# Qualitätsmanagement mit Continuous Integration
## Untersuchung anhand einer Machbarkeitsstudie in der Praxis

A feasibility study designed to look at
Quality Management with Continuous Integration

Fachhochschule Brandenburg

Betreuer: Prof. Dr. rer. nat. Gabriele Schmidt
Name: Lars K.W. Gohlke, Diplom-Inf. (FH)
Abgabetermin: Februar 2010

Eingereicht an der Fachhochschule Brandenburg
Fachbereich Informatik und Medien.

# Inhaltsverzeichnis


1 Einleitung 1

2 Kontinuierliche Integration 5
- 2.1 Agile Softwareentwicklung . . . 5
- 2.2 Praktiken . . . 11

3 Fallbeispiel 19
- 3.1 Der IST-Zustand - vor der Einführung . . . 20
- 3.2 Formulierung des SOLL-Zustandes . . . 22
- 3.3 Umsetzung des Konzeptes . . . 23
  - 3.3.1 Auswahl des Werkzeuges . . . 24
  - 3.3.2 Integration der Projekte in Hudson . . . 27
  - 3.3.3 Einbindung von Analysewerkzeugen . . . 31
  - 3.3.4 Einbinden der Entwickler . . . 35
  - 3.3.5 Aufwand . . . 35
- 3.4 Abschließende Einschätzung . . . 36

4 Zusammenfassung 39

Tabellenverzeichnis 43

Abbildungsverzeichnis 45

Quellen- und Literaturverzeichnis 47

Literaturverzeichnis 49

Erklärung zur Studienarbeit 51

# 1 Einleitung

*"In der professionellen Softwareentwicklung spielen Begriffe wie Konsistenz, Reproduzierbarkeit, Transparenz, Qualität, Stabilität, Automatisierung, Vermeidung von Redundanzen und Minimierung von Risiken eine wichtige Rolle. Arbeitet man im Team, entwickeln diese Begriffe eine neue Dimension. Häufig treten Probleme auf, weil die Kommunikation nicht stimmt und die Arbeit nicht gut koordiniert ist. Abhilfe schafft ein Continuous Build and Test System (CBTS), das die kontinuierliche automatische Übersetzung der Codebasis mit anschließender Ausführung des Testcodes unterstützt."* (Hal04, S.24)

Die Entwicklung von Software stellt seit jeher eine große Herausforderung dar. Das betrifft vor allem *die* Zeit, seit der an den Projekten mehrere Entwickler beteiligt sind. Dabei ist zu beachten, dass das Problem der Integration auch innerhalb des Entwicklungsprozesses gelöst werden muss. Der Prozess der Softwareentwicklung soll so gestaltet werden, dass organisatorische Probleme - natürlich so weit wie möglich - eingeplant und gelöst werden können. Eines dieser organisatorischen Probleme sind die Zusammenarbeit der Entwickler und deren Arbeitsmodule. Bisher waren Integrationsphasen an das Erreichen von Teilzielen in der Projektplanung gebunden. Im Gegensatz dazu soll die begleitende Integration während der Entwicklung parallel erfolgen und beinahe in Echtzeit Rückmeldungen über eventuelle Probleme geben (das Konzept dieser Vorgehensweise wird im Allgemeinen als *Kontinuierliche Integration*[1] bezeichnet). Die frühzeitige Beseitigung von Konflikten[2] - im Zusammenspiel der Teile des zu erstellenden Produktes - schafft die Voraussetzung für die Herstellung

[1] engl.: Continuous Integration, abgekürzt CI. Im Weiteren wird die deutsche Übersetzung "Kontinuierliche Integration" verwendet.

[2] Funktionale als auch strukturelle Abweichungen in der Spezifikation.

einer qualitativ anspruchsvollen Software. Außerdem werden erhebliche Kosten eingespart. Das ist darauf zurückzuführen, dass die Suche nach Fehlern, die in einer früheren Phase der Entwicklung gemacht wurden, ungleich aufwendiger und kostenintensiver ist. Abweichungen im Zeitplan des Projektes werden jeweils vor dem Beginn einer Integrationsphase sichtbar und lassen somit den Verlauf des Projektes besser einschätzen (Fre09, Die07, Tec09).

Diese Studienarbeit entsteht im Rahmen des Projektes „Cidas[2]“[3], welches, in Zusammenarbeit der FH Brandenburg und eines in Gründung befindlichen Unternehmen ein komplexes Java Enterprise Produkt entwickelt. An dem Projekt sind acht Mitarbeiter beteiligt. Die *Kontinuierliche Integration* soll mit Hilfe geeigneter Software umgesetzt werden. Auf diese Weise werden Entwicklungszyklen kurz und auf einem hohen Qualitätsniveau ablaufen.

### Ziel der Arbeit

Aus den bisherigen Ausführungen ergibt sich für diese Arbeit folgende Zielstellung: Untersuchung der Machbarkeit des Konzeptes der *Kontinuierlichen Integration* mit Hilfe eines zu wählenden Werkzeuges im Rahmen eines konkreten Entwicklungsprozesses. Das Konzept sieht vor, den Entwicklungsprozess in seiner Implementierungsphase zu jeder Zeit auf einem qualitativ hohen Stand zu halten. Auf diese Weise sollen die Phasen eines instabilen[4] Produktes nahezu eliminiert werden.

[3]Gefördert durch das Ministerium für Wissenschaft, Forschung und Kultur des Landes Brandenburg im Rahmen der Forschungs- und Innovationsförderung zur Steigerung der Innovationskraft an Brandenburger Hochschulen.

[4]"Stabilität in der Informatik bedeutet, dass sich ein Betriebssystem oder eine Anwendungssoftware unter möglichst vielen Randbedingungen so verhält, dass es weder zu Datenverlust, noch zu unerwünschten Schutzverletzungen oder Systemabstürzen kommt" (wik09b).

Dabei sind folgende Fragen zu beantworten:

- Wie kann der Entwicklungsprozess nachhaltig und wirksam verbessert werden?
- Wie läßt sich das Konzept in den Entwicklungsprozess integrieren?
- Können Entwicklungsrisiken wirksam minimiert werden?
- Wird die Qualität langfristig erhöht oder ergeben sich auch kurzfristig sichtbare Resultate?

Im Rahmen dieser Untersuchungen werden u.a. angrenzende Themenbereiche, wie z.B. Risikomanagment und testgetriebene Entwicklung, berührt. Diese werden jedoch nur in der für diese Arbeit notwendigen Tiefe erörtert.

#### Aufbau der Arbeit

Zur Beantwortung dieser Fragen werden wir uns im Kapitel 2 mit der *Agilen Softwareentwicklung* im Allgemeinen und Praktiken zur Umsetzung des Konzeptes der *Kontinuierlichen Integration* im Speziellen befassen. Im Kapitel 3 wird dann deren praktische Umsetzung beschrieben: die Vorgehensweise, auftretende Probleme, deren Lösungen und entsprechende Verbesserungen des Enwicklungsprozesses. Abschließend sind im Kapitel 4 die Arbeitsergebnisse dieser Arbeit zusammenfassend dargestellt.

# 2 Kontinuierliche Integration

*"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. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly"* (Fow06).

Das Konzept der *Kontinuierlichen Integration* ist eine Vorgehensweise, die die Entwicklung von Software über den gesamten Prozess der Implementierung mit regelmäßigen Integrationsphasen begleitet. Dieses Konzept ist eine der Säulen[1] im agilen Softwareentwicklungsprozess (Tho06). Um die Bedeutung der *Kontinuierlichen Integration* innerhalb des agilen Softwareprozesses richtig einzuschätzen, ist es notwendig, auf diesen nachfolgend besonders einzugehen.

## 2.1 Agile Softwareentwicklung

*We are uncovering better ways of developing*
*software by doing it and helping others do it.*

Vorwort zum Manifest der Agilen Softwareentwicklung (u.a01).

Mit der steigenden Popularität agiler Softwareentwicklung, hervorgerufen durch das von Kent Beck im Jahre 1999 veröffentlichte Buch "*Extreme Programming*" und dem Wunsch, traditionelle schwerfällige Vorgehensmodelle[2] zugunsten leichter und agiler Vorgehensweisen abzulösen,

[1] Außerdem existieren weitere 13 Praktiken, die von ergänzender Bedeutung sind.
[2] Wasserfall-Modell, V-Modell u.a..

wurde das “*Manifest der Agilen Softwareentwicklung*” unterzeichnet.
In diesem Zusammenhang ist es notwendig, den Begriff *Agilität* zu definieren:

> “***Agilität*** *ist die Fähigkeit, positiv auf erforderliche Veränderung hinsichtlich der Geschwindigkeit, Zielrichtung, Massnahmen und des Zeitrahmens zu reagieren”* (fTQM);
>
> “***agil*** *betriebsam, beweglich, energiegeladen, geschäftig, geschickt, gewandt, lebhaft, quecksilbrig, rege, rührig, temperamentvoll, unruhig, vital, wendig; ...”*(DUD09).

Die Aufgabe bestand darin, die Entwicklung von Software zu verbessern und solche Methoden anzuwenden, die den Prozess insgesamt agiler werden lassen. *Agilität* bzw. *agiles Vorgehen* beschreibt den Gegensatz zu traditionellen Vorgehensweisen, deren lange Iterationszyklen und streng formalisierten Arbeitsabläufe einer zeitnahen Anpassung im Wege stehen. Dadurch wird die Entwicklung beschleunigt und es fallen weniger Kosten an. Diese und weitere Überlegungen führten zur Festschreibung folgender Grundwerte in Form von Prioriäten (u.a01):

- **Individuen und Interaktionen** vor Prozessen und Werkzeugen,
- **funktionsfähige Software** vor umfassender Dokumentation,
- **Zusammenarbeit mit dem Kunden** vor Vertragsverhandlungen,
- **Flexibilität hinsichtlich veränderter Anforderungen** vor Planerfüllung[3].

Zu berücksichtigen sind außerdem zwölf Prinzipien (u.a01), die diesen Grundwerten entsprechen. An dieser Stelle werden jedoch nur jene *vier* Prinzipien genannt, denen im weiteren Verlauf eine besondere Bedeutung zukommt:

[3] *„Je mehr Du nach Plan arbeitest, um so mehr bekommst Du das, was Du geplant hast, aber nicht das, was Du brauchst.“* Leitsatz der agilen Softwareentwicklung (Wik09a).

- "*Unsere höchste Priorität besteht darin, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller/nützlicher/geschätzter Software zufrieden zu stellen.*"[4]

- "Akzeptieren von sich ändernden Anforderungen, auch zu einem späten Zeitpunkt der Entwicklung. Agile Prozesse nutzen Änderungen, um Vorteile für die Wettbewerbsfähigkeit des Kunden zu erzeugen."[5]

- "Liefere funktionsfähige Software regelmäßig aus, innerhalb einiger Wochen bis zu einigen Monaten, bevorzuge dabei eine kürzere Zeitspanne."[6]

- "Funktionsfähige Software ist das primäre Maß für den Fortschritt."[7]

Der agile Softwareentwicklungsprozess nach (Tho06) ist, ebenso wie das gleichnamige *Agile Modell* und *XP*[8], als abstrakter Oberbegriff für konkrete Vorgehensmodelle zu verstehen. Dabei wurden Schlüsselverfahrensweisen entwickelt, die sich in allen Modellen dieser Gattung wiederfinden. Diese werden kurz beschrieben, um auf diese Weise der Entwicklung von Software besondere Aufmerksamkeit zu widmen. Dabei soll ausdrücklich betont werden, dass diese Techniken zuerst in anderen Bereichen Anwendung fanden und erst dann den Besonderheiten der Softwareentwicklung angepasst wurden.
Im Weiteren einige Ausführungen zu den entsprechenden Vorgehensmodellen.

- **Iterative Entwicklung** bedeutet, dass Entwicklungszyklen klein gehalten werden sollen, um mit wenigen, aber wichtigen Zielen ein Ergebnis zu erreichen, das getestet und überarbeitet werden kann. Die einzelnen Phasen sollen dabei eine konstante Dauer haben und vorzugs-

[4] "*Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*" (u.a01)
[5] "*Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*" (u.a01)
[6] "*Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.* " (u.a01)
[7] "*Working software is the primary measure of progress.*" (u.a01)
[8] Abkürzung für *Extreme Programming*.

weise konkrete Geschäftsnutzwerte erfüllen[9]. Diesem Konzept der iterativen Entwicklung liegt die Idee der *"Schmalen Produktion"*[10] zugrunde (Pop09).

- **Automatisiertes Testen**. Durch die Entwicklungsgeschwindigkeit und *iterative Entwicklung* ergibt sich die Notwendigkeit, regelmäßig lauffähige Software zu veröffentlichen und zu überprüfen. Mit automatisierten Unittests und Akzeptanztests kann dazu ein wesentlicher Beitrag geleistet werden. Dabei wird das Testen, was jedoch nicht unbedingt erforderlich ist, durch ein testgetriebenes Design und eine testgetriebene Entwicklung unterstützt[11].

