitsbeschlüssen getroffen werden, sodass keine hierarchische Machtverteilung entsteht. Bei größeren Projekten ist oftmals, aus Gründen der Übersichtlichkeit, die Gliederung der Software in funktionale Einheiten, sog. Packages oder Module, notwendig. Für jedes dieser Module, werden ein bis mehrere zuständige MaintainerInnen bestimmt, die als AnsprechpartnerInnen für den jeweiligen Teilbereich fungieren. Die nächste Ebene stellt die Community dar, die aus einer nicht genauer festgelegten Anzahl an Mitwirkenden bestehen kann. Die Community setzt sich aus AnwenderInnen zusammen, die sich freiwillig als EntwicklerInnen für das Projekt engagieren möchten. Für die Mitglieder der Open Source Community gibt es meistens keine festgelegte Aufgabenverteilung, sodass jede Person für sich selbst entscheidet, welchen Beitrag sie zum Projekt leisten möchte (Grassmuck 2004).

Community Mitglieder können sich auf unterschiedliche Art und Weise an der Entwicklung beteiligen. Zum einen besteht die Möglichkeit mittels Übermittlung von Fehlerberichten (*Bug Reports*) auf Fehler in der Software hinzuweisen. Eine weitere Option ist die Einbringung von Ideen zur Weiterentwicklung der Software oder Vorschlägen für neue Funktionen (*Feature Requests*). Die aktivste Ebene wäre die Erstellung von Patches oder größeren Funktionserweiterungen. Diese Patches werden an die jeweiligen zuständigen MaintainerInnen geschickt, die diese sammeln und schließlich an das Core-Team übermittelt. Die von allen Seiten eintreffenden Patches werden vom Core-Team selektiert und zur Abstimmung an die Community weitergegeben, um zu bestimmen, welche Patches beim nächsten Release mit eingebunden werden sollten. In manchen Fällen wird die Entscheidung auch ohne Beteiligung der Community direkt vom Core-Team getroffen. Nach den allgemeinen Lizenzbestimmungen der Open Source Software, sollte es auch einzelnen EntwicklerInnen erlaubt sein, unabhängig von den MaintainerInnen und dem Core-Team, ein Release der Software herauszugeben und diese weiter zu verteilen. In bestimmten Fällen muss jedoch kenntlich gemacht werden, dass es sich nicht um eine offizielle Entwicklerversion handelt (vgl. Grassmuck 2004; Erenkrantz 2003).

6.2.1 Open Source Development und Build-Prozess

Da nun der Beteiligungsgrad der Community, die sich aus mehreren hundert oder tausend Mitgliedern zusammensetzen kann, am Entwicklungsprozess dargestellt wurde, ist es nur allzu verständlich, dass zur Verwaltung des Quellcodes ein effektives Source Control Management System notwendig ist. Standardmäßig wird bei Open Source Projekten ein CVS-Server eingesetzt, der einen freien Zugriff auf die Quellcodebäume der Software ermöglicht. Wenn jemand den Quellcode modifizieren möchte, so muss er sich vorher registrieren, damit er/sie für weitere EntwicklerInnen ansprechbar ist. Möchte man an den Dateien im Repository arbeiten, so ist die Ausführung des *Check-out/Check-in* Befehls [11] erforderlich. Dabei wird zur Konfliktlösung das *copy-modify-merge* Modell angewandt, sodass keine Sperrung der Quelldateien stattfindet und mehrere EntwicklerInnen gleichzeitig an einer Datei arbeiten und Änderungen vornehmen können (Grassmuck 2004).

Bei Open Source Projekten werden sog. *stable Builds* ausgeführt, wenn ein bestimmter Meilenstein in der Entwicklung erreicht wurde (Lee 2008). Dies ist dann der Fall, wenn gewisse festgelegte Ziele für die Herausgabe der nächsten Version erfüllt und ein bestimmter Grad an Stabilität erreicht wurden, sodass ein offizielles *major Release* zur Nutzung seitens UserInnen freigegeben werden kann. Wichtig ist, dass Entwicklerversionen frühzeitig und in kurzen Zeitabständen veröffentlicht und zur Weiterentwicklung an die Community weitergegeben werden (Grassmuck 2004). Daneben finden tägliche automatisierte *Development Builds* statt, sodass zuletzt seitens Community-MitarbeiterInnen getätigte Änderungen einem Build-Prozess unterzogen, getestet und released werden können (Lee 2008). Anschließend steht die Software wieder zur weiteren Entwicklung und Bearbeitung zur Verfügung. Der Grad der Änderungen lässt sich dabei an der Versionsnummer ablesen, die bei jedem Release erzeugt wird (Grassmuck 2004).

Die Richtung der Entwicklung wird vorrangig vom Core-Team bzw. den zuständigen Release-ManagerInnen bestimmt, welche die Patches bzw. neu hinzugefügte Funktionalitäten vom den MaintainerInnen übermittelt bekommen. Sie entscheiden entweder eigenständig über die aufzunehmenden Patches und Funktionalitäten, oder holen die Meinung der Community dazu ein. In weiterer Folge wird ein Release-Kandidat herausgegeben, der über eine Mailingliste an alle Community-Mitglieder verteilt wird. Diese haben nun die Aufgabe den Kandidaten auf Stabilität und Qualität via Tests zu überprüfen und Feedbacks zurückzuerstatten. Somit können vor dem eigentlichen Release entdeckte Fehler beseitigt und Aktualisierungen durchgeführt werden. Diese Art des Tests wird als *Pre-release Testing* bezeichnet (Erenkrantz 2003).

Für die Open Source Entwicklung ergeben sich somit einige Anforderungen an den Build-Prozess, wie beispielsweise das Erfordernis, täglichen automatisierte Builds und Unit Tests

[11] Siehe dazu Kapitel 4.1.3

auszuführen, um die Integrität der Änderungen überprüfen zu können. Außerdem sollten Build-Skripte für alle EntwicklerInnen erreichbar sein, damit diese in der Lage sind, den Build-Prozess bei Bedarf auch selbstständig ausführen zu können (Lee 2008).

Open Source Software ist im Grunde eine Entwicklungsform die erst durch die Geburt des Internets ermöglicht wurde, denn ohne die weltweite Vernetzung und den Onlinezugang zu Quellressourcen wäre sie nicht möglich gewesen (Grassmuck 2004). Für diese Art der Entwicklung ist es daher erforderlich, dass die Software über verschiedene Mirrors bzw. über eine Projekthomepage zum Download angeboten wird. Auf dieser projekteigenen Webseite sollten zudem der aktuelle Quellcode, Informationen zu Build-Artefakten und Dokumentationen erhältlich sein (Fogel & Bar 2001).

