Nachhaltige Software-Entwicklung mit Open-Source-Tools und automatisierten Workflows

Klaus Quibeldey-Cirkel, Christoph Thelen

Fachbereich Mathematik, Naturwissenschaften und Informatik
Technische Hochschule Mittelhessen
Wiesenstr. 14
D-35390 Gießen
klaus.quibeldey-cirkel@mni.thm.de
christoph.thelen@mni.thm.de

Abstract: Durch automatisierte Abläufe, eine darin fest verankerte Qualitätssicherung und zahlreiche Feedback-Stufen lässt sich der Wartungsaufwand von Software deutlich verringern. Zum einen werden neue Entwickler schneller in das Projekt integriert, da sie einen roten Faden vorfinden. Zum anderen entlasten automatisierte Build-, QA- und Deployment-Prozesse die Hauptentwickler und Administratoren. Umgesetzt mit renommierten Open-Source-Werkzeugen sorgt dieser pragmatische Ansatz dafür, dass die Software-Entwicklung in Teams mit hoher Fluktuation nachhaltig, das heißt unter Wahrung der erreichten Softwaregüte erfolgen kann.

1 Einleitung

Der Begriff „Nachhaltige Entwicklung“[1] fokussiert auf die verantwortungsvolle Nutzung von Ressourcen. Die Ressourcen in der Software-Entwicklung sind vor allem die Entwickler und die Qualität der bislang entwickelten Software. Beide sind eng verknüpft: Werden die Entwickler-Ressourcen strapaziert, geht das meist auf Kosten der Qualitätssicherung; eine Verschlechterung der Qualität bindet die Entwickler an Wartungsaufgaben. Das kann dazu führen, dass Software entwickeln zu einer nervenaufreibenden Angelegenheit wird. Besonders dann, wenn das Projekt ein Altsystem ist, das einen gewissen Umfang überschritten hat, eine hohe Fluktuation an Entwicklern aufweist und nur wenige Hauptentwickler kontinuierlich daran arbeiten.

Müssen neue Entwickler in ein Projekt integriert werden, zum Beispiel bei Neueinstellungen, Umstrukturierungen oder Firmenzusammenschlüssen, haben die Hauptentwickler und Administratoren alle Hände voll zu tun, die Arbeit zu koordinieren und dafür zu sorgen, dass die Qualität und der Betrieb der Software durch Code-Änderungen nicht beeinträchtigt werden. Die eigentlichen Wartungsaufgaben bleiben dabei oft auf der Strecke. Dies ist auf Dauer dem Projekt nicht zuträglich und widerspricht dem Gedanken einer nachhaltigen, das heißt *qualitätserhaltenden* Software-Entwicklung.

[1] http://de.wikipedia.org/wiki/Nachhaltige_Entwicklung (Abruf 28.07.2012)

Die an der THM entwickelte und betriebene Kollaborationsplattform *eCollab*[2] stand vor der Herausforderung, neue Entwickler so schnell wie möglich in das Projekt zu integrieren. Die Entwickler sind Studierende aus Informatik-Lehrveranstaltungen, in denen die Plattform weiterentwickelt wird. Da sich die Weiterentwicklung am Semesterbetrieb orientiert, herrscht eine hohe Fluktuation an Entwicklern: Die meisten Studierenden sind nur wenige Monate in das Projekt involviert und scheiden am Ende des Semesters wieder aus. Somit bleibt für alle Beteiligten nur wenig Zeit, sich intensiv mit der Software auseinanderzusetzen. Die ständigen Projektmitarbeiter waren hauptsächlich damit beschäftigt, neue Entwickler in das Projekt einzuweisen und deren erste Änderungsversuche zu überprüfen.

Auf Grund dieser Problematik entstand im Laufe der Zeit mit Hilfe von Open-Source-Werkzeugen ein automatisierter Prozess. Er besteht aus mehreren miteinander verbundenen Abläufen (Abb. 1), die für ein kontinuierliches Feedback und eine automatisierte Qualitätssicherung (QA) sorgen. Der Prozess ist für jeden Entwickler als individueller Entwicklungsablauf zu verstehen und schafft so einen roten Faden, an dem er sich orientieren kann.