- **Kontinuierliche Integration** bezieht sich auf eine vollständig automatisierte Umgebung, die den Buildprozess und das Testen der Software übernimmt. Prinzipiell wird nach jeder kleinen Änderung das Gesamtsystem überprüft. Auf diese Weise sollen die Kosten der jeweiligen Veränderung hinsichtlich ihrer Auswirkung auf das Gesamtsystem gesenkt und die eventuellen Ursachen für das Auftreten von Fehlern schnell gefunden werden. Ungewollte Seiteneffekte werden durch Tests erkannt und erstatten somit zeitnah Rückmeldung über etwaige Abweichungen von der Spezifikation. Die *Kontinuierliche Integration* bietet auch die Möglichkeit, in einer späten Phase der Entwicklung von Software auf veränderte Anforderungen des Kunden zu reagieren und diese entsprechend zu integrieren.

- **User-Stories** sind die Grundlage der agilen Entwicklung. Sie werden als Kurzbeschreibung entsprechend einer Anforderung vom Kunden verfasst, meist als langer Einzeiler (Tho06). Aus den *User-Stories* werden die Akzeptanzkriterien gebildet, deren Erfüllung Grundlage für die weitere Entwicklung des Projektes ist.

[9] Jeder User-Story ist ein Geschäftsnutzwert (engl.: Businessvalue) zugewiesen.

[10] Besser bekannt unter dem englischen Begriff des "*Lean Manufacturing*", welches als eine generische Prozessmanagementphilosophie hauptsächlich von Toyota abgeleitet wurde (Lea09).

[11] Man verwendet in diesem Zusammenhang oft die englischen Begriffe *Test-driven-design* und *test-driven-development*.

- Die **Einbeziehung des Kunden** ist bei der agilen Vorgehensweise besonders wichtig. So wird in (ST09, S.4) festgestellt, dass von 260 Projekten mehr als 60% erfolgreich abgeschlossen worden sind. Dabei wurde der Kunde kontinuierlich oder zumindest mehrmals wöchentlich in die Entwickung einbezogen. Bei dieser engen Bindung können kontinuierlich Feedback gegeben und Fehlentwicklungen vermieden werden. Außerdem steigt die Produktivität durch kurze Antwortzeiten. Eventuell notwendig gewordene Korrekturen falscher Annahmen können signifikant reduziert werden.
  Diese Fakten beweisen eindeutig die Vorteile der Einbeziehung des Kunden in den *agilen* Prozess.

Die langjährigen Erfahrungswerte eines Individualsoftwaredienstleisters[12] bestätigen, dass die gesetzten Ziele in der Regel erreicht werden. Es wird Software entwickelt, die den Anforderungen des Kunden besser gerecht wird. Dabei ist hervorzuheben, dass die *agile* im Vergleich zur *klassischen* Vorgehensweise kostengünstiger ist. Das wird besonders in Abb. 2.1, S. 10 veranschaulicht.

Die Grundidee bei der *Kontinuerlichen Integration* besteht darin, nach jeder kleinen Veränderung das gesamte System bzw. Produkt *automatisch* validieren zu lassen. Diese Validierung setzt die Anfertigung entsprechender Tests voraus[14]. Um den Überprüfungsvorgang zu automatisieren und bei Wiederholung mit minimalem Aufwand durchzuführen, müssen die Tests ohne regelmäßige manuelle Eingriffe erfolgen können. Erst dann ist es überhaupt möglich, gesamte Systeme nach erfolgten kleinen Änderungen umfangreichen und komplexen Tests zu unterziehen.

Weshalb wird eigentlich der Durchführung regelmäßiger Tests so große Bedeutung beigemessen?

[12] ANECON wurde 1998 gegründet und beschäftigt mehr als 90 Mitarbeiter. Die Firma war an 48 Projekten im öffentlichen und privatwirtschaftlichen Bereich beteiligt. `http://www.anecon.com/.success-stories/` – 2009.11.16 18:56

[13] Als klassisches Vorgehensmodell werden u.a. VT, als agiles Vorgehensmodell XP genannt.

[14] Hier wird das Thema der testgetriebenen Entwicklung berührt und die damit verbundene Problematik deutlich.

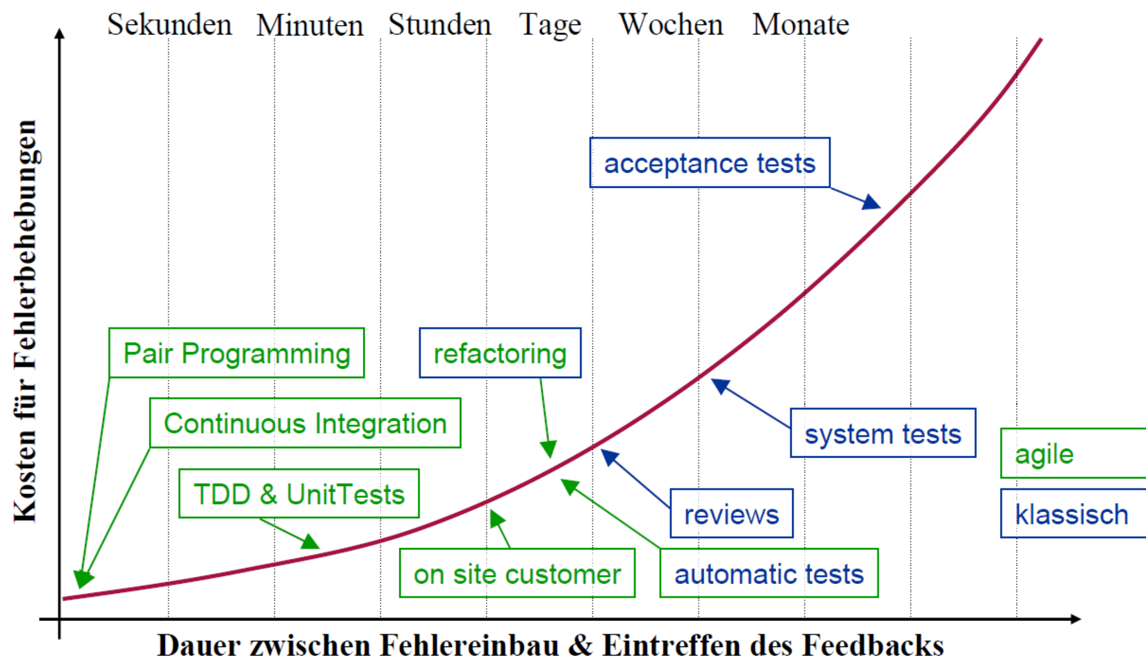


Abbildung 2.1: Gegenüberstellung der Kostenentwicklung hinsichtlich der Fehlerbehebung beim Einsatz eines *klassischen* im Vergleich zu einem *agilen* Vorgehensmodell[13] (Die07).

Die Idee, das System kontinuierlich zu testen, folgt der Erkenntnis, dass die Kosten zur Beseitigung des Fehlers mit der weiteren Entwicklung des Projektes *exponentiell* ansteigen. Dem kann jedoch entgegen gewirkt werden, indem die Software mit Hilfe *Kontinuierlicher Integration* frühzeitig überprüft wird. Es ist bekannt, dass 85% der Fehler in der Implementierungsphase - also in einer Niedrigkostenphase - entstehen bzw. auftreten (siehe dazu Abb. 2.2, S. 11).

Zwecks Reduzierung der Kosten durch Minimierung von Risiken sind im Rahmen der *Kontinuierlichen Integration* entsprechende Arbeitstechniken entwickelt worden. Auf einige dieser Techniken soll im Weiteren näher eingegangen werden. Dabei wird deutlich, wie die gestellte Zielsetzung zu erreichen ist.

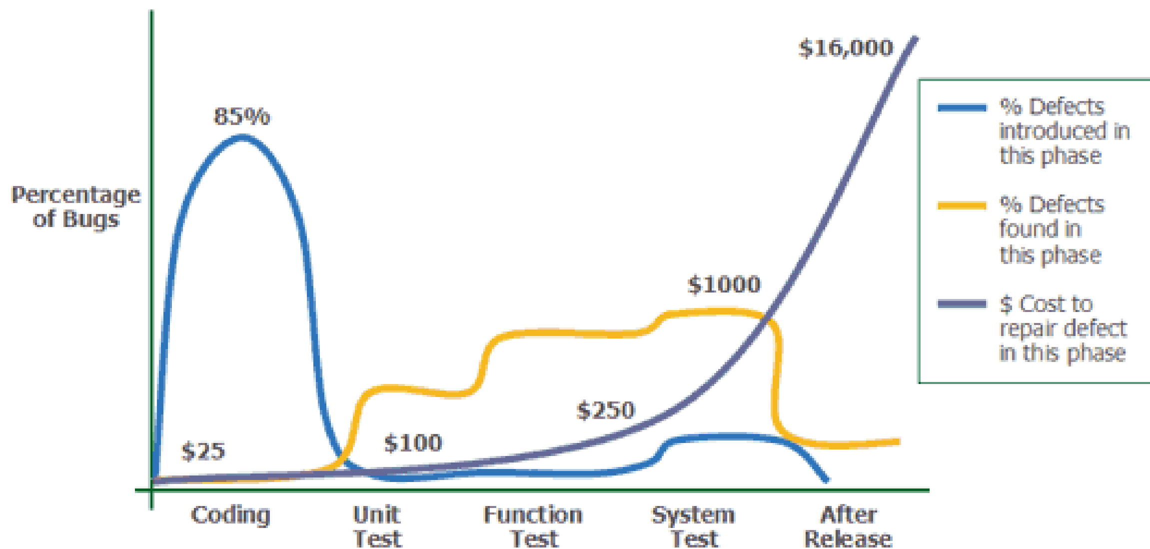


Abbildung 2.2: Entwicklung der Kosten für die Fehlerbereinigung in Abhängigkeit zur Zeit und der Verteilung der entstandenen Fehler (Tec09).

## 2.2 Praktiken

Martin Fowler benennt für den Einsatz der *Kontinuierlichen Integration* notwendige Vorraussetzungen und Praktiken, die sich als nützlich erwiesen haben (Fow06).

- **Zentrale Quellcodeversionierung**. Softwareprojekte bestehen aus vielen Dateien und entstehen durch die Zusammenarbeit unterschiedlicher Entwickler (siehe Abb. 2.3, S. 12). Um dieses Zusammenwirken zu koordinieren, benötigt man eine zentrale Instanz. Im einfachsten Fall sind das Verzeichnisse im Dateisystem, auf die gemeinsam zugegriffen und in denen der Code abgelegt wird. Eine komfortablere und zeitgemäßere Lösung stellt ein Quellcodeverwaltungssystem[15] im Sinne von CVS[16] dar, im Folgenden als *SCM*[17] bezeichnet. Weitere Hilfsmittel werden - neben der reinen Verwaltung der Versionen - durch

[15] Engl.: source code managment system.

[16] Concurrent Version System. Viele Jahre ist CVS eines der weitverbreitetesten Versionsverwaltungen im Bereich der Softwareentwicklung gewesen. Es wird nun zunehmend durch *Subversion* und verteilte Versionsverwaltungen, wie *git* und *Mercurial*, abgelöst.

[17] Abkürzung für Software Configuration Management.

moderne *SCM* angeboten. Sie bieten neben der Netzwerkfähigkeit auch die Darstellung der Änderungshistorie an. Das *SCM* ist ein integrales Werkzeug für die Zusammenarbeit der Entwickler. Alles, was für die Erstellung einer Version der Software notwendig ist, sollte in das *SCM* hineingeladen werden. Dazu gehören Testskripte, Einstellungsdateien, Datenbankschemata, Installationsskripte, aber auch externe Programmbibliotheken. Dabei muss beachtet werden, dass nicht etwas hineingeladen wird, was aus dem Buildprozess selbst entsteht.

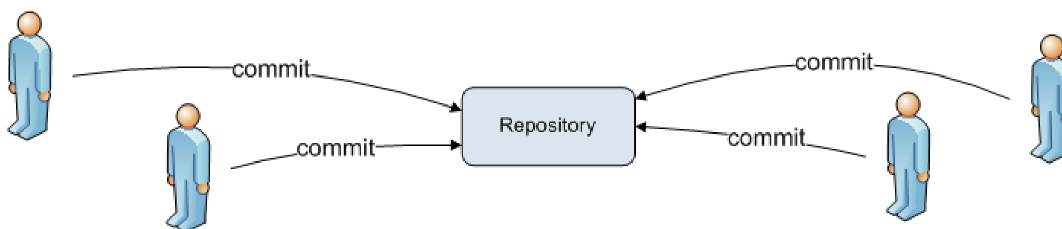


Abbildung 2.3: Übertragung[18]in eine zentrale Quellcodeverwaltung.