6.3 Traditional Enterprise Development

Die Traditionelle Enterprise Entwicklung umschließt gewisse Methoden, so z.B. den *Rational Unified Process* (RUP) und Team Software Prozess (TSP). Diese Form der Entwicklung kommt meist bei Unternehmens- oder Handelsumgebungen zur Anwendung, die sich besonders großen Softwareentwicklungsprojekten widmen. Solche Projekte erfordern eventuell eine Laufzeit über mehrere Jahre, an denen mehrere unterschiedliche Teams oder sogar verschiedene Unternehmen an der Entwicklung beteiligt sein können. Da das Traditional Enterprise Development stärker planorientiert ist, findet vor der Implementierung die Festlegung der Anforderungen für das System statt. Diese Anforderungen werden anschließend durch verschiedene Anwendungsfälle beschrieben, die den gesamten Prozess steuern. Ein besonders großer Schwerpunkt liegt auf der Qualität der Architektur und des Designs. Ein solches architekturzentriertes Modell muss die Umsetzung der Anwendungsfälle und das Zusammenspiel der einzelnen Komponenten unterstützen. Die beispielhafte Beschreibung des Traditional Enterprise Development wird im Folgenden anhand der RUP-Methode erfolgen (vgl. Zuser et al. 2004; Lee 2008; Fuchsberger & Remsperger 1999).

6.3.1 Rational Unified Process (RUP)

Beim Rational Unified Process wird eine objektorientierte Vorgehensweise verfolgt, sodass hier eine „konkrete Implementierung des Unified Process“ (Zuser et al. 2004, S. 91) stattfindet. RUP wurde ursprünglich von der Firma *Rational* entworfen und als Produkt vertrieben. Nach der Fusionierung mit dem IBM Konzern im Jahr 2002 ist es schließlich unter dem neuen Unternehmensdach weiterentwickelt worden (Wikipedia, RUP).

Bei RUP existieren zwei Dimensionen der Prozessstruktur: die dynamische und die statische Dimension. Die dynamische Dimension repräsentiert den Lebenszyklus, in dem

der Prozess verläuft. Auf der logischen Dimension hingegen findet die logische Gruppierung von zusammengehörigen Kernprozessen statt (Fuchsberger & Remsperger 1999). Die Prozessstrukturen lassen sich anhand der charakteristischen Merkmale beider Dimensionen erläutern und werden in Abb. 29 übersichtlich dargestellt.

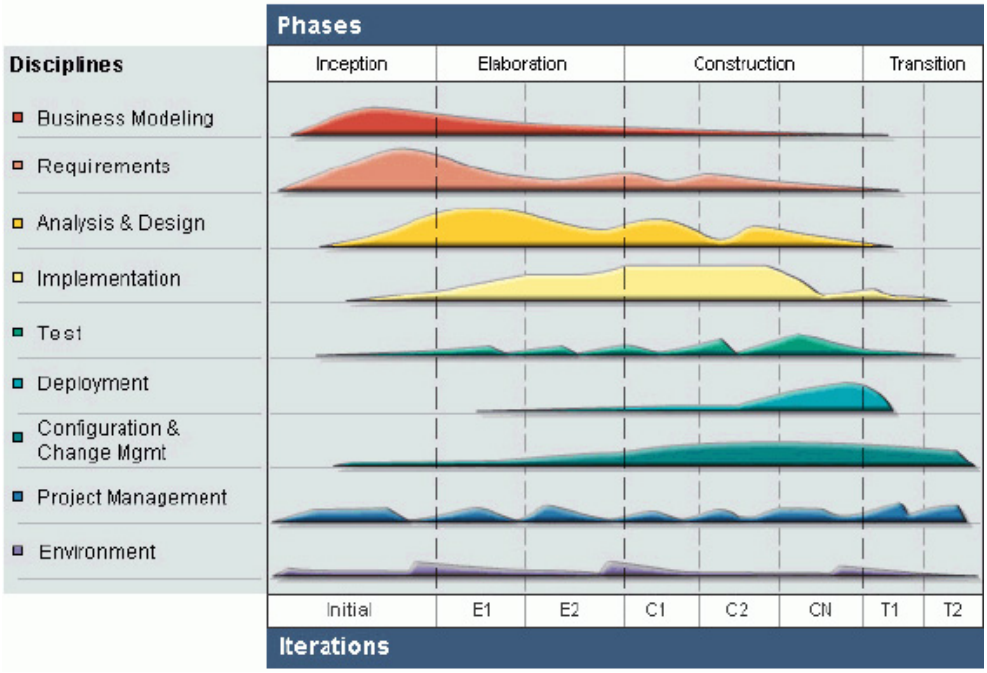


Abbildung 35: Rational Unified Process (aus Mancin et al. 2007, S. 12)

Zur dynamischen Prozessstruktur gehört die Implementierung eines iterativen Prozesses, wobei jede Iteration einem kleinen Wasserfall entspricht. Wie in Abb. 29 ersichtlich, finden innerhalb einer Iteration bestimmte Aktivitäten statt. Diese Aktivitäten gliedern sich in sechs Arbeitsschritte des *Engineering Workflows* und drei weitere unterstützende Arbeitsschritte. Die Engineering Workflows bestehen aus dem *Business Modeling*, *Requirements*, *Analysis & Design*, *Implementation*, *Test* und *Deployment*. Die unterstützenden Arbeitsschritte sind: *Configuration & Change Management*, *Project Management* und *Environment*. Jede Phase einer solchen Iteration wird mit einem Meilenstein abgeschlossen. Bei diesen iterativen Phasen (*Major Milestones*) handelt es sich um: *Inception, Elaboration, Construction* und *Transition* (Zuser et al. 2004).

Während der Business Modeling- und Requirements-Phase wird die Idee zum Produkt geboren und der Umfang des Projektes festgehalten. Diese Phase wird mit einem sog. *Life Cycle Objective Milestone* abgeschlossen, der den geplanten Lebenszyklus wiedergibt. In der anschließenden Analysis & Design-Phase findet die Planung notwendiger Aktivitäten und Ressourcen statt. Zudem werden die Eigenschaften des Produkts weiter spezifiziert und eine Architekturanalyse vollzogen, die mit einem Architektur-Meilenstein (*Life Cycle*

Architektur-Milestone) abgeschlossen wird. Die Implementierungs- und Testphase ist von besonderer Bedeutung, da hier das eigentliche Building und die Integration der verschiedenen Subsysteme vollzogen werden. Somit entsteht hier ein handfestes Produkt, wobei diese Phase mit einem sog. *Initial Operational Capability-Milesone* abgeschlossen wird. Das Deployment stellt die letzte Phase der Iteration dar, womit der gesamte Lebenszyklus abgeschlossen wird. Abschließend wird nun das Produkt an die EndbenutzerInnen ausgeliefert und mit dem sog. Release-Meilenstein (*Release-Milestone*) abgeschlossen (vgl. Zuser et al. 2004; Fuchsberger & Remsperger 1999).

Die statische Prozessstruktur dient der Beschreibung, wer etwas, wie oder wann, zu erledigen hat. Der Begriff *Worker* steht als Bezeichnung für eine Rolle, die einer einzelnen Person oder einer Gruppe zugeordnet wird. Unter einer *Aktivität* versteht man den Teil einer Aufgabe, die ein Worker erledigen muss. Diese Aktivität kann darin bestehen eine neue Information zu erzeugen oder bereits Bestehende zu manipulieren. Der Begriff *Artefakt* beschreibt Teile einer Information oder die Information selbst. Bei den Artefakten handelt es sich um die Inputs und Outputs, mit denen der Worker arbeitet. Solche Artefakte werden in anderen Prozessen als *work units* bezeichnet, deren Summe im Vergleich zum Rational Unified Process nicht als fertiges Produkt ausgeliefert werden muss. Unter dem Begriff *Workflow* wird die Abfolge aller Aktivitäten des RUP zusammengefasst. Workflows dienen der Koordinierung der Worker und ihrer Aktivitäten (Fuchsberger & Remsperger 1999).

6.3.2 RUP und Build-Prozess

Im Rational Unified Process findet der Build-Prozess erst ab der Implementierungsphase statt, wobei ab diesem Zeitpunkt täglich oder mindestens ein Mal pro Woche automatisierte Builds ausgeführt werden sollten (Zuser et al. 2004). Mittels solcher Builds wird nämlich sichergestellt, dass die Komponenten erfolgreich in die fortlaufende Entwicklung integriert werden. Für die Erstellung und Ausführung des Build-Prozesses ist die Bestimmung von zuständigen Build Engineers erforderlich, die für den gesamten Prozessablauf verantwortlich sind. Eine wichtige Rolle spielt zudem die Sicherheit und Wiederverwendbarkeit der Build-Skripte. Bei Projekten mit Mehrfachkomponenten werden Build Skripte generell über Komponenten definiert. Zur allgemeinen Zeitersparnis ist es daher empfehlenswert, dass Build-Routinen mehrfach nutzbar und wiederverwendbar gemacht werden (Lee 2008).

Im Zuge der Integration werden unterschiedliche Komponenten bzw. Subsysteme zu einem Ganzen zusammengefügt. Es ist zu betonen, dass die Integration von Iterationen inkrementell erfolgen sollte, d.h. mindestens ein Mal pro Woche durchzuführen ist (Zuser et al. 2004). Zu diesem Zweck sollte für jede neue Komponente, in regelmäßigen Zeitabständen oder zumindest nach der Erreichung bestimmter Milestones, ein *System*

Integration Build ausgeführt und anschließend der aktuelle Zustand des Systems mit einer Baseline festgehalten werden. Neben diesem Integrationsschritt sind auch funktionelle Integrationstests notwendig, um die Qualität und Funktionalität der integrierten Komponenten sicherstellen zu können (Lee 2008). Zur visuellen Darstellung der geschilderten Komponentenintegration soll an dieser Stelle auf die Abb. 30 verwiesen werden.

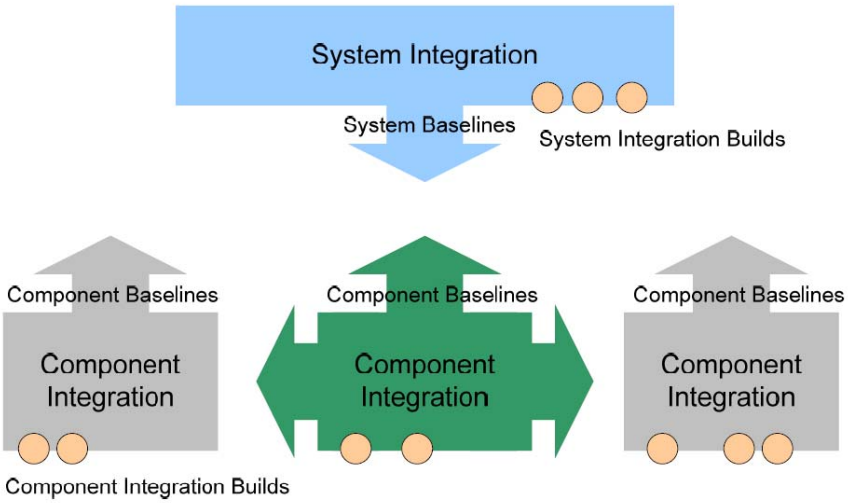


Abbildung 36: System Integration Methode (aus Lee 2008, S. 90)

In der abschließenden Deployment-Phase findet die Übergabe der fertiggestellten Software an die KundInnen statt. Die Herausgabe der Installationspakete mitsamt Supportmaterial (Benutzerhandbücher oder Wartungshinweise) oder Testergebnissen ist dabei unbedingt erforderlich (Zuser et al. 2004).

7. Zusammenfassung

Der Build-Prozess spielt eine unverzichtbare Rolle in der erfolgreichen Entwicklung und Vermarktung von Software. Die Softwareindustriewelt ist auf Grund rasanter Innovationen einem ständigen Wandel unterworfen, sodass die Umgebung, in der ein Produkt entwickelt wird, den aktuellen Bedingungen am Markt angepasst werden muss. Daher ist es umso wichtiger für Unternehmen, den wachsenden technischen und wirtschaftlichen Herausforderungen nachzukommen, um die Konkurrenzfähigkeit ihres Betriebes und ihrer Produkte steigern zu können. Diesbezüglich ist es erforderlich, bei der projektinternen Produktion von Software einen gewissen Grad an Flexibilität und Handlungsspielraum einzuräumen, sodass für die Bewältigung von möglichen Konflikten eine Hintertür offen gelassen wird. Erst dadurch kann sichergestellt werden, dass das Unternehmen schneller auf Änderungen, wechselnde Anforderungen und Wünsche der KundInnen reagieren kann.

Ein wichtiges Hilfsmittel, um einen reibungslosen und raschen Projektablauf und die Entlastung der ProjektmitarbeiterInnen zu erwirken, ist die Projektautomatisierung, d.h. das entweder eine vollständige Automatisierung des Build-Prozesses oder zumindest wichtiger Teilbereiche stattfindet. Automatisierung gewährleistet somit, dass sich das Projektteam die Auseinandersetzung mit banalen Einzelheiten ersparen kann, die nur zu einem unerwünschten Zeitverlust führen würden.