- **Automatisierter Build**. Zwecks Erstellung einer neuen lauffähigen Version ist es notwendig, den Buildprozess erfolgreich ablaufen zu lassen. Da dieser sehr umfangreich sein kann, müssen alle Teilschritte soweit automatisiert werden, um einen manuellen Eingriff zu vermeiden. Diese Automatisierung kann mit Hilfe von Kompilierwerkzeugen[19] erfolgen. Diese Werkzeuge sollen alle Aufgaben - vom Herunterladen der aktuellen Quellen über das Anlegen der Datenbankschemata bis hin zum Ausführen der Tests - durchzuführen. Die Faustregel besagt, dass es auf einem neu eingerichteten Rechner mit grundsätzlichen Voraussetzungen - dazu gehören z.B. das Betriebssystem, die Laufzeitumgebung und einige wenige Basisprogramme - mit wenigen manuellen Eingriffen möglich sein sollte, die Software zum Laufen zu bringen.

- **Selbsttestender Build**. Ein *Build* beschreibt den Kompiliervorgang, der aus dem Quellcode eine lauffähige Software erstellt. Mit Hilfe des

[18]deutsch: Übertragung; Hiermit ist die Übertragung einer Änderung in ein Quellcodeverwaltungsystem gemeint.
[19]Dazu gehören u.a. *Ant*, *make* und *maven*.

Einsatzes statisch typisierter Programmiersprachen lassen sich schon während des Kompiliervorgangs viele formale Fehler erkennen. Trotzdem werden jedoch infolge der Komplexität von Software viele weitaus schwerwiegendere Fehler nicht erkannt. Deshalb ist es empfehlenswert, für jede Funktionalität, die bereitgestellt werden soll, entsprechende Spezifikationen in Form von Tests anzufertigen. Diese sollen dann im Anschluß an den Kompiliervorgang ausgeführt werden. Wenn ein Test den gestellten Anforderungen nicht genügt, gilt der Build als fehlgeschlagen. Hierbei ist anzumerken, dass es immer günstiger ist, über zumindest unvollkommene Tests zu verfügen. Nicht vorhandene Tests stellen die schlechtere Variante dar.

- **Commit in die Hauptentwicklungslinie**. Bei der Integration geht es primär um Kommunikation. Um die Zusammenarbeit zu gewährleisten, sollten die Entwickler alle Änderungen in eine gemeinsame Hauptentwicklungslinie hineinladen. Deshalb müssen sie regelmäßig Gespräche führen, um auf diese Weise Auswirkungen von Veränderungen sofort sichtbar werden zu lassen.
  Bei kleinen Änderungen ist die Fehlersuche erheblich einfacher. Wenn die Arbeit von Tagen – mit mehreren hundert Zeilen Code – eingestellt wird, stellt sich das wesentlich schwieriger dar. Deshalb ist es wichtig, die eingebrachten Änderungen am Code klein und übersichtlich zu halten und oft einzuchecken. Vorraussetzung für das Übertragen der Änderung in das *SCM* ist, dass die Ausführung der Tests zum Build lokal beim Entwickler fehlerfrei abläuft.

- **Kompilierung jeder Version auf einem Integrationsrechner**. Nach dem Übertragen einer neuen Version in das *SCM* sollte diese getestet werden. Das muss durch einen kompletten "Neubau"[20] auf einem Integrationsrechner erfolgen. Die einfachste Form, den Kompiliervorgang zu beginnen, besteht darin, ihn manuell zu starten. Als wesentlich bessere Variante ist allerdings ein Integrationsserver zu betrachten. Dieser kann selbstständig auf Änderungen im *SCM* reagieren und einen definierten Integrationsvorgang durchführen. Der Integrationsvorgang

[20] *Neubau* bezieht sich hierbei auf den englischen Begriff des "clean build". In diesem Fall sind inkrementelle Kompiliervorgänge ausdrücklich untersagt.

umfasst das Kompilieren, das Kopieren der Software in eine definierte Zielumgebung und die Durchführung von Tests. Die Tests können sich auf die geforderte Funktionalität hinsichtlich der Spezifikation beziehen, aber auch allgemeingültig sein. In diesem Zusammenhang soll auf die statische Quellcodeanalyse hingewiesen werden.
Sollte sich während des automatisierten Integrationsvorgangs ein Bruch in der Stabilität der Hauptentwicklungslinie zeigen, muss der Korrektur höchste Priorität beigemessen werden. Weil darauf zu achten ist, dass die Hauptenwicklungslinie stabil bleibt.

- **“Keep the build fast!”**[21]. Die *Kontinuierliche Integration* im Umfeld der agilen Softwareentwicklung basiert auf der Arbeit mit geringen Verzögerungen. Dazu gehört auch, dass das Feedback des Integrationsvorgangs möglichst *unmittelbar* nach dem Einstellen einer Änderung erfolgt (siehe Abb. 2.4, S. 15). Bis zu zehn Minuten werden für die Dauer eines Builds als angemessen angesehen.
  Für größere Projekte hat sich die Aufteilung in einen zweistufigen Buildprozess als sinnvoll erwiesen. Auf diese Weise erfolgen in der ersten Stufe rudimentäre und schnelle Funktionstests. In der zweiten Stufe werden - einschließlich der relativ zeitintensiven Datenbankabfragen - größere Tests durchgeführt. Bei diesem Modell kann eine Verteilung auf mehrere Integrationsserver einen zeitlichen Vorteil ergeben.

- **Test in geklonter Produktivumgebung**. Grundsätzlich sollen alle Tests in einer den realen Bedingungen entsprechenden Umgebung stattfinden. Dass dabei Probleme hinsichtlich der möglichen Variationen und Seiteneffekte auftreten, darf an dieser Stelle nicht verschwiegen werden.

- **Einfacher Zugang zur neuesten ausführbaren Version**. Dem Risiko einer mangelhaften Kommunikation zwischen *Stakeholder* und Entwickler wird damit begegnet, dass in kurzen Abständen neue Entwicklungsversionen zur Verfügung gestellt werden. Auf diese Weise können kritische Hinweise direkt in den Entwicklungsprozess einbezogen werden.

[21] Sinngemäß: Achte darauf, dass der Kompiliervorgang nicht zu langsam verläuft.

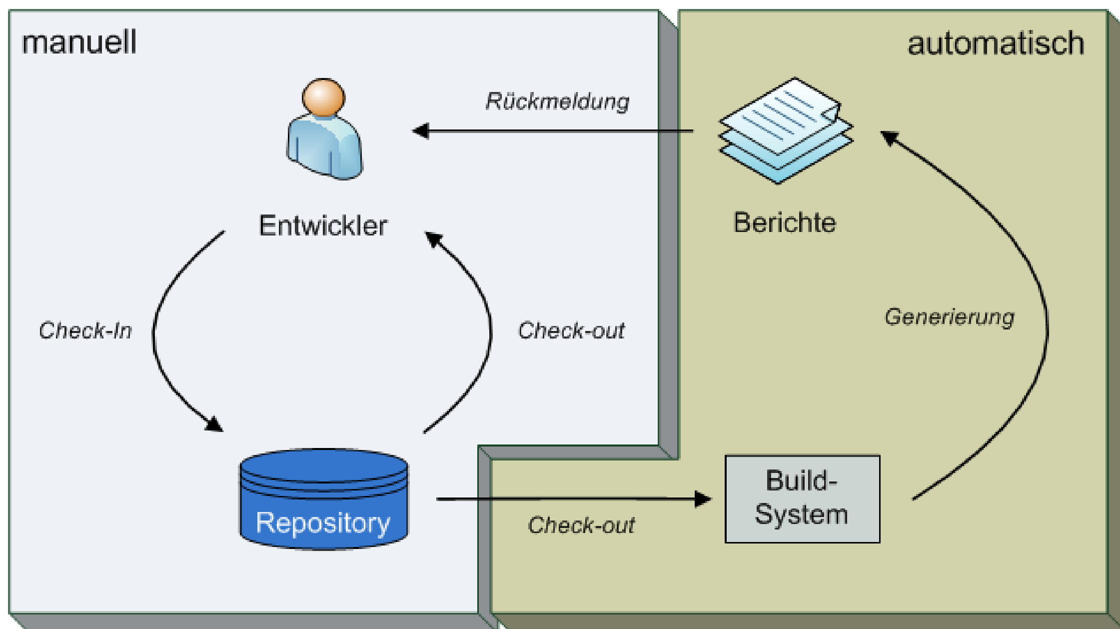


Abbildung 2.4: Empfohlener Arbeitsfluß des Entwicklers mit kontinuierlichen Integrationstests und Berichten über Erfolg- und Mißerfolg einzelner Testläufe. Hierbei ist wichtig, daß der automatische Teil des Prozesses in minimaler Zeit abläuft.

Dadurch kann von Seiten des Auftraggebers der Kurs auch ohne größere Verzögerung korrigiert werden. Das ist deshalb wichtig, weil die Formulierung einer konkreten Erwartungshaltung oft nicht ganz einfach ist.

- **Jeder soll Veränderungen wahrnehmen**. Während des Entwicklungsprozesses ist es notwendig, dass allen Beteiligten Einblick in die Änderungen gegeben wird. Jeder Beteiligte muss erkennen können, welche Änderungen erfolgt sind, ob sie eventuell Auswirkungen auf die eigene Bereiche haben. Aktuelle *SCM* bieten die Möglichkeit, in die Änderungshistorie Einblick zu nehmen. Werkzeuge im Umfeld dieser *SCMs* stellen die Änderungshistorie bereits übersichtlich in Form von Webfrontends dar (siehe Abb. 2.5, S. 16.). Diese Werkzeuge verknüpfen z.B. Kommentare mit einer konkreten Änderungsübersicht oder

Aufgaben mit entsprechenden *changesets*[22]. Mit Hilfe dieser Werkzeuge ist es verhältnismäßig einfach, die Entwicklung eines Projektes in Echtzeit zu verfolgen. Dadurch erhält man einen Überblick über den Fortschritt des Projektes.

**Changeset 3468**

Timestamp: 01/19/10 16:31:26 (3 hours ago)
Author: jens
Message: getter / setter angepasst
Files: 1 modified
trunk/services/BaseService/src/de/idema/entity/Provider/Provider.java (4 diffs)

View differences: inline
Show 10 lines around each change
Ignore:
Blank lines
Case changes
White space changes
Update

Unmodified Added Removed

trunk/services/BaseService/src/de/idema/entity/Provider/Provider.java — Tabular | Unified

```
r3421 r3468
38 38 import de.idema.entity.ProfileTemplate.Template_Profile;
39 39 import de.idema.helper.CollectionHelper;
40 40
41 41 // TODO: Auto-generated Javadoc
42 42 /**
43 43  * The Class Provider.
44 44  */
45 45 @Entity
46 46 @Table(name = "Provider", uniqueConstraints = { @UniqueConstraint(columnNames = { "username" }) })
47 47 @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
48    public class Provider extends EntityAbstract<Provider>
49    {
   48 public class Provider extends EntityAbstract<Provider>{
```

Abbildung 2.5: Übersicht über eine Änderung am Beispiel von *trac*[23].

- **Automatische Auslieferung**[24] Das Verteilen neuer Softwareversionen soll automatisisch erfolgen. Somit können die Stufe-1 und Stufe-2 Tests (siehe dazu Abschnitt 2.2, S. 14) schneller, einfacher und öfter ablaufen. Außerdem muss die automatische Auslieferung auch den Fehlerfall beinhalten und gleichzeitig das *Zurücksetzen* der Software in einen Zustand, der auf einer früheren Version basiert, ermöglichen. Nachdem diese beiden Fälle automatisiert sind, steht dem Einsatz des

[22] deutsch: Änderungsmengen. Änderungen können in mehreren Einstellungen zusammengefasst werden. Eine Menge von Änderungen kann sich somit auf mehrere aufeinanderfolgende Versionen erstrecken.

[23] "Trac ist ein freies, webbasiertes Projektmanagement-Werkzeug zur Softwareentwicklung. Es enthält eine webbasierte Oberfläche zum Betrachten von Subversion-Repositories, ein Wiki zum kollaborativen Erstellen und Pflegen von (z. B.) Dokumentation und einen Bugtracker zum Erfassen und Verwalten von Programmfehlern und Erweiterungswünschen."(Wik10c) (siehe `http://trac.edgewall.org/`)

[24] Hiermit ist die Auslieferung in eine definierte Zielumgebung gemeint.

gesamten Mechanismus für die Produktivumgebung nichts mehr im Weg.

Die Umsetzung des Konzeptes der *Kontinuierlicher Integration* wird durch die Anwendung der genannten Vorgehensweisen *vereinfacht*, erfordert diese jedoch nicht zwangsläufig.

Beim Einsatz dieses Konzeptes steht die Minimierung von Risiken, die mit der Entwicklung von Software verbunden sind, im Vordergrund. Wichtige Risiken sind u.a. folgende:

- Erstellung eines Produktes, dass der Kunde so nicht wollte,
- Falsche Aufwandabschätzungen,
- Verschiebungen im Terminplan,
- zu hohe Fehlerrate,
- Einsatz einer zu komplexen Technologie.

Auf diese Weise ist es möglich, die Zeitdauer für das Erreichen eines Teilziels besser einzuschätzen und "tote Winkel"[25] in Form von Integrationsproblemen zu vermeiden. Insgesamt wird die Fehlersuche durch die beschriebenen Vorgehensweisen erheblich beschleunigt. Es sei auch angemerkt, dass dieser Vorteil direkten Einfluß auf die Qualität der zu erstellenden Software hat und die Anfertigung einer angemessenen Testsuite erforderlich macht. Abschließend merkt Martin Fowler in (Fow06) dazu an:

> *"As a result projects with Continuous Integration tend to have dramatically less bugs, both in production and in process."*

[25]In Anlehnung an den englischen Ausdruck "blind spots".

# 3 Fallbeispiel

Nachdem im Kapitel 2, S. 5 die theoretischen Aspekte der *Kontinuierlichen Integration* erläutert wurden, soll im Folgenden untersucht werden, welche Auswirkungen deren Anwendung in der Praxis hat. Dabei wird zunächst die Ausgangssituation beschrieben. Anschließend werden Vorteile genannt, die infolge der Anwendung des neuen Konzeptes erkennbar geworden sind. Danach wird der für die erfolgreiche Umsetzung notwendige Aufwand dargelegt. Auf die dabei auftretenden Probleme wird ebenfalls eingegangen. Ergebnisse dieser Arbeit bzw. entsprechende Erkenntnisse werden am Ende dieses Kapitels in Form eines Fazits dargelegt.

Das Projekt, welches dieser Arbeit zugrunde liegt, ist ein verteiltes Java-Projekt. Sowohl die Entwicklung der Software, als auch ihr Betrieb erfolgen verteilt. In das Projekt sind mehrere Entwickler eingebunden, die an verschiedenen Orten arbeiten und ihre Arbeit über moderne Kommunikation (Telefon / Skype[1] / Emails etc.) koordinieren. Die Software setzt als Technologie - außer Java - den Applikationsserver JBoss[2], EJB[3] und jBPM[4] ein. In dem Team arbeiten Einsteiger und erfahrene Entwickler. Für sie alle ist die Arbeit in einem größeren verteilten Projekt neu. Von Beginn an war es die Absicht der Projektleitung, den Prozess in einer *agilen* Art und Weise in Abstimmung mit den Entwicklern zu gestalten. Somit kann dieses Projekt als repräsentativ gelten.

[1] Schließt Instantmessaging und VoIP mit ein.
[2] `www.jboss.org`.
[3] Enterprise Java Beans.
[4] Business Modeling Notation für Java.

Die Anpassung des Entwicklungsprozesses und die Einarbeitung des Konzeptes der *Kontinuierlichen Integration* erforderte folgende Arbeitsschritte:

- **Erfassen des IST-Zustandes**,
- **Formulieren des SOLL-Zustandes**,
- **Umsetzung des Konzeptes**.

## 3.1 Der IST-Zustand - vor der Einführung

Vor der Einführung von *Kontinuierlicher Integration* kamen bereits einige Praktiken bzw. Methoden, wie sie Martin Fowler in (Fow06) vorschlägt, zur Anwendung. So wurde schon im Vorfeld ein SCM in Form von *Subversion* genutzt und *in die Hauptentwicklungslinie eingecheckt*, so dass *von jedem Entwickler Änderungen wahrgenommen werden konnten*. Von den in Abschnitt 2.2, S. 11 empfohlenen Arbeitspraktiken wurden somit bereits diese drei umgesetzt, wie in Tab. 3.1, S. 21 zusammenfassend dargelegt worden ist.

Die Durchführung der Tests mit der Entwicklungsumgebung führte bereits zu einer relativ guten Zeit beim Kompilieren, erforderte jedoch eine manuelle Bedienung. Somit wurden die Forderungen nach einem *automatisierten* und *selbsttestenden Build* nicht erfüllt. Die fehlende Automatisierung führte dazu, dass Teile des Systems unregelmäßig getestet wurden.

Die Durchführung des gesamten Kompiliervorganges durch eine zweite Person führte zu prinzipbedingten zeitlichen Verzögerungen, so dass die Forderung nach "*keep the build fast*" (siehe Abschnitt 2.2, S. 14) nur teilweise erfüllt wurde. Dies gilt gleichermaßen für die Forderung den *Test in einer geklonten Produktivumgebung* durchzuführen.

Die Ausführung eines Testes machte es notwendig, dass eine Person die betreffenden Teile aus dem SCM aktualisierte und manuell auslieferte. Um diesen Vorgang in Bewegung zu setzen, wurde mit den Entwicklern jeweils abgesprochen, ob der entsprechende Teil zum Testen bereit ist. Stellte man

jedoch bei einem Test Fehler fest, so machte der Tester den Entwickler darauf aufmerksam. Gemeinsam wurden dann die Fehler beseitigt und der Test erneut angestoßen. In dieser Zeit waren oft beide Personen mit dieser Aufgabe voll beschäftigt und somit für andere Tätigkeiten blockiert.

| Arbeitsziele | IST | SOLL |
|---|---|---|
| Zentrale Quellcodeversionierung | ✓ | ✓ |
| Commit in die Hauptentwicklungslinie | ✓ | ✓ |
| Jeder soll Veränderungen wahrnehmen | ✓ | ✓ |
| "Keep the build fast!" | ❋ | ✓ |
| Test in einer geklonten Produktivumgebung | ❋ | ✓ |
| Automatisierter Build | ✗ | ✓ |
| Selbsttestender Build | ✗ | ✓ |
| Kompilierung jeder Version auf einem Integrationsrechner | ✗ | ✓ |
| Einfacher Zugang zur neuesten ausführbaren Version | ✗ | ✓ |
| Automatische Auslieferung | ✗ | ✓ |
| Legende : ✓= ja/vollständig, ✗= nein, ❋= teilweise/unvollständig | | |

Tabelle 3.1: Status der Arbeitsziele (auf den Praktiken von 2.2, S. 11ff. aufbauend).

Diese Vorgehensweise nahm für einen kompletten Integrationstest teilweise bis zu einen Tag in Anspruch. Der Verkürzung dieser Zeitdauer dienten Hilfsmaßnahmen, um so den manuellen Eingriff zu minimieren. Deshalb wurden die aus der Projektstruktur entstandenen vielen kleinen Projekte in Metaprojekte zusammengefaßt. Diese Metaprojekte dienten dem vereinfachten Verteilen der Software und der Einsparung von Zeit. Dadurch konnte die Dauer eines kompletten Tests - ohne Fehlerkorrektur - auf eine halbe Stunde verkürzt werden (Zeitdauer vorher ca. 1/2 Tag).

Allerdings kam es oftmals zu Verzögerungen durch vom Tester verursachte Fehler. Zum einen lag das am modularen System, wo einige Teile durch das manuelle Durchführen nicht aktuell gehalten bzw. übersehen wurden. Zum anderen lag es am Tester, da dieser gleichzeitig in der Entwicklung tätig war. Dabei wurde seine Ausführungsumgebung sporadisch verändert.

Die Bedingungen für den Test konnten somit nicht stabil gehalten werden.

Die Arbeit des Entwicklers und des Testers erfordert, wenn sie von einer Person durchgeführt wird, einen sehr großen Aufwand. Der Aufwand für die Einrichtung einer identischen Testumgebung erschien jedoch zu diesem Zeitpunkt unangemessen hoch und wurde daher nicht in Erwägung gezogen. Das Testen insgesamt war langwierig und monoton. Auf diese Weise wurde einer Häufung von Fehlern durch den manuellen Betrieb Vorschub geleistet.

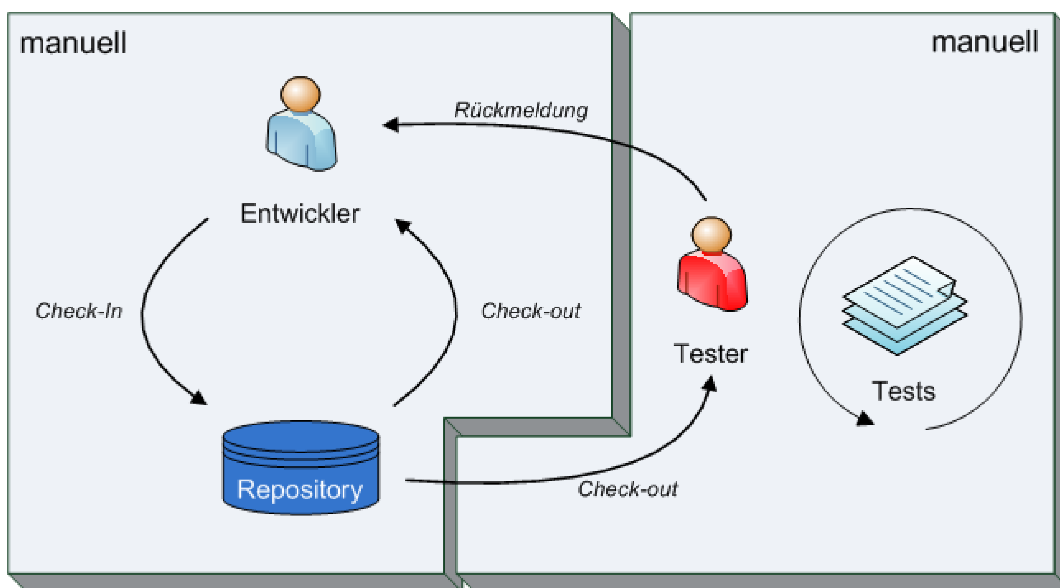


Abbildung 3.1: Der Arbeitsfluss nach IST-Stand mit einer starken *manuellen* Ausprägung - verbunden mit hohem zeitlichen und personellen Aufwand.

Schwerpunkt der Umsetzung wird, wie aus Tab. 3.1, S. 21 hervorgeht, der *Build*-Prozess sein. Die weiteren Ausführungen werden das belegen.

## 3.2 Formulierung des SOLL-Zustandes

Wie die auf der Grundlage der Einführung des Konzeptes der *Kontinuierlichen Integration* definierten Ziele zu erreichen sind, wird im Folgenden aus den einzelnen Arbeitsschritte ersichtlich (siehe dazu Tab. 3.1, S. 21).

Grundsätzlich kann jetzt damit begonnen werden, die Bestandteile des Konzeptes, wie sie von Martin Fowler formuliert wurden, in die Praxis umzusetzen (siehe dazu Kapitel 2.2, S. 11ff). Dabei soll auf eine nahtlose Integration der bestehenden Projektkonfiguration und auf die Motivation der Entwickler zu deren Nutzung geachtet werden.

Die Umsetzung des Konzeptes erfolgt in mehreren Schritten, die im Weiteren dargelegt werden.

## 3.3 Umsetzung des Konzeptes

Die Erreichung der in Tab. 3.1, S. 21 formulierten Arbeitsziele kann durch ein Werkzeug, welches speziell für die Aufgabe der *Kontinuierlichen Integration* entwickelt wurde, erheblich vereinfacht werden. Diese Form des Werkzeuges wird allgemein als *Continuous Integration Server*[5] bezeichnet. Dabei ist hervorzuheben, dass die Organisation des Prozesses, aufbauend auf dem Konzept von Martin Fowler, im Wesentlichen von diesem Werkzeug gesteuert wird. Aus diesem Grund kommt der Wahl des Werkzeuges große Bedeutung zu.

Nach der Wahl des Werkzeuges wird – um Fehler durch monotone Wiederholungen von Seiten des Testers in Zukunft zu vermeiden – die Testumgebung *sukzessiv* automatisiert. In diesem Zusammenhang soll die *sukzessive* Umstellung betont werden, weil die Entwicklung der Tests zu Beginn in einem hohen Grad von manuellen Eingriffen abhängig war und deshalb ein entsprechender Aufwand zur Adaption vorhandener Tests eingeplant werden musste.

Danach erfolgt – gewissermaßen als Abschluß – die Einbindung von Quellcodeanalysemetriken. Auf diese Weise soll eine Möglichkeit zur nachhaltigen Verbesserung der Qualität des Produktes geschaffen werden.

[5] Server für die Kontinuierliche Integration.

### 3.3.1 Auswahl des Werkzeuges

Am Anfang stand die Entscheidung für ein umfangreich integriertes Werkzeug, das die *Kontinuierliche Integration* begleiten sollte. Dabei ist die Wahl des Werkzeuges an *harte* und *weiche* Bedingungen gebunden (siehe dazu Tab. 3.2, S. 24).

| Harte Bedingungen | Weiche Bedingungen |
|---|---|
| Unterstützung von Subversion als CVS | einfache Installation |
| Weite Verbreitung | Ergonomie |
| geringe/keine Lizenzkosten | Erweiterbarkeit |

Tabelle 3.2: Bedingungen für die Wahl des Werkzeuges.

Der Unterschied zwischen diesen beiden Kategorien besteht darin, dass *harte* Bedingungen k e i n e Abweichung zulassen und somit vom Projekt vorgegeben sind. Die *weichen* Bedingungen erlauben hinsichtlich ihrer Erfüllbarkeit durchaus Abstriche und Toleranzen.

Im Weiteren einige Bemerkungen zu den *harten* Bedingungen.