Im Rahmen dieser Diplomarbeit wurden die wesentlichen Aspekte und Voraussetzungen der erfolgreichen Projektautomatisierung und die dafür notwendige Gestaltung des Build-Prozesses ausführlich durchleuchtet. Neben einer einleitenden Darlegung der essentiellen Definitionen zu den Begriffen *Build-Prozess* und *Automatisierung*, wurden die bedeutsamen Vorteile und Eigenschaften, welche mit der Automation einhergehen, einleuchtend dargestellt. Daneben war es notwendig, am Rande auch auf den manuell-gesteuerten Build-Prozess einzugehen, um die Unterschiede beider Strategien dem/der LeserIn deutlicher vor Augen führen zu können. Um die Notwendigkeit der Projektautomatisierung verdeutlichen zu können, wurde der Build-Prozess in seinen einzelnen Bestandteilen und Schritten, die im Zuge des gesamten Prozesses durchlaufen werden, in detaillierter Form beschrieben und analysiert, sodass einwandfrei zu erkennen ist, zu welchem Zweck die Automatisierung eingesetzt wird und an welcher Stelle sie von besonders großem Nutzen ist. Eine ebenso wichtige Aufgabenstellung war die Beschreibung einzelner ausgewählter Build-Tools, welche eine tragende Rolle für die Umsetzung des Build-Prozesses spielen und die Rahmenbedingungen und das notwendige Gerüst für die Automatisierung bereitstellen. Daneben wurden die wichtigsten drei Build-Varianten beschrieben und gegenübergestellt, sodass ihre Gemeinsamkeiten bzw. Unterschiede herausgearbeitet werden konnten. Zum Abschluss der Untersuchung wurden die wichtigsten Developmentmethoden der letzten Jahre aufgegriffen und deren grundlegende Prinzipien beschrieben, sodass abschließend analysiert wurde, wie sich die einzelnen Entwicklungsmethoden auf den Build-Prozess auswirken.

Die Analyse der einzelnen Themenbereiche hat gezeigt, dass die Automatisierung des Build-Prozesses in jedem Fall für die reibungslose und qualitätsvolle Entwicklung von Softwareprodukten dringend anzuraten ist. Die Aufgaben und Schritte, die im Zuge des Build-Prozesses durchlaufen werden, sind bei jedem Projekt im Wesentlichen dieselben. Insofern wird das Projektteam stets mit denselben oder ähnlichen Aufgabenstellungen und Tätigkeiten konfrontiert sein, die im Zuge der Softwareentwicklung mehrmals und in gleicher Form wiederholt werden müssen. Daher ist es besonders zu empfehlen, jene Prozessschritte, die mehr als ein Mal wiederholt werden, zu standardisieren und automatisieren, sodass alle ProjektmitarbeiterInnen auf derselben Grundlage und mit derselben Schnelligkeit ihre Aufgaben bewältigen können. Es würde insofern keinen Sinn machen und nur zu unnötigem Zeitverlust führen, wenn diese festgelegten Aktivitäten, die immer auf dieselbe Art und Weise ablaufen, vom Projektteam manuell gesteuert und ausgeführt werden. Solche Entwicklungsszenarien verlangen, dass eine Automatisierung der genannten Prozessschritte initiiert wird, sodass durch eine einmalige Automatisierung dieser Aktivitäten der zukünftige zeitliche Aufwand und der menschliche Beteiligungsfaktor an wiederkehrenden Funktionsabläufen minimiert werden.

Um an dieser Stelle nochmals genauer auf die Ergebnisse der Diplomarbeit hinzuweisen, sollen die Erkenntnisse, die aus den verschiedenen Kapiteln dieser Untersuchung hervorgegangen sind, summarisch dargestellt werden.

Die Eigenschaften, die mit der Automatisierung des Build-Prozesses einhergehen, haben gezeigt, dass durch sie eine Reihe von vorteilhaften Dimensionen eröffnet werden, die ein manuell gesteuerter Prozess in dieser Form nicht gewährleisten könnte. Mittels Automatisierung wird nicht nur die Wahrscheinlichkeit eines Build-Zusammenbruchs verringert, sondern es wird auch die Kommunikation und Interaktion zwischen den ProjektmitarbeiterInnen auf einer gesicherten Ebene hergestellt. Auf Basis dieser Voraussetzungen ist es dem Projektteam möglich, bei der Entwicklung des Softwareproduktes mit sicheren und stabilen Schritten voranzuschreiten und auf eventuelle Krisenszenarien vorbereitet zu sein, für deren Bewältigung auch genügend Zeit eingeräumt werden kann. Ein vorab gut definierter und automatisierter Build-Prozess minimiert das Risiko für das spätzeitige Auftreten von Fehlern, da die Software schon nach Projektstart auf automatischem Wege und in regelmäßigen Intervallen auf Stabilität und Korrektheit überprüft wird. Die geschilderten Faktoren begünstigen, dass sowohl die Qualität des Softwareentwicklungsprozesses selbst, als auch des ausgelieferten Produkts in summa gesteigert werden können.

Eine dieser Eigenschaften ist, dass automatisierte Builds stets auf Vollständigkeit und Korrektheit ausgerichtet sind, sodass in diesem Zusammenhang jegliche menschliche Intervention, die zu einem Fehler führen könnte, untersagt ist. Das eigentliche Ziel der Projektautomatisierung ist es, die Eigenschaft der Wiederholbarkeit und in weiterer Folge der Reproduzierbarkeit des Build-Prozesses zu gewährleisten. Somit ergibt sich der

positive Effekt, dass mittels Wiederholbarkeit jeder Schritt, der zur Erstellung eines Produkts notwendig ist, jederzeit wiederholt werden kann, wobei die Reproduzierbarkeit zusätzlich einen rückläufigen Eingriff auf den Build-Prozess ermöglicht. Unter diesen Voraussetzungen kann die Stabilität und Qualität des Softwareprodukts sichergestellt werden, da stets die Möglichkeit besteht, einen fehlerhaften Schritt nochmals zu wiederholen und zu korrigieren oder bei Bedarf auf eine frühere stabile Version der Software zurückzugreifen. Automatisierte Builds müssen zudem stets informativ sein, d.h das alle Prozessschritte automatisch erfasst und dokumentiert werden, sodass die notwendigen Informationen jederzeit vom Projektteam eingesehen werden können. Dies hat zur Folge, dass Fehler rascher aufgespürt und beseitigt werden können und somit die Funktionalität des Prozesses bewahrt wird. Abhängig vom Projektablauf und der herangezogenen Entwicklungsmethode, kann es notwendig sein, den Build-Prozess zu unterschiedlichen Zeitintervallen oder Anlässen auszuführen. Deshalb ist es äußerst wichtig, dass die zeitliche Steuerung der Builds an den jeweiligen Projektrhythmus angepasst wird. Zu diesem Zweck bieten sich die verschiedenen Automatisierungstypen an, die eine ereignisgesteuerte oder zeitgesteuerte Ausführung des Build-Prozesses veranlassen. Zusätzlich ist es bei Notwendigkeit auch noch möglich, beide Automatisierungstypen in Kombination zum Einsatz zu bringen. Zur vollen Ausschöpfung dieser zusammengefassten Vorteile ist es zudem ratsam, für den Build-Prozess einen eigenen Server einzurichten, wobei stets darauf zu achten sein sollte, dass der Build-Server stets portabel bleibt. Zuletzt bleiben eine Reihe weiterer Eigenschaften zu erwähnen, die durch die Projektautomatisierung entstehen. Dazu gehören z.B. die Skalierbarkeit, Integrierbarkeit, Wiederverwendbarkeit und Erreichbarkeit des Build-Prozesses. Alle genannten Eigenschaften sprechen eindeutig für die Automatisierung des Build-Prozesses.