Eine wesentliche Bedingung für den Auswahlprozess war zweifelsohne die Unterstützung von Subversion als SCM, da im Projekt die Versionskontrolle bereits damit organisiert wurde. Martin Fowler plaziert in der Liste der Voraussetzungen für die Umsetzung des gesamten Konzeptes das *SCM* an erster Stelle,. Damit betont er dessen zentrale Rolle.
Der Höhe der Lizenzkosten ist ebenfalls Bedeutung beizumessen, da sich dieses Projekt in der Anfangsphase befindet und keine nennenswerten Finanzmittel zur Verfügung stehen. Deshalb können teure Werkzeuge nicht zum Einsatz kommen.

Die *weichen* Bedingungen besitzen — im Vergleich zu den *harten* — für den Betrieb und Erfolg des Einsatzes des Werkzeuges einen etwas höheren Stellenwert. Weil die weichen Bedingungen die abschließenden Akzeptanz der Nutzer in Hinblick auf Bedienbarkeit und Aussehen darstellen und somit den langfristigen Einsatz etablieren.

Die Installation sollte nicht zu komplex sein. Im Fehlerfall könnte diese dann nicht reproduziert werden. Zudem kann auch der Fall eintreten, dass fehlerhafte Einstellungen den gestellten Anforderungen nicht gerecht werden. Das Design sollte annähernd aktuellen Gesichtspunkten der Oberflächengestaltung entsprechen, um durch ein aufgeräumtes Aussehen die Nutzer zu beeindrucken und zur Anwendung zu ermutigen. Obwohl die optische Gestaltung eines Teils der Ergonomie der Software einen hohen Stellenwert besitzt, kommt der haptischen Qualität ebenfalls eine große Bedeutung zu. Klar verständliche und logische Entscheidungspfade bei der Konfiguration der Projekte sollen Anpassungen *nach* der Installation nicht behindern.

*"Aus Sicht eines Software einsetzenden Unternehmens stehen vor allem wirtschaftliche Vorteile im Zentrum: Eine ergonomisch gestaltete Software führt zu einer erhöhten Arbeitsproduktivität, einer geringeren Fehlerquote, höherer Motivation der Mitarbeiter und zu geringeren Schulungs- und Betreuungskosten. (..) Aus Sicht des Benutzers können durch ergonomische Software psychische und physische Belastungen minimiert und damit vorzeitige Ermüdung sowie gesundheitliche Beeinträchtigungen vermieden werden."* (LB03, S.11). Das Bildschirmfoto in Abb. 3.2 soll diese Meinung bestätigen.

...Base... | ...EJB | ...Service | ...Test | Alle | MetaProjects | **Projekte** | extreme | Übersicht | +

| S | W | Job ↓ | Last Success | Last Failure | Last Duration | Console ↓ |
|---|---|---|---|---|---|---|
| | | AccountService | 20 hr (#144) | 15 days (#94) | 36 sec | |
| | | AccountServiceEJB | 20 hr (#127) | 2 days 0 hr (#121) | 41 sec | |
| | | | | | c | |
| | | | | | 47 sec | |
| | | | | | c | |
| | | BackendWrapperEJB | 15 days (#10) | 1 day 22 hr (#33) | 16 sec | |
| | | BaseTest | 17 hr (#243) | 1 day 23 hr (#236) | 36 sec | |
| | | de.idema.BaseService | 17 hr (#261) | 4 days 1 hr (#236) | 1 min 20 sec | |

| W | Description | % |
|---|---|---|
| | Number of pmd violations is 198 | 68 |
| | Build stability: No recent builds failed. | 100 |

Abbildung 3.2: Bedienoberfläche von Hudson im Browser (Hud09).

Im Rahmen der Recherchen fiel, die Entscheidung auf *Hudson*. Außerdem standen *Cruise Control*[6] und *Continuum*[7] als ernsthafte Alternativen in der engeren Wahl. Die Entscheidung war knapp und fiel auf *Hudson*, weil die Bedingungen in Tab. 3.2, S. 24 in vollem Umfang erfüllt waren und die Oberfläche einen besonders positiven Eindruck im Hinblick auf die einfache Bedienbarkeit erweckt hat[8].

Für die Zukunft stehen folgende Funktionen zur Verfügung stehen, die eine bereits absehbare notwendige Erweiterung erheblich vereinfachen:

- **Skalierung** durch die Möglichkeit einer Clusterbildung mit mehreren *Hudson*-Instanzen und die Einbindung von Amazon-EC2[9] und Hadoop[10].
- **Graphische Aufbereitung** der Analyseergebnisse von Qualitätssicherungswerkzeugen[11] in Form von Diagrammen und Hinweislisten.
- **Integration in die unternehmensweite Sicherheitsinfrastruktur** mittels LDAP.
- **Unterstützung für sehr viele Buildwerkzeuge**: Ant, Maven, Shell Skripte, Rake, Gant etc. (zur Zeit 30).
- **Unterstützung von über 20 verschiedenen Projektverwaltungswerkzeugen**, u.a. Bugzilla, JIRA, Trac, Redmine, Mantis.

Diese Funktionen sind derzeit noch als ungenutzes Potential anzusehen, bieten perspektivisch jedoch gute Möglichkeiten *Hudson* zu erweitern. Durch die fehlenden Limitierungen besteht darüber hinaus nicht die Gefahr eines *Lock-in* Effektes[12].

[6] http://cruisecontrol.sourceforge.net/
[7] http://continuum.apache.org/
[8] Die Werkzeuge *Cruise Control* und *Continuum* wären durchaus auch geeignet und würden den gestellten Anforderungen in gleichem Maße entsprechen.
[9] Dienst, der infolge des Hypes um Cloud-Computing bekannt geworden ist. Er bietet Rechen- und Speicherkapazität in einer vorher nicht dagewesenen Form an.
[10] *"Hadoop ist ein Open-Source-Java-Framework für skalierbare, verteilt arbeitende Software"* (Wik).
[11] Hiermit sind Werkzeuge zur statischen Quellcodeanalyse und Testabdeckung gemeint.
[12] *"In den Wirtschaftswissenschaften werden als Lock-in-Effekt (von to lock in: einschließen, einsperren) Kosten bezeichnet, die eine Änderung der aktuellen Situation unwirtschaftlich machen.(...)" (Wik10b).*

### 3.3.2 Integration der Projekte in Hudson

Das Einbinden der einzelnen Projekte in Hudson (siehe Abb. 3.3) setzt den Einsatz eines Buildwerkzeuges voraus. Buildwerkzeuge sind *der* Teil der Verarbeitungskette des Kompiliervorganges, bei welchem die Abhängigkeiten des zu kompilierenden Projektes vor der Ausführung des eigentlichen Programmiersprachencompilers aufgelöst werden. Aus der sich entwickelnden Komplexität erfolgt die Integration der Projekte in zwei Stufen:

1. Die erste Stufe sieht vor, den Kompilierprozess, die Verteilung der Software und die Ausführung der Tests mit einem Buildwerkzeug im Einzelnen zu automatisieren.

2. In der zweiten Stufe werden diese einzelnen Automatismen zusammengefaßt und als vollständige Prozesse in Hudson integriert.

Abbildung 3.3: Bedienoberfläche von Hudson mit integrierten Projekten.

Aufgrund der vorhandenen Kenntnisse von *Make*[13] fiel die Wahl auf *Ant*[14]. *Ant* folgt dem gleichen Konzept wie *Make*, indem es "Ziele" und abhängige "Ziele" definiert. Im Gegensatz zu *Make* erweitert *Ant* die Beschreibung der Konfiguration um XML und die Formulierung von Befehlsfolgen in Form von *Ant*-Aufgaben. Hinsichtlich der Zielsetzung werden die Anweisungen als *Ant*-Aufgaben formuliert. Das Konzept, so läßt sich zusammenfassend feststellen, beinhaltet die Beschreibung von Aufgaben und die Formulierung von Abhängigkeiten. In Abb. 3.4 ist die erforderliche Erstellung einer EJB-Teilkomponente[15] des EAR[16] mithilfe dieser *Ant*-Aufgaben schematisch dargestellt.

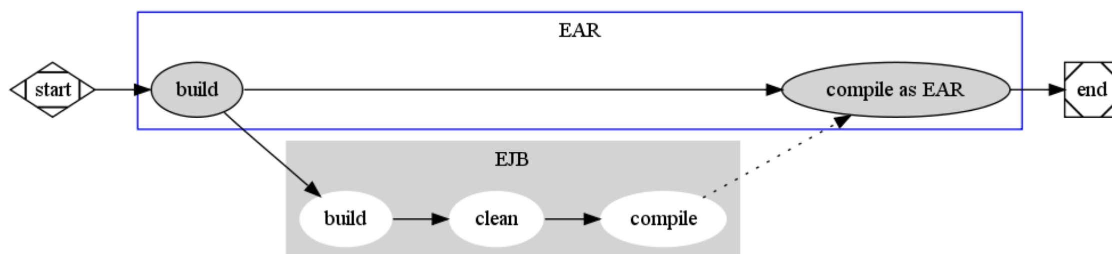


Abbildung 3.4: Schematische Darstellung der Kompilierung eines EAR-Archives mit Hilfe von *Ant*.

Aufgrund der erhöhten Komplexität – hervorgerufen durch mehrstufige Abhängigkeiten der Projekte untereinander (siehe Beispiel Abb. 3.5, S. 29) – war es nach einiger Zeit nicht mehr möglich, die Abhängigkeiten über interne Konventionen mit einem Antskript aufzulösen. Zum anderen ergab sich durch die Verwendung von *Eclipse* als Entwicklungsumgebung[17] – und der damit verbundenen Konfiguration innerhalb dieser Umgebung – das Problem, ein Projekt an zwei verschiedenen Stellen auf verschiedene Art und Weise zu konfigurieren. Somit entstünde potentiell eine neue Fehlerquelle und würde der Durchführung von Tests mit Hudson notwendiges

[13] `http://www.antmake.de/`
[14] `http://ant.apache.org/`
[15] Enterprise Java Bean
[16] Abkürzung für engl.: Enterprise Archive. EAR-Archive sind im JavaEE Umfeld ein übliches Format, um Anwendungen in einer Datei auszuliefern. In diesen Archiven können zusätzliche Bibliotheken in Form von Jars oder Komponenten in Form von EJB zusammengefaßt sein.
[17] Auch als IDE bezeichnet, engl.: Integrated Development Environment.

Vertrauen entziehen, weil Konfigurationsfehler zu oft Ursache auftretender Schwierigkeiten wären. Deshalb wurde die Entscheidung getroffen, eine angepaßte Ant-Aufgabe zu erstellen.

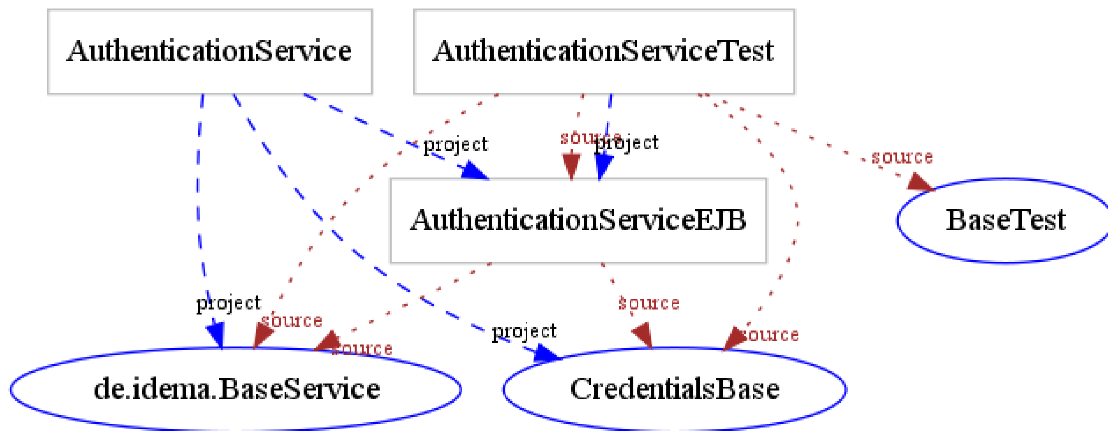


Abbildung 3.5: Ausschnitt des Abhängigkeitsgraphens mit dem Radius 1 um die Projekte der "Authentication"-Gruppe.

Das Auslesen der Eclipsekonfiguration ist durch eine einfache Abbildung der XML-Strukturen aus den Projektkonfigurationsdateien in Java-Datenstrukturen möglich. Der daraus entstehende Konfigurationsbaum – mit den hierarchischen Strukturen innerhalb der Projekte – bildet die Grundlage für die Erkennung von Abhängigkeitsbeziehungen. Aufgrund dieser Beziehungen können benötigte Projekte gezielt für die Kompilierung ausgewählt werden.