Anhand ausgewählter Build-Tools, darunter Make, Ant und Maven, wurde exemplarisch dargelegt, welch tragende Rolle diese Werkzeuge in der Umsetzung des Build-Prozesses spielen. Im Bezug auf diese Tools, die als unterstützendes Framework zur Gestaltung und Ausführung des Build-Prozesses herangezogen werden, ist jedoch anzumerken, dass sie zwar als wichtiges Hilfsmittel den Umgang mit dem Build-Prozess erleichtern und für die Projektautomatisierung dienlich sein können, aber ebenso die Gefahr besteht, dass spezifische Eigenschaften der Tools zum Hindernis werden, sodass gewisse Aufgaben nicht wie gewünscht bewältigt werden können. An dieser Stelle wären die Nachteile des Tools Make anzusprechen, das seinerzeit für C und C++ Projekte zwar ein beliebtes Werkzeug gewesen ist, aber gegenwärtig anderen Tools in wichtigen Aspekten nachhinkt. Das wichtigste Problem mit Make ist, dass die Befehle mit Shell-Skripten ausgeführt werden. Folglich ist der Build-Prozess nicht nur stark von der Systemumgebung abhängig, sondern auch unportabel. Daneben zeigt sich bei der Verwendung von Make in größeren Softwareprojekten eine deutliche Verlangsamung des Prozesses. Ebenso ist die Dokumentation des Build-Prozesses ein weiterer Problembereich, wobei noch hinzukommt, dass die Organisation der Verzeichnisstruktur mit dem Wachsen des Verzeichnisbaums immer schwieriger wird und zu Instabilität führt.

Aus diesen Gründen ist bei der Auswahl des passenden Build-Tools immer mit besonderer Sorgfalt vorzugehen und abzuwägen, ob das ausgewählte Werkzeug den Anforderungen des Projekts nachkommt. Diesbezüglich ist die im Projekt verwendete Programmiersprache von besonderer Bedeutung bei der Auswahl des geeigneten Werkzeugs, da Build-Tools oftmals an eine bestimmte Programmiersprache gebunden sind. Jedoch bieten einige Werkzeuge, wie Ant und Maven, auch Lösungen bzw. Alternativen an, sodass ihr Einsatz auch für andere Programmiersprachen ermöglicht wird. Das Build-Tool Ant zeichnet sich im Gegensatz zu Make durch seine Plattformunabhängigkeit aus, welche durch die Java-Technologie und die verwendete Skriptsprache XML bedingt ist. Obwohl Ant für den Einsatz bei Projekten mit unterschiedlichen Programmiersprachen bzw. Technologien geeignet ist, werden zusätzlich auch Alternativen, wie NAnt für .NET-Projekte, angeboten. Unter den untersuchten Tools ist Maven jenes, das vergleichsweise die meisten Optionen und Dienste bereithält. Bei Maven handelt es sich um ein modelbasiertes Tool, das unter anderem als *Build Management Framework* bezeichnet wird. Die Neuerung, die mit diesem Werkzeug einhergeht, ist die Nutzung eines deklarativen Ansatzes über das *Project Object Model* (POM), welches alle notwendigen Informationen zur Erstellung eines Produkts bereitstellt. So gibt es in Maven auch einen einheitlichen Build Lebenszyklus und eine standardisierte Verzeichnisstruktur, die sich zudem sehr gut für die einfache Projektautomatisierung eignen. Bei Maven wird das Problem der Programmiersprachen gelöst, ohne dass eine neue Implementierung durchgeführt werden muss. Da die eigentliche Funktionalität von Maven über Plugins hergestellt wird, kann es mit einem solchen Plugin problemlos erweitert und an die gewünschte Programmiersprache angepasst werden. Abschließend bleibt noch anzumerken, dass durch den Einsatz der zahlreichen angebotenen Tools die Handhabung des Build-Prozesses wesentlich erleichtert werden kann, wobei die künftige Qualitäts- und Funktionsverbesserung dieser Tools mit einer weiteren Qualitätserhöhung des Build-Prozesses und folglich der erstellten Produkte einhergehen wird.

Die Gegenüberstellung der drei Build-Varianten Entwickler-, Integrations- und Release-Build hat gezeigt, dass technisch gesehen keine großen Unterschiede zwischen ihnen festzustellen sind, sodass sie sich lediglich dadurch unterscheiden, zu welchem Anlass, in welcher Häufigkeit und von wem bzw. für wen sie ausgeführt werden. Der Entwickler-Build ist dabei zweifellos jene Variante, die im Zuge des Softwareentwicklungsprozesses am häufigsten zum Einsatz kommt. Dieser wird nämlich von EntwicklerInnen bei der Generierung oder Änderung des Quellcodes auf der lokalen Arbeitsumgebung ausgeführt, um bereits kleinste Änderungen oder Erweiterungen am Quellcode auf Konsistenz und Korrektheit zu überprüfen. Der Entwickler-Build kann dabei als vollständiger, oder, aus zeitlichen Gründen, als inkrementeller Build ausgeführt werden. Als nächstes folgt der Integrations-Build, der zwar seltener aber dennoch in regelmäßigen Zeitintervallen ausgeführt wird. Das Ergebnis des Integrations-Builds richtet sich an das gesamte Projektteam, da er alle existierenden Quellcodes zu einem einzigen lauffähigen System integriert. Der Release-Build hingegen wird am seltensten und nur dann ausgeführt, wenn

das Produkt zur Auslieferung an KundInnen freigegeben wird. Alle drei Build-Varianten können bzw. sollten beim Eintreten eines festgelegten Ereignisses oder, je nach Projektrhythmus, in kontinuierlichen und regelmäßigen Zeitintervallen auf automatisierte Art und Weise ausgeführt werden.