Der Kompilierprozess konnte durch das Aussortieren von Wiederholungen optimiert werden. Sollten zwei Projekte `P1` und `P2` von einem gleichen dritten Projekt `P3` abhängig sein (siehe Abb. 3.6, S. 30), hätte sich aus dem trivialen Abhängigkeitsbaum folgende Situation ergeben: Es wäre zuerst `P3` erstellt worden, erst dann `P1` und danach für `P2` das Projekt `P3` erneut (siehe Abb. 3.7, S. 30).

Das vorliegende Projekt enthält ein großes Basisprojekt, viele kleine Fach- und einige Unterfachprojekte. Allen gemeinsam ist die Abhängigkeit vom großen Basisprojekt.

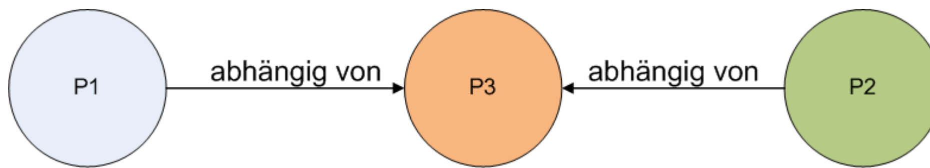


Abbildung 3.6: Darstellung des Abhängigkeitsgraphen der Projekte P1, P2 vom Basisprojekt P3.

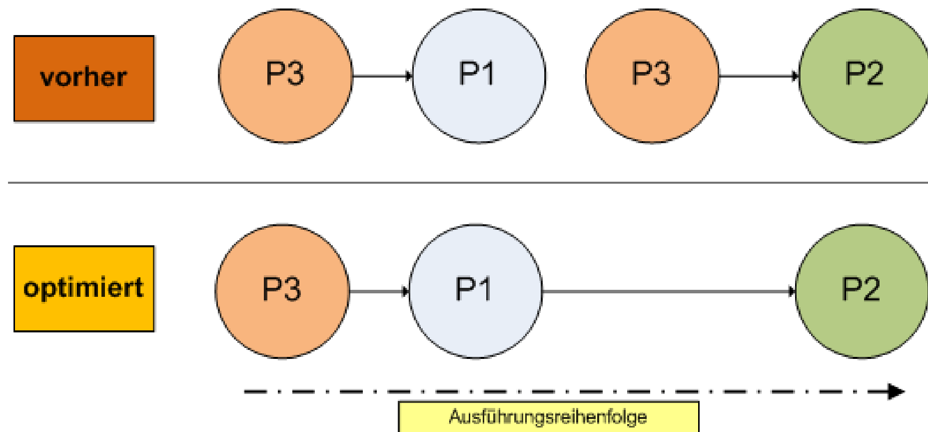


Abbildung 3.7: Gegenüberstellung der trivialen und der optimierten Abarbeitungsreihenfolge. Dabei wurde bereits bei diesem einfachen Beispiel die einfache Kompilierzeit von P3 eingespart.

Die Kompilierzeit der trivialen Variante läßt sich in folgender Formel[18] mit der Zeit T (vereinfacht für drei Abhängigkeitsebenen) darstellen:

$$T_{gesamttrivial} = T_{Fachprojekte} * T_{Unterfachprojekte} * T_{Basisprojekt}$$

Es ist leicht nachvollziehbar, dass jede Erweiterung des Projektes mit neuen Fachprojekten zu einer Potenzierung der Kompilierzeit führt. Diese Potenzierung bedeutet jedoch eine Verlangsamung und steht im Gegensatz zur Forderung *"Keep the build fast"* (siehe Abschnitt 2.2, S. 14). Es genügt durchaus, jedes der Projekte *einmal* zu kompilieren. Ausgehend von dieser Überlegung und auf der Grundlage des *topologischen Sortierens*[19] läßt

[18] Diese Formel veranschaulicht die klar getrennten Hierarchieebenen. Hierbei sind Querabhängigkeiten und Überlappungen ausgeschlossen. Es soll nur eine prinzipielle Annäherung an die Realität gezeigt werden.

[19] http://en.wikipedia.org/wiki/Topological_sorting

sich der Aufwand wie folgt reduzieren (siehe auch Abb. 3.7, S. 30):

$$T_{gesamtoptimiert} = T_{Fachprojekte} + T_{Unterfachprojekte} + T_{Basisprojekt}$$

Auf diese Weise kann durch Optimierung des Kompiliervorganges der Aufwand auf das notwenige Maß reduziert und im gleichen Zuge die Akzeptanz unter den Entwicklern erhöht werden.

### 3.3.3 Einbindung von Analysewerkzeugen

Im Zuge der Automatisierung der Integrationstests ist die Einbindung ergänzender Analysewerkzeuge in Betracht gezogen worden. Hierbei kommen Werkzeuge aus der statischen Quellcodeanalyse und deren Umfeld zur Anwendung.

Die *statische Quellcodeanalyse* ist Teil der falsifizierenden Whitebox-Software-Testverfahren. Sie untersucht den vorliegenden Quelltext nach Fehlern. Außerdem ist die statische Quellcodeanalyse, neben expliziten Modultests, die Grundlage für die Festlegung von Maßnahmen zur weiteren Verbesserung der Qualität. Eine kleine Übersicht über die Aufgaben ausgewählter Analysewerkzeuge[20] beinhaltet Tab. 3.3.

| Name | Beschreibung |
|---|---|
| *findbugs* | Sucht in Java Quelltext nach Fehlermustern. |
| *pmd* | Untersucht Java Quelltext auf Fehlermuster und prüft die Einhaltung von Regeln. |
| *cpd* | Spürt redundante Quelltextpassagen auf. |
| *checkstyle* | Überprüft Java Quelltext auf die Einhaltung von Programmierrichtlinien. |

Tabelle 3.3: Übersicht im Einsatz befindlicher Werkzeuge der statischen Quellcodeanalyse in Hudson.

[20]Die Werkzeuge sind als Erweiterungen in Form von Plugins eingebunden. Darüber hinaus stehen viele weitere zur Verfügung. `http://wiki.hudson-ci.org/display/HUDSON/Plugins`.

Die aus den Analysewerkzeugen abgeleiteten Metriken ergeben mit entprechender Trendanalyse einen Indikator für die weitere Entwicklung des Projektes. Abb. 3.8 veranschaulicht eine aus Hudson entnommene und als Beispiel dienende Zusammenfassung durchgeführter Überprüfungen bzw. Untersuchungen. Die entsprechenden Ergebnisse dienen einer späteren Feinjustage der einzelnen Metriken bzw. der Erkennung formaler Schwachstellen in den einzelnen Builds.

Es existiert eine Vielzahl von Metriken, die sich mithilfe der genannten Werkzeugen errechnen lassen. Kritisch ist dabei anzumerken, dass Metriken keine Gewähr für die gute *Qualität* von Software sind – sie gelten nur als Indizien. So kann z.B. ein hoher Prozentsatz an Testabdeckung ein Indiz für gute Qualität sein. Dass diese Tests aber auch die kritischen Stellen des Produkts betreffen, ist damit nicht belegt.

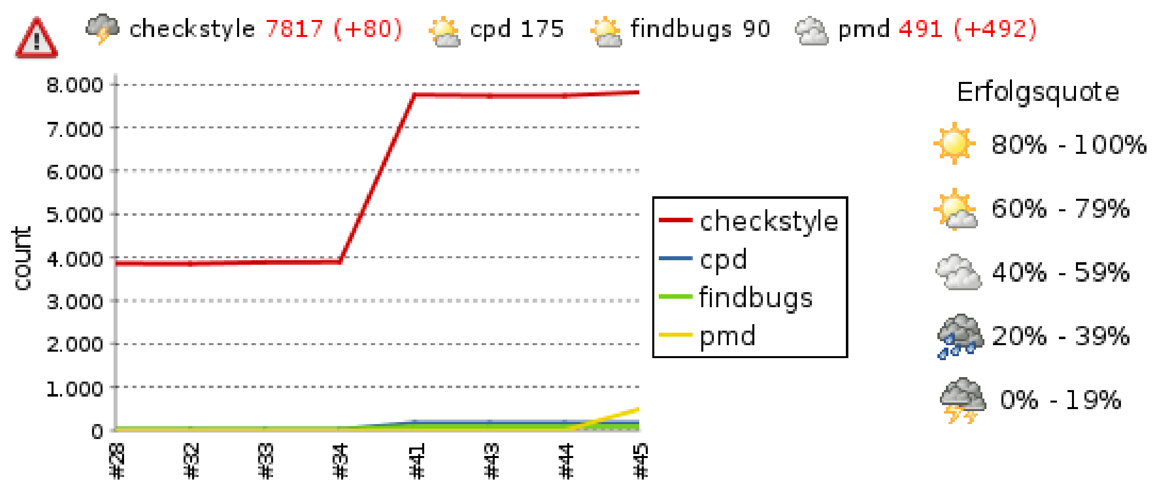


Abbildung 3.8: Akkumulierte Übersicht der Anzahl der Vorkommnisse der Hudson-Plugins (*checkstyle*, *cpd* , *findbugs* , *pmd*) im zeitlichen Verlauf der Kompiliervorgänge.

Werkzeuge, die im Umfeld der statischen Quellcodeanalyse anzusiedeln sind, weil sie nach formalen Aspekten handeln, sind folgende: Codezeilenzählung, Codestilüberprüfung, Zählung der Testabdeckung (Bereich der Laufzeitanalyse) und Coderedundanzprüfung.

Die beiden bekanntesten Vetreter der statischen Quelltextanalyse im Java-Bereich sind zweifelsohne FindBugs[21] und PMD[22]. Beide Werkzeuge sortieren entsprechende Fehler in Kategorien bzw. anhand von Regellisten ein, umso den manuellen Analyseprozess zu erleichtern (siehe Tab. 3.4).

| Findbugs-Kategorien[1] | PMD-Regellisten[2] |
|---|---|
| Schlechte Praxis | Basic, Braces, Code Size, Naming, Design |
| Korrektheit | String and StringBuffer, Type Resolution, Strict Exception, Import Statement, Finalizer, Migration |
| Experimentell | Controversial |
| Leistung | Optimization |
| Sicherheit | Security Code Guidelines |
| Fragwürdigkeit | Coupling, Unused Code |

[1] Internationalisierung, Verwundbarkeit und Multithread-Korrektheit sind mangels korrespondierender PMD-Regeln bei der Gegenüberstellung nicht berücksichtigt.
[2] Regeln zu Framework-Konventionen und zur Migration sind nicht genannt.

Tabelle 3.4: Gegenüberstellung der Fehlerkategorien von Findbugs und der Regellisten von PMD.

Im Folgenden werden mehrere Beispiele für das Auftreten von Fehler angeführt, die ohne des Einsatzes von *Hudson* und größeren Anstrengungen nicht zu finden wären.

In Abb. 3.9 ist der Hinweis auf einen Fehler abgebildet, der, um ihn zu finden, viel Zeit und Erfahrung der Entwickler in Anspruch nehmen würde. Es handelt sich hierbei um die Verletzung des Vertrages, der an die Nutzung und Implementierung der *clone()*-Methode gekoppelt ist. In diesem Fall wurde die Kovarianz[23] in der Klassenhierarchie nicht berücksichtigt (Ull09, Kapitel 10.2.4.).
Ein weiterer, durchaus nicht trivialer Fehler, welcher leicht unentdeckt bleibt, ist der nicht korrekte Vergleich zwischen Objekten (siehe Abb. 3.10).

[21] `http://findbugs.sourceforge.net/bugDescriptions.html` – 20.10.2010
[22] `http://pmd.sourceforge.net/rules/index.html` – 20.10.2010
[23] "In der objektorientierten Programmierung bedeutet Kovarianz (...), ob ein Aspekt gleichartig der Vererbungsrichtung (kovariant) (...) ist" (Wik10a).

File: AddressInformation.java, Line: 28, Type: CN_IDIOM_NO_SUPER_CALL, Priority: High, Category: BAD_PRACTICE

CN: de.idema.entity.Profile.AddressInformation.clone() does not call super.clone()

This non-final class defines a clone() method that does not call super.clone(). If this class ("*A*") is extended by a subclass ("*B*"), and the subclass *B* calls super.clone(), then it is likely that *B*'s clone() method will return an object of type *A*, which violates the standard contract for clone().

If all clone() methods call super.clone(), then they are guaranteed to use Object.clone(), which always returns an object of the correct type.

Abbildung 3.9: Hinweis zur fehlerhaften *clone()*-Methode.

So würde im ersten Teil dieses Vergleichs die Identität beider Objekte, nicht jedoch, wie beabsichtigt, deren Wertäquivalenz verglichen werden.

RC: Suspicious comparison of Long references in de.idema.entity.Credentials.Credential.isSame(Credential)

This method compares two reference values using the == or != operator, where the correct way to compare instances of this type is generally with the equals() method. It is possible to create distinct instances that are equal but do not compare as == since they are different objects. Examples of classes which should generally not be compared by reference are java.lang.Integer, java.lang.Float, etc.

```
174    public boolean isSame(Credential credential)
175    {
176        return credential.getId() == id && equals(credential);
177    }
```

Abbildung 3.10: Fehlerhafter Vergleich von Referenzendatentypen.

Die Beispiele in Abb. 3.9 und in Abb. 3.10 verdeutlichen, dass der Einsatz der statischen Quellcodeanalyse wichtig ist. Er ist – allein schon durch die Tatsache, dass viele Werkzeuge inzwischen Allgemeingut und frei zugänglich sind – eine dringende Notwendigkeit. Der Gründlichkeit und der Gewissenhaftigkeit der Entwickler zum Trotz kann so auch unter Zeit- und Kostendruck Qualität der Arbeit garantiert werden.

Weitere Werkzeuge aus der statischen Quellcodeanalyse und deren Umfeld, die während der Recherchen Interesse hervorgerufen haben und daher Erwähnung finden sollten, jedoch aus zeitlichen oder organisatorischen Gründen nicht untersucht werden konnten, sind – wie bereits

erwähnt – die Zählung der Testabdeckung[24], die Softwaretelemetrie[25] und die Softwarearchitekturüberwachung[26].

Zur Einbindung weiterer Werkzeuge stehen viele Plugins für Hudson[27], ja sogar der Quelltext[28]„ zur Verfügung, um so fehlende Werkzeuge anzubinden. Es wurde gezeigt, dass mit *Hudson* der Einsatz dieser Werkzeuge erheblich erleichtert wird und somit vorhandenes Potential in der Qualitätsanalyse ohne zusätzlichen Mehraufwand im laufenden Betrieb genutzt werden kann.

### 3.3.4 Einbinden der Entwickler

Der Einsatz neuer Werkzeuge bei der Entwicklung eines Produktes macht die Mitarbeit der Entwickler notwendig. Durch ausführliche Demonstration vieler Möglichkeiten und wiederholtes Aufzeigen von Zeitersparnis durch den Einsatz der Werkzeuge kann deren Nutzung regelrecht trainiert werden.

### 3.3.5 Aufwand

Die Anwendung in der Praxis erforderte die Einarbeitung in Hudson, in Ant als Buildwerkzeug, in die Auslieferungsvorgänge unterschiedlicher Java-Archivformate[29] und in die Funktionsweise des Hotdeployments[30] beim JBoss. Wie sich der Aufwand auf die Lösung der einzelnen Aufgaben verteilte, geht aus der folgenden Übersicht hervor (siehe dazu Tab. 3.5, S. 36).

[24]engl.: code coverage

[25]Mithilfe von Sensoren den Entwicklungsprozess visualisieren, siehe dazu auch `http://code.google.com/p/hackystat/`.

[26]`http://www.hello2morrow.com/products/sonarj`

[27]`http://wiki.hudson-ci.org/display/HUDSON/Plugins`.

[28]`http://wiki.hudson-ci.org/display/HUDSON/Source+code`

[29]Diese Archivformate sind *JAR*, *EAR* und *PAR*. *EAR* ist das Format für die Auslieferung eines Enterprise Java Projektes in den JBOSS. *PAR* ist das Archiv für die Auslieferung einer Prozessbeschreibung über die jBPM-Webconsole.

[30]Das *Hotdeployment* beschreibt den Vorgang, dass Code während der Ausführung verändert bzw. aktualisiert wird und dadurch Stillstandzeiten auf ein Minimum reduziert werden können.

| Aufgabe | Zeit in Tagen |
|---|---|
| Recherche des CI-Servers | 3 |
| Einarbeitung in Hudson | 5 |
| Einarbeitung in Ant | 5 |
| Einarbeitung in Archivformate | 2 |
| Analyse des Hotdeployments beim JBoss | 2 |
| Anfertigen der Ant-Skripte | 40 |
| Werkzeuge, die bereits bekannt waren | |
| Subversion (SCM) | 30 |
| Trac (Änderungsverfolgung) | 10 |

Tabelle 3.5: Verteilung des zeitlichen Aufwandes bei der Umsetzung in die Praxis.

Es ist anzumerken, dass es sich bei diesen Werten um akkumulierte Zeiten handelt. Der anfänglich schnelle Beginn verzögerte sich im weiteren Verlauf durch das Auftreten von Problemen. Dies ist auch der Grund für die Erreichung so hoher Zeitwerte.

## 3.4 Abschließende Einschätzung

Nach der Umsetzung des Konzeptes in die Praxis, hat sich der Arbeitsfluß verändert. Vor seiner Einführung mußte außer dem Entwickler eine zweite Person alle Tests manuell ausführen. Nach der Einführung des Konzeptes wird die Ausführung aller Tests direkt nach einem Commit automatisch angestoßen und ist nach kurzer Zeit abgeschlossen. Der Entwickler erfährt nach einer Wartezeit von nicht einmal fünf Minuten das Ergebnis seiner Arbeit. Er kann somit sofort auf Probleme, die durch seine letzten Änderungen ausgelöst wurden, reagieren. Die lückenlose Einbindung von Hudson in den gesamten Entwicklungsprozess hat dazu geführt, dass dieses Konzept in der täglichen Arbeit Anwendung findet, und die Qualität des Produktes signifikant gestiegen ist, änderungsbedingte Fehler können zeitnah gefunden und behoben werden konnten. In Abb. 3.11, S. 37 und in Abb. 3.12, S. 37 wird die Umgestaltung des Entwicklungsprozesses noch einmal veranschaulicht.

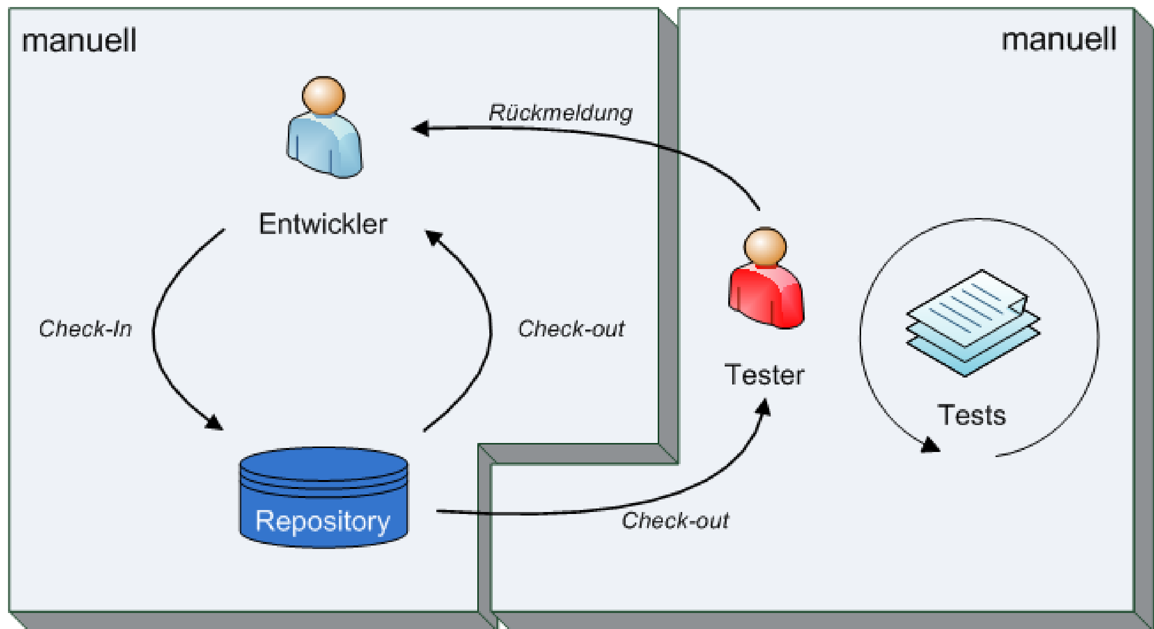


Abbildung 3.11: Der Arbeitsfluss vor der Umsetzung des Konzeptes.

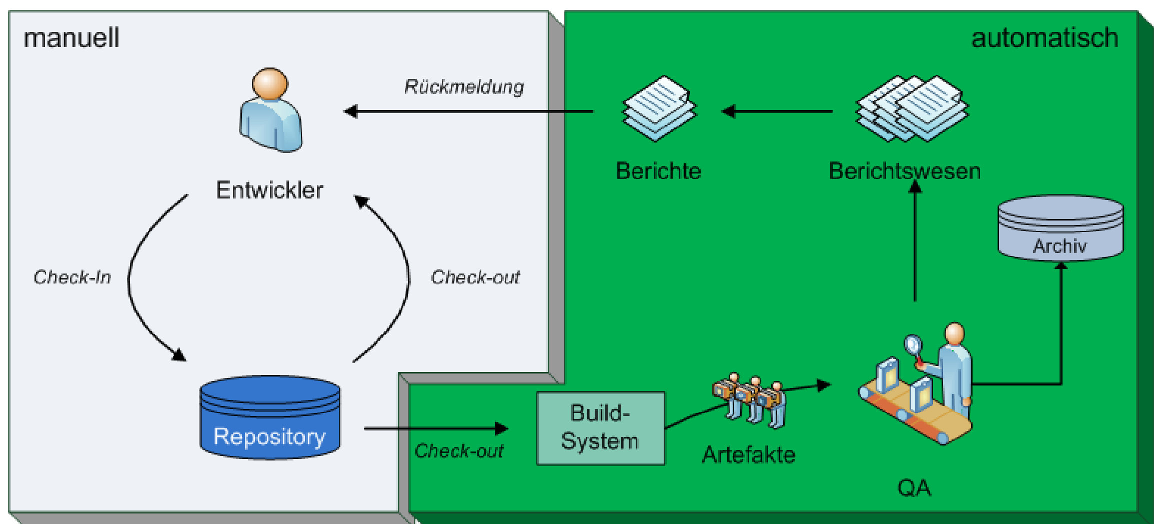


Abbildung 3.12: Der Arbeitsfluß auf der Grundlage des Konzeptes der *Kontinuierlichen Integration*.

# 4 Zusammenfassung

### Zusammenfassung und Auswertung

Im Rahmen der Arbeit wurde das Konzept der *Kontinuierlichen Integration* auf seine Anwendbarkeit in der Praxis untersucht. Als Basis für die softwaretechnische Unterstützung wurde *Hudson* als Werkzeug ausgewählt und eingesetzt. Es wurde - soweit wie möglich - versucht, das Konzept in das Projekt "Cidas$^2$" zu integrieren, um auf diese Weise wesentliche Verbesserungen zu erzielen. Das Konzept sieht vor, den Entwicklungsprozess in seiner Implementierungsphase zu jeder Zeit auf einem qualitativ hohen Stand zu halten. Auf diese Weise sollen die Phasen eines instabilen Produktes nahezu eliminiert werden.

Zu Beginn wurden Änderungen bereits in ein zentrales SCM in die Hauptentwicklungslinie übertragen, so dass jeder der Entwickler diese wahrnehmen konnte. Die vorliegenden Tests wurden manuell durchgeführt. Aus diesem Grunde stellte sich die Automatisierung der Tests und des Buildprozesses als zentrales Problem dar. Der Buildprozess musste für die Entwickler auf einfache Art und Weise automatisiert werden. Die Komplexität des Buildprozesses besteht in der Auflösung von Abhängigkeiten und der Nutzung von bestehenden Konfigurationsstrukturen der Eclipse-Entwicklungsumgebung, um dadurch zusätzlichen Konfigurationsaufwand zu vermeiden. Zu diesem Zweck wurde Ant als Build-Werkzeug ausgewählt. Weil Ant keine ausreichende Integration von Eclipse-Konfigurationen anbietet, mußte diese Aufgabe separat in Form von Ant-Tasks gelöst werden. Dies begründet den Aufwand, der in Ant floss.

Daraus leitete sich die Aufgabe ab, die Verpackung in einem EAR und die Auslieferung in einen JBoss Applikationsserver mit Hilfe von Ant zu lösen. Nachdem der Buildprozess erfolgreich implementiert war, mußte ein

geeigneter Integrationsservers ausgewählt werden, um den abschließenden Schritt in die *Kontinuerliche Integration* vollziehen zu können. Dafür wurde Hudson eingesetzt.

Das kontinuierliche Integrieren begleitet den Softwareerstellungsprozess über die gesamte Phase der Implementierung. Mit Hudson – als zentrales Werkzeug für *Kontinuierliche Integration* – ergaben sich erhebliche Verbesserungen (siehe dazu Abschnitt 3.4, S. 36).

So wurden z.B. Fehler bereits wenige Minuten nach einer Übertragung in das SCM erkannt und behoben. Vor dem Einsatz von Hudson vergingen jedoch mehrere Stunden bevor man auf diesen Fehler aufmerksam wurde. Die Integrationsphasen waren in die Entwicklung eingebettet, weil nach jeder Änderung das System vollständig getestet wurde. Dadurch konnte sofort auf Fehler reagiert werden.

Zusammenfassend läßt sich feststellen, dass *Kontinuierliche Integration* als Konzept bereits bekannte Techniken der Softwareentwicklung kombiniert und organisatorisch verbindet (siehe dazu Abschnitt 2.1, S. 7ff). Die Techniken sind den Entwicklern vertraut und lassen sich verhältnismäßig leicht umsetzen. Durch die Tatsache, dass die Vorgehensweise des Konzeptes in die Arbeit des Entwicklers nicht eingreift, sondern ihn zusätzlich unterstützt und dabei sogar entlastet (siehe Abb. 3.12, S. 37), wurde es bereits nach kurzer Zeit akzeptiert. Deshalb wird es auch regelmäßig genutzt (Fre09).