Der abschließende Teil der Diplomarbeit wurde den Entwicklungsmethoden Agile Development, Open Source Development und Traditional Enterprise Development (mit Schwerpunkt auf dem Rational Unified Process) eingeräumt, die auch den Abschluss der Zusammenfassung bilden sollen. So wurden die Auswirkungen der einzelnen Developmentmethoden auf den Build-Prozess untersucht und dargestellt, welche Anforderungen Builds diesbezüglich zu erfüllen haben. Der wichtigste Aspekt, der all diesen Developmentmethoden zu Grunde liegt ist, dass sie die Automatisierung des Build-Prozesses sehr stark befürworten. Agile Softwareprozesse beruhen auf dem Prinzip, dass KundInnen bereits in einer sehr frühen Projektphase mit lauffähigen Softwarteilen beliefert werden, sodass in diesem Zusammenhang der Build-Prozess und seine Automatisierung eine enorme Bedeutung trägt. In diesem Szenario muss der Build-Prozess nicht nur frühzeitig definiert und implementiert, sondern auch in regelmäßigen kurzen Zeitintervallen ausgeführt werden, um den Prozess bestmöglich überwachen und Fehler frühzeitig entdecken zu können.

Bedingt durch die Open Source Philosophie und die Anzahl der Community Mitglieder sowie aktiven EntwicklerInnen ist es unbedingt erforderlich, dass für die Organisation des Quellcodes ein effektives Source Control Management System verwendet wird. Da EntwicklerInnen unabhängig voneinander Änderungen oder Erweiterungen am Quellcode durchführen können, sind tägliche automatisierte Builds unverzichtbar, um die Integrität der Änderungen sicherzustellen. Die Build-Skripte müssen den EntwicklerInnen zudem immer zur Verfügung gestellt werden, damit diese den Build-Prozess bei Bedarf auch selbstständig ausführen können. Da bei Open Source Projekten der Quellcode frei zugänglich sein muss, hat in diesem Zusammenhang auch die Dokumentation, Veröffentlichung und Erreichbarkeit einen wichtigen Stellenwert.

Auf das Traditional Enterprise Development wurde am Beispiel des Rational Unified Process (RUP) genauer eingegangen, bei dem eine komponentenbasierte Entwicklung stattfindet und der Build-Prozess erst mit der Implementierungsphase einsetzt. Ab diesem Zeitpunkt ist es jedoch wichtig, dass Builds täglich oder mindestens ein Mal pro Woche ausgeführt und automatisiert werden. Besonders die regelmäßige Ausführung von Integrations-Builds ist für die Integrierung der erstellten Komponenten in die fortlaufende Entwicklung notwendig. Die Komponentenintegration sollte mitsamt Integrationstest mindesten ein Mal pro Woche, wenn auch nur inkrementell, stattfinden. Für den Rational Unified Process ist es wichtig, dass der Build-Prozess jederzeit wiederholbar sein sollte, sodass in jedem Fall die Automatisierung der Builds zu veranlassen ist.

Literaturverzeichnis

agilemanifesto.org. http://agilemanifesto.org/(Zugriff am 10. 09 2008).

Aho, Alfred, Ravi Sethi, und Jeffrey Ullman. *Compilerbau.* Bonn: Addison-Wesley, 1999.

Amelingmeyer, Jenny, und Günther Specht. *Technologiemanagement & Marketing: Herausforderungen eines integrierten Innovationsmanagements.* Wiesbaden: Deutscher Universitäts-Verlag, 2005.

ANSI/IEEE Standard 1042 IEEE Guide to Software Configuration Management. 1982.

ANSI/IEEE Standard 1061 IEEE Standard for a Software Quality Metrics Methodology . 1998.

ANSI/IEEE Standard 610-12 Standard Glossary of Software Engineering Terminology. 1990.

„ant.apache.org.“ *Introduction.* http://ant.apache.org/manual/intro.html (Zugriff am 20. 06 2008).

„ant.apache.org.“ *Property.* http://ant.apache.org/manual/intro.html (Zugriff am 20. 06 2008).

Appleton, Brad, Stephen Berczuk, Ralph Cabrera, und Robert Orenstein. „Streamed Lines: Branching Patterns for Parallel Software Development.“ *Conference on Pattern Languages of Programs.* 1998.

Balzert, Helmut. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Heidelberg: Spektrum Akademischer Verlag, 1997.

Basili, Victor, Gianluigi Caldiera, und Dieter Rombach. „Goal Question Metric Paradigm.“ *Encyclopedia of Software Engineering*, 1994: 527–532.

Berczuk, Stephen, und Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Boston: Addison Wesley, 2002.

Berczuk, Stephen, Brad Appleton, und Steve Konieczka. „Agile SCM–Build Management for anAgile Team.“ *CM Crossroads the Configuration Management Community*, 2003.

Blanton, Sean. „mil-embedded.com.“ *Using version and build control plug-ins for eclipse.* 2006. http://www.mil-embedded.com/articles/authors/blanton/ (Zugriff am 22. 05 2008).

Böhmer, Matthias. „m-boehmer.de.“ *Metrik basierte Technologie- und Prozessbewertung.*

2006. http://www.m-boehmer.de/pdf/Metrik_basierte_Technologie-_und_Prozessbewertung.pdf (Zugriff am 06. 05 2005).

Breu, Ruth, Thomas Matzner, und Friedericke Nickl. *Software-Engineering: ObjektorientierteTechniken, Methoden und Prozesse in der Praxis.* Oldenbourg: München, Wien, 2005.

Buchsein, Ralf, Victor Frank, Günther Holger, und Volker Machmeier. *IT-Management mit ITIL® V3: Strategien, Kennzahlen, Umsetzung.* Wiesbaden: Vieweg + Teubner Verlag, 2007.

Clark, Mike. *Pragmatisch Programmieren - Projekt-Automatisierung.* München, Wien: Hanser, 2006.

Collins-Sussman, Ben, Brian Fitzpatrick, und Michael Pilato. *Version Control with Subversion: For Subversion 1.5.* California: O'Reilly., 2008.

Conway, Melvin. „How do Committees Invent?“ *Datamation*, 1968: 28-31.

Cusumano, Michael, und Richard Selby. *Microsoft Secrets: How the World's Most Powerful*

Software Company Creates Technology, Shapes Markets, and Manages People. New York: The Free Press, 1995.

Cymermann, Michael. „Automate your build process using Java and Ant: Introducing the powerful XML-based scripting tool, Ant.“ 2000. http://www.javaworld.com/javaworld/jw-10-2000/jw-1020-ant.html.

Dearle, Alan. „Software Deployment, Past, Present and Future.“ *International Conference onSoftware Engineering , 2007 Future of Software Engineering.* Washington, DC: IEEE Computer Society, 2007. 269-284.

DeMarco, Tom. *Controlling Software Projects.* New York: Yourdon, 1982.

Denny, Mitch. „notgartner.wordpress.com.“ *TFVC: Concurrent Development with Branching.*2006. http://notgartner.wordpress.com/2006/01/25/tfvc-concurrent-development-with-branching/ (Zugriff am 21. 09 2008).

Dumke, Reiner R. *Softwaremetrie - Grundlagen und Ziele.* Preprint Nr. 9, Magdeburg: Fakultät für Informatik, Universität Magdeburg, 1997.

Erenkrantz, Justin R. „Release Management Within Open Source Projects.“ *Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering.* ACM, 2003. 51-55.

Fleischer, Andre. „Metriken im Praktischen Einsatz.“ *Objektspektrum*, 2007: 58-63.

Fogel, Karl, und Moshe Bar. *Open Source-Projekte mit CVS: Verteilte Softwareentwicklung mit dem Concurrent-Versions-System.* Bonn: mitp, 2002.

Fowler, Martin. „martinfowler.com." *Continuous Integration.* 2006. http://martinfowler.com/articles/continuousIntegration.html (Zugriff am 16. 09 2008).

Franz, Klaus. *Handbuch zum Testen von Web-Applikationen: Testverfahren,Werkzeuge, Praxistipps.* Berlin: Springer, 2008.

Fuchsberger, Stefan, und Andreas Remsperger. „The Rational Unified Process." 1999.

Gerlich, Rainer, und Ralf Gerlich. *111 Thesen zur erfolgreichen Softwareentwicklung: Argumente und Entscheidungshilfen für Manager, Konzepte und Anleitungen für Praktiker, mit 9 Tabellen.* Berlin: Springer, 2005.

Gilb, Tom, und Dorothy Graham. *Software Inspection: An Effective Method for Software Project Man.* Wokingham: Addison-Wesley, 1993.

Glinz, Martin. *Skriptum zur Vorlesung Einführung in Software Engineering.* Zürich: Universität Zürich, 2005.

Grassmuck, Volker. *Freie Software: zwischen Privat- und Gemeineigentum.* Bonn: Bundeszentrale für Politische Bildung, 2004.

Hatcher, Erik, und Steve Loughran. *Java-Entwicklung mit Ant.* Bonn: mitp, 2004.

Highsmith, Jim. „agilemanifesto.org." *History: The Agile Manifesto.* 2001. http://agilemanifesto.org/history.html (Zugriff am 01. 09 2008).

Hoffmann, Dirk. *Software-Qualität.* Berlin: Springer, 2008.

Holzmann, Clemens. „ssw.uni-linz.ac.at." *Metriken.* 2003. http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2002/reports/Holzmann/#6.1 (Zugriff am 29. 09 2008).

Holzner, Steve. *Ant : the definitive guide.* Sebastopol, CA: O'Reilly, 2005.

Hunt, Andrew, und David Thomas. *Der Pragmatische Programmierer.* München: Hanser, 2003.

Hunt, Andrew, und Thomas David. *Unit-Tests mit JUnit.* München, Wien: Hanser, 2004.

Hüttermann, Michael. *Agile Java-Entwicklung in der Praxis: von der Methode bis zur technischen Infrastruktur.* Köln: O'Reilly, 2008.

„it-agile.de." *Automatisierter Build-Prozess.* http://www.it-agile.de/build.html (Zugriff am 16. 05 2008).

Kastens, Uwe. *Übersetzerbau.* München, Wien: Oldenbourg, 1990.

Kepser, Stephan. „A Simple Proof for the Turing-Completeness of XSLT and XQuery.“ *Extreme Markup Languages.* Montréal, Québec, 2004.

Klima, Robert, und Siegfried Selberherr. *Programmieren in C.* Wien: Springer, 2003.

Koomen, Tim, und Martin Pol. *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing.* Harlow, Reading: Addison-Wesley, 1999.

Lasse, Koskela. „javaranch.com.“ *Accenture Technology Solutions.* 2004. http://www.javaranch.com/journal/200409/Journal200409.jsp#a2 (Zugriff am 15. 09 2008).

Lee, Kevin. *IBM Rational ClearCase, Ant, and CruiseControl: The Java Developer's Guide to Accelerating and Automating the Build Process.* Indianapolis: IBM Press, 2006.

Lee, Kevin. *The Buildmeister's Guide: Achieving Agile Software Delivery.* United States: Lulu Press, 2008.

Link, Johannes, Frank Adler, Achim Bangert, Ekard Burger, Peter Fröhlich, und Ilja Preuß. *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung.* Heidelberg: dpunk, 2005.

Mancin, Enrico, Cécile Péraire, Angelo Fernandes, Mike Edwards, und Kathy Carroll. *redbooks:The IBM Rational Unified Process for System z.* NY: IBM, 2007.

Massol, Vincent, Jason van Zyl, John Casey, Brett Porter, und Carlos Sanchez. *Better Builds with Maven: The How-to Guide for Maven 2.0.* United States: Mergere Library Press, 2008.

Matzke, Bernd. *Ant: eine praktische Einführung in das Java-Build-Tool.* Heidelberg: dpunkt, 2005.

„maven.apache.org.“ *Introduction to the POM.* http://maven.apache.org/guides/introduction/introduction-to-the-pom.html (Zugriff am 10. 07 2008).

„maven.apache.org.“ *Introduction to Maven 2.0 Plugin Development.* http://maven.apache.org/guides/introduction/introduction-to-plugins.html (Zugriff am 11. 07 2008).

„maven.apache.org.“ *Introduction to Repositories.* http://maven.apache.org/guides/introduction/introduction-to-repositories.html (Zugriff am 03. 07 2008).

McCarthy, Jim. *Dynamic of Software Develepment.* Redmond, WA: Microsoft Press, 1995.
Mecklenburg, Robert. *GNU make.* Köln: O'Reilly, 2005.

Meyer, Harald. „rz.uni-karlsruhe.de.“ *Kochrezepte für den automatischen Software-Build mit Cruise Control.* 2005. http://www.rz.uni-karlsruhe.de/~Harald.Meyer/QACC-Staging/QACC-Howto-HEAD/docs/docbookhtmlout/index1.html (Zugriff am 22. 08 2008).

Miller, Peter. „Recursive Make Considered Harmful.“ *AUUGN Journal of AUUG,* 1998: 14-25.

Morsy, Hussein, und Tanja Otto. *Ruby on Rails 2: das Entwickler-Handbuch.* Bonn: Galileo Press, 2008.

„msdn.microsoft.com.“ *Reguläre Ausdrücke von .NET Framework.* http://msdn.microsoft.com/de-de/library/hs600312.aspx (Zugriff am 04. 06 2008).