### Abschließende Einschätzung

Die *Kontinuierliche Integration* hat sich im Alltag als geeignet erwiesen. Dieses Konzept, auf dessen Grundlage gleichzeitig eine Qualitätssicherung hinsichtlich der Früherkennung von Fehlern erreicht wurde, hat sich in der Praxis voll bewährt. Der Aufwand der Installation ist überschaubar, die Lernkurve niedrig und die eingesparten Kosten infolge frühzeitig erkannter und gelöster Probleme (Tec09) nachweisbar (siehe Abb. 4.1, S. 41).

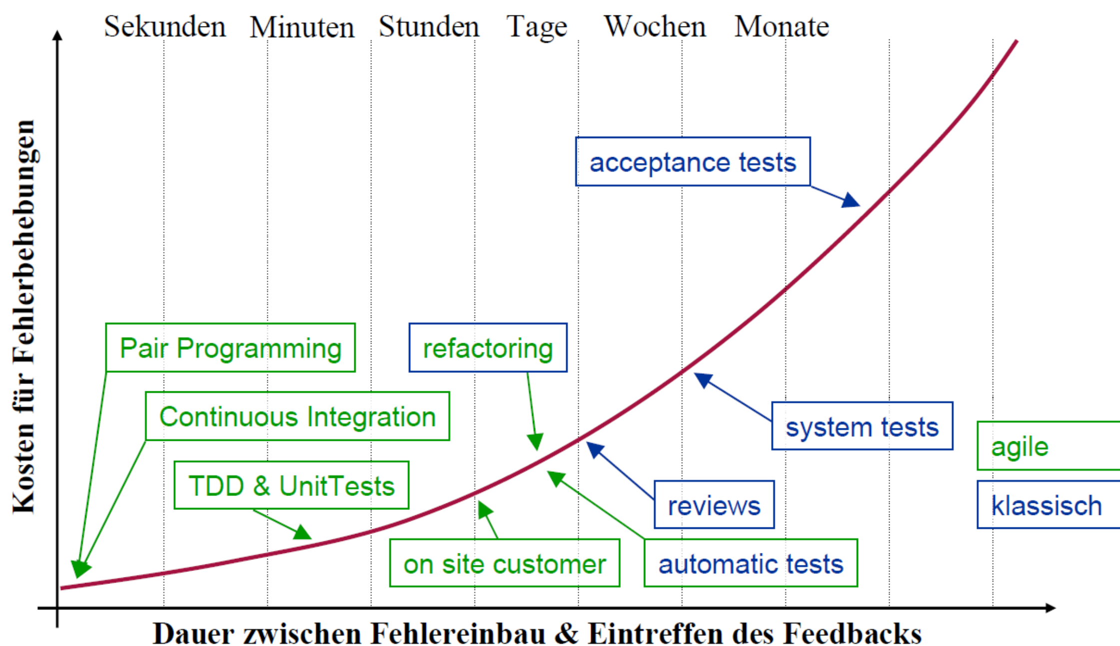


Abbildung 4.1: Darstellung der Kostenentwicklung im Verlauf des Entwicklungsprozesses (Die07).

Nicht allein diese Ergebnisse bestätigen die Machbarkeit dieses Konzeptes in der Praxis.

#### Ausblick

In Zukunft wird es möglich sein, den Buildprozess zu skalieren und die Last vieler Projekte zu *clustern*. Hudson bietet u.a. die Möglichkeit, Amazons *ElasticCloud*-Instanzen einzubinden. Außerdem könnte ein umfassendes Authentifizierungskonzept mithilfe eines zentralen Authentifizierungsserver umgesetzt werden. Dafür kämme z.B. LDAP in Frage. Sollte es sich als notwendig erweisen, ein anderes Projektverwaltungswerkzeug als das in diesem Fall verwendete *Redmine* einzusetzen, stünden 19 andere zur Verfügung. Gleichfalls kann sich durch die Nutzung externer Projekte die Notwendigkeit ergeben, ein zusätzliches Buildwerkzeug, z.B. *Maven*, zu verwenden.

Wie aus den Ausführungen hervorgeht, stellt der Einsatz von *Hudson* eine für die Zukunft durchaus sinnvolle Strategie dar.

# Tabellenverzeichnis

3.1 Status der Arbeitsziele (auf den Praktiken von 2.2, S. 11ff. aufbauend). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.2 Bedingungen für die Wahl des Werkzeuges. . . . . . . . . . . . 24
3.3 Übersicht im Einsatz befindlicher Werkzeuge der statischen Quellcodeanalyse in Hudson. . . . . . . . . . . . . . . . . . . . . . 31
3.4 Gegenüberstellung der Fehlerkategorien von Findbugs und der Regellisten von PMD. . . . . . . . . . . . . . . . . . . . . . . . 33
3.5 Verteilung des zeitlichen Aufwandes bei der Umsetzung in die Praxis. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

# Abbildungsverzeichnis


2.1 Gegenüberstellung der Kostenentwicklung hinsichtlich der Fehlerbehebung beim Einsatz eines *klassischen* im Vergleich zu einem *agilen* Vorgehensmodell[1] (Die07). . . . . . . . . . . . 10
2.2 Entwicklung der Kosten für die Fehlerbereinigung in Abhängigkeit zur Zeit und der Verteilung der entstandenen Fehler (Tec09). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3 Übertragung[2] in eine zentrale Quellcodeverwaltung. . . . . . 12
2.4 Empfohlener Arbeitsfluß des Entwicklers mit kontinuierlichen Integrationstests und Berichten über Erfolg- und Mißerfolg einzelner Testläufe. Hierbei ist wichtig, daß der automatische Teil des Prozesses in minimaler Zeit abläuft. . . . . . . . . . . . . . . 15
2.5 Übersicht über eine Änderung am Beispiel von *trac*[3]. . . . . . 16

3.1 Der Arbeitsfluss nach IST-Stand mit einer starken *manuellen* Ausprägung - verbunden mit hohem zeitlichen und personellen Aufwand. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.2 Bedienoberfläche von Hudson im Browser (Hud09). . . . . . . 25
3.3 Bedienoberfläche von Hudson mit integrierten Projekten. . . . 27
3.4 Schematische Darstellung der Kompilierung eines EAR-Archives mit Hilfe von *Ant*. . . . . . . . . . . . . . . . . . . . . . . . 28
3.5 Ausschnitt des Abhängigkeitsgraphens mit dem Radius 1 um die Projekte der "Authentication"-Gruppe. . . . . . . . . . . . 29
3.6 Darstellung des Abhängigkeitsgraphen der Projekte `P1`, `P2` vom Basisprojekt `P3`. . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.7 Gegenüberstellung der trivialen und der optimierten Abarbeitungsreihenfolge. Dabei wurde bereits bei diesem einfachen Beispiel die einfache Kompilierzeit von `P3` eingespart. . 30

3.8 Akkumulierte Übersicht der Anzahl der Vorkommnisse der Hudson-Plugins (*checkstyle*, *cpd* , *findbugs* , *pmd*) im zeitlichen Verlauf der Kompiliervorgänge. . . . . . . . . . . . . . . . 32
3.9 Hinweis zur fehlerhaften *clone()*-Methode. . . . . . . . . . . . . . 34
3.10 Fehlerhafter Vergleich von Referenzendatentypen. . . . . . . . 34
3.11 Der Arbeitsfluss vor der Umsetzung des Konzeptes. . . . . . . . . 37
3.12 Der Arbeitsfluß auf der Grundlage des Konzeptes der *Kontinuierlichen Integration*. . . . . . . . . . . . . . . . . . . . . . . . 37

4.1 Darstellung der Kostenentwicklung im Verlauf des Entwicklungsprozesses (Die07). . . . . . . . . . . . . . . . . . . . . . . . . . 41

# Quellen- und Literaturverzeichnis


(Die07) Sebastian Dietrich. Qualität durch agilität - qualitätsmanagement in der agilen software entwicklung, 2007. Available from World Wide Web: http://www.softwarequalitaet.at/pages/texte/22.%20Fachtagung/BeitrDietrich.pdf. 2009.11.16 12:53.

(DUD09) Duden, 2009. Available from World Wide Web: http://www.duden.de/definition/agil. 2010.01.04 10:30.

(Fow06) Martin Fowler. Continuous integration, 05 2006. Available from World Wide Web: http://www.martinfowler.com/articles/continuousIntegration.html. 2009.11.16 12:53.

(Fre09) Jürgen Freudig. Continuous integration und testautomatisierung mit open source, 2009. Available from World Wide Web: http://www.linuxtag.org/2009/fileadmin/www.linuxtag.org/vp/papers/Juergen_Freudig-Continuous%20Integration%20und%20Testautomatisierung%20auf%20Basis%20von%20Open-Source-LinuxTag2009-slides.odt. 2009.11.16 12:53.

(fTQM) Institute for Total Quality Management. Glossar. Available from World Wide Web: http://www.itqm.ch/index.php?sector=01&pg=38&language=de. 2010.01.04 11:05.

(Hal04) Sven Hall. Continuous build and test im projektalltag. *Javaspektrum*, 06:24–26, 2004. Available from World Wide Web: http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2004/06/hall_JS_06_04.pdf. 2009.11.16 15:01.

(Hud09) 2009. Available from World Wide Web: http://wiki.hudson-ci.org//download/attachments/753667/1.png. 2010.01.05 11:40.

(LB03) Wolfgang Schneider Lothar Bräutigam. *Projektleitfaden Software-Ergonomie Band 43*. Hessisches Ministerium für Wirtschaft, Verkehr und Landeswentwicklung, 2003.

(Lea09) Lean manufacturing, 11 2009. Available from World Wide Web: http://en.wikipedia.org/wiki/Lean_manufacturing. 2009.11.17 12:53.

(Pop09) Mary Poppendieck. Lean programming, 10 2009. Available from World Wide Web: http://www.poppendieck.com/lean.htm. 2009.11.17 12:53.

(ST09) Markus Wittwer Stefan Toth, Uwe Vigenschow. Einfluss klassischer und agiler techniken auf den erfolg von it-projekten, 5 2009. Available from World Wide Web: http://www.oose.de/fileadmin/Dateien/Publikationen/Ergebnisbericht_Projektmanagementstudie.pdf. 2009.11.17 12:53.

(Tec09) Agitar Technologies. Why unit testing?, 2009. Available from World Wide Web: http://www.agitar.com/solutions/why_unit_testing.html. 2009.11.16 13:07.

(Tho06) BT ThoughtWorks. Agile cookbook - a wiki guide to agile delivery in the 90 day cycle, 02 2006. Available from World Wide Web: http://agilecookbook.com/. 2009.11.16 12:53.

(u.a01) Kent Beck u.a. Manifesto for agile software development, 2001. Available from World Wide Web: http://www.agilemanifesto.org/. 2009.11.17 12:53.

(Ull09) Christian Ullenboom. Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6, 2009. Available from World Wide Web: http://openbook.galileocomputing.de/javainsel8/javainsel_10_002.htm#mj66a1b642fb021f6ed587757e308eb86f. 2009.11.14 14:00.

(Wik) Hadoop. Available from World Wide Web: http://de.wikipedia.org/wiki/Hadoop. 2010.01.05 15:13.

(Wik09a) Agile softwareentwicklung, 2009. Available from World Wide Web: http://de.wikipedia.org/wiki/Agile_Softwareentwicklung. 2009.11.17 12:53.

(wik09b) Stabilität, 11 2009. Available from World Wide Web: http://de.wikipedia.org/wiki/Stabilität. 2009.11.16 13:00.

(Wik10a) Wikipedia. Kovarianz und kontravarianz — wikipedia, die freie enzyklopädie, 2010. Available from World Wide Web: http://de.wikipedia.org/w/index.php?title=Kovarianz_und_Kontravarianz&oldid=69332846. Online; Stand 20. Januar 2010.

(Wik10b) Wikipedia. Lock-in-effekt — wikipedia, die freie enzyklopädie, 2010. Available from World Wide Web: http://de.wikipedia.org/w/index.php?title=Lock-in-Effekt&oldid=70392680. Online; Stand 22. Februar 2010.

(Wik10c) Wikipedia. Trac — wikipedia, die freie enzyklopädie, 2010. Available from World Wide Web: http://de.wikipedia.org/w/index.php?title=Trac&oldid=68852283. Online; Stand 20. Januar 2010.

# Erklärung zur Studienarbeit

Hiermit erkläre ich, die vorliegende Studienarbeit selbstständig und ohne fremde Hilfe verfasst zu haben. Auf die verwendeten Quellen habe ich in entsprechender Weise hingewiesen und diese im Literaturverzeichnis aufgeführt.

Brandenburg a. d. H., den 11. März 2010

Lars Gohlke