Mücke, Florian, und Adalbert Ochotta. *Systeminstallation und Softwareverteilung.* Augsburg:Universität Augsburg, 2006.

Nabrzyski, Jarek. *Methodology for Quality Assurance: GridLab Project Management.* Poznan: Institute of Bioorganic Chemistry PAS, 2002.

Neagu, Adrian. „freshmeat.net.“ *What is Wrong with Make?* 2005. http://freshmeat.net/articles/view/1702/ (Zugriff am 03. 06 2008).

O'Brien, Tim, John Casey, Fox Brian, Bruce Snyder, und Jason van Zyl. *Maven: The Definitive Guide.* Palo Alto, CA: Sonatype, 2008.

„opensource.org.“ *The Open Source Definition.* http://opensource.org/docs/osd (Zugriff am 05. 09 2008).

Partl, Hubert. „boku.ac.at.“ *XML-Extensible Markup Language.* 2000. http://www.boku.ac.at/htmleinf/xmlkurz.html (Zugriff am 15. 05 2008).

Penner, Volker. *Konzepte und Praxis des Compilerbaus: eine Einführung.* Braunschweig: Vieweg, 1994.

Polzer, Leslie. *Reguläre Ausdrücke.* 2002. http://www.devmag.net/webprog/regulaere_ausdruecke.htm (Zugriff am 22. 06 2008).

Popp, Günther. *Konfigurationsmanagement mit Subversion, Ant und Maven: Praxishandbuch für Softwarearchitekten und Entwickler.* Heidelberg: dpunkt, 2006.

„pragmaticautomation.com.“ *Pragmatic Automation.* http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi (Zugriff am 18. 08 2008).

Redmond, Eric. „The Maven 2 POM demystified : The evolution of a project model.“ 2006. http://www.javaworld.com/javaworld/jw-05-2006/jw-0529-maven.html.

Richardson, Jared, und William Gwaltney. *Ship it!: Software-Projekte erfolgreich zum Abschluss bringen.* München, Wien: Hanser, 2006.

Ruminer, Michael. „manicprogrammer.com.“ *Recommendations in SCM Branching Patterns in TFS.* 2006. http://manicprogrammer.com/cs/blogs /michaelruminer/archive/2006/09/07/88.aspx (Zugriff am 20. 09 2008).

Runeson, Per. „A Survey of Unit Testing Practices.“ *IEEE Software*, 2006: 22-29.

Schindler, Egbert. „ancoso-development.de.“ *Handbuch Prozessmanagement Teil I Prozessdefinition.2004.* http://www.ancoso-development.de /resources/documents/Prozessmanagementhandbuch%20I%20_D_.pdf.

Scumniotales, John. „scrumone.typepad.com.“ *Rhythm and Pace.* 2006. http://scrumone.typepad.com/agile_product_development/2006/06/ (Zugriff am 12. 05 2008).

Shaw, Mary, und David Garlan. *Software architecture: perspectives on an emerging discipline.* Upper Saddle River, NJ: Prentice Hall, 1996.

Sink, Eric. „ericsink.com.“ *Introduction.* 2004. http://www.ericsink.com/scm/scm_intro.html (Zugriff am 10. 09 2008).

Smart, John Ferguson. „An introduction to Maven 2 : How applied best practices can optimize the Java build process.“ 2005. http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html.

Spillner, Andreas, und Linz Tilo. *Basiswissen Softwaretest.* Heidelberg: dpunkt, 2004.

Steg, Rene. „Agile Softwareverteilung.“ *Javaspektrum*, 2004: 26- 29.

Ullenboom, Christian. *Java ist auch eine Insel : programmieren mit der Java Platform Standard Edition 6.* Bonn: Galileo Press, 2008.

Ulrich, Uwe. „Plattformübergreifendes Build-Management mit (N)Maven: Gebaut nach Plan.“ *dotnetpro*, 2008: 80-83.

van Zyl, Jason. „maven.apache.org.“ *History of Maven.* http://maven.apache.org/background/history-of-maven.html (Zugriff am 01. 08 2008).

Vance, Stephen. „Advanced SCM Branching Strategies.“ 1998. http://www.vance.com/steve/perforce/Branching_Strategies.html.

Vehns, Rainer. „codecentric.de.“ *Maven 2.0.* codecentric GmbH. 2006. http://www.codecentric.de/export/sites/www/_resources/pdf/2008_02_DRV_Jahrestagung_-_Maven_2.0.pdf (Zugriff am 15. 07 2006).

Versteegen, Gerhard, Knut Salomon, und Rainer Heinold. *Change Management bei Software Projekten.* Berlin: Springer, 2001.

Vigenschow, Uwe. *Objektorientiertes Testen und Testautomatisierung in der Praxis.* Heidelberg: dpunkt, 2005.

Vonhoegen, Helmut. *Einstieg in XML: für Entwickler und XML-Einsteiger.* Bonn: Galileo Press, 2004.

Walrad, Chuck, und Darrel Strom. „The Importance of Branching Models in SCM.“ *Computer*, 2002: 31-38.

Whittaker, James. „What is software testing? And why is it so hard?“ *IEEE Software*, 2000: 70-79.

„Wikipedia.“ *Ant.* http://de.wikipedia.org/wiki/Ant (Zugriff am 10. 06 2008).

„Wikipedia.“ *Bill of Materials.* http://en.wikipedia.org/wiki/Bill_of_materials (Zugriff am 29. 08 2008).

„Wikipedia.“ *Extensibler Markup Language.* http://de.wikipedia.org/wiki/Extensible_Markup_Language (Zugriff am 07. 06 2008).

„Wikipedia.“ *Open Source.* http://de.wikipedia.org/wiki/Open_Source (Zugriff am 07. 09 2008).

„Wikipedia.“ *Release-Engineering.* http://en.wikipedia.org/wiki/Release_engineering (Zugriff am 24. 05 2008).

„Wikipedia.“ *Rational Unified Process.* http://de.wikipedia.org/wiki/Rational_Unified_Process (Zugriff am 26. 08 2008).

„Wikipedia.“ *XSL Transformation.* http://de.wikipedia.org/wiki/XSL_Transformation (Zugriff am 19. 06 2008).

„Wikipedia.“ *Revision Control.* http://en.wikipedia.org/wiki/Revision_control (Zugriff am 15. 09 2008).

„Wikipedia.“ *McCabe-Metrik.* http://de.wikipedia.org/wiki/McCabe-Metrik (Zugriff am 21. 05 2008).

Zuser, Wolfgang, Thomas Grechenig, und Monika Köhle. *Software-Engineering: mit UML und dem Unified Process.* München: Pearson Studium, 2003.

Abbildungsverzeichnis

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder in gleicher, noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hasan Kilic

Wien, 27. Oktober 2008