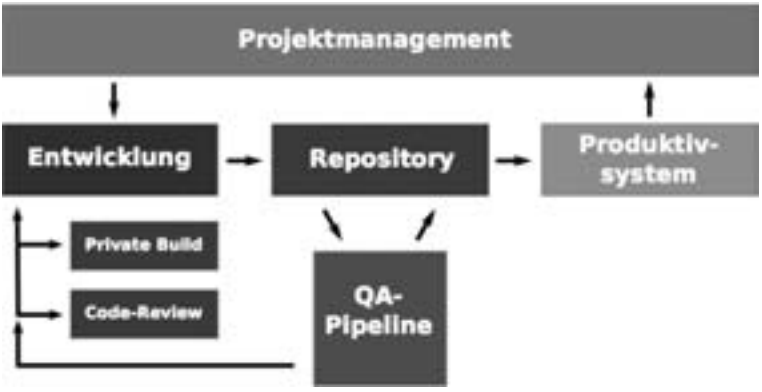

Abbildung 1: Entwicklungs-Workflow der Kollaborationsplattform *eCollab*

Konkret bringt der Prozess zwei Vorteile: Erstens wird neuen Entwicklern die Hand gereicht, sodass ihnen durch einen beinahe geführten Ablauf der Einstieg in das Projekt erleichtert wird. Zweitens sorgt die automatisierte Qualitätssicherung dafür, dass Fehler nicht in das produktive System gelangen. Die Hauptentwickler werden dabei gleichzeitig entlastet, da sie den Ablauf nicht überwachen müssen. Des Weiteren ist im Prozess die Möglichkeit vorgesehen, vor der Qualitätssicherung bei Bedarf ein formales Code-Review durchzuführen, von dem die Entwickler zusätzlich profitieren können.

Dies ist es, was unter nachhaltiger Software-Entwicklung zu verstehen ist: Die Software bleibt dauerhaft und für alle Beteiligten in einem beherrschbaren Zustand. Um diesen Zustand beizubehalten, ist eine ständige und möglichst umfassende Qualitätsüberwa-

[2] Die Plattform *eCollab* ist in Umfang (ca. 900.000 LOC) und Funktionalität vergleichbar mit den Lernplattformen *Moodle* und *ILIAS*, siehe die Gegenüberstellung unter https://ecollab.thm.de/infos/impressum.php.

chung notwendig. Sie beinhaltet neben den funktionalen Aspekten vor allem auch eine Bewertung der Design-Güte in Form von Metriken, anhand derer ein möglicher Qualitätsverfall sichtbar wird. Nur wenn sie automatisiert ist, kann sie an verschiedenen Stellen in den Prozess integriert werden: während der Entwicklung durch den *Private Build* auf dem lokalen System und später für alle in der *QA-Pipeline.* Letztere verhindert aktiv Raubbau am Software-Design, indem problematische Änderungen im *Repository* abgewiesen und zur Überarbeitung zurück an die Entwickler gesendet werden.

Für die Umsetzung des Prozesses kommen in der Open-Source-Welt bekannte Tools zum Einsatz: *Git* und *Gitorious* für die Versionsverwaltung aller Projektquellen und der Ablaufsteuerung sowie *Jenkins* als Continuous-Integration-Server für die automatisierte Qualitätssicherung. Mit *Redmine* steht außerdem ein Projektmanagement-Werkzeug mit Unterstützung der Scrum-Methode zur Verfügung.

2 Evolution des Quellcodes

Um die Nachhaltigkeit der Softwaregüte bewerten zu können, sind Software-Metriken erforderlich; sie liefern Fakten über die quantitativen und qualitativen Merkmale des Projekts. Am Beispiel der Kollaborationsplattform *eCollab* werden im Folgenden die Metriken zusammengetragen, die für eine Bewertung der Softwaregüte infrage kommen.

Der Online-Dienst *Ohloh* berechnet für Open-Source-Projekte aus den Repository-Daten der gesamten Projekthistorie (Quellcode und Commits) verschiedene quantitative Metriken. Diese geben einen Überblick, wie das Projekt während seiner Laufzeit gewachsen ist. Dabei wird zum einen auf die Code-Zeilen Bezug genommen, zum anderen aber auch auf die Aktivität der Entwickler. Beide Punkte werden verknüpft, um den Einfluss einzelner Personen auf das Projekt zu verdeutlichen:

- „Contributors“: Es wird der Prozentsatz der Commits berechnet, die einzelne Entwickler am Projekt insgesamt, in den letzten zwölf Monaten und in den letzten 30 Tagen getätigt haben (Abb. 2). Für *eCollab* ist interessant, dass 50% aller Commits von Entwicklern stammen, die nicht mehr in das Projekt involviert sind. Für die vergangenen zwölf Monate beträgt dieser Wert 25%.
- „Languages“: Übersichten der im gesamten Projektverlauf zum Einsatz kommenden Programmiersprachen, deren Anteil an der Gesamtgröße und welcher Art die Zeilen sind (Code, Kommentar, Leerzeile). Es wird auch ermittelt, welcher Entwickler welche Sprachen in welchem Umfang in das Projekt eingebracht hat.

Interessante qualitative Metriken liefern „Instability vs. Abstraction“ [Ma94] und die „Metriken-Pyramide“ [LM06], siehe Abb. 3 und Abb. 4. Im Vergleich zur Code-Analyse von *Ohloh* wird hier aber nicht die gesamte Projekthistorie, sondern der Status quo der Software berücksichtigt. Dies ist insofern als problematisch anzumerken, da sich die Nachhaltigkeit so nicht direkt erkennen lässt. Erst über eine Trend- bzw. Langzeitanalyse kann auf einen Blick geurteilt werden, ob die Maßnahmen letztlich zu einer Wahrung oder gar Verbesserung der Qualität führen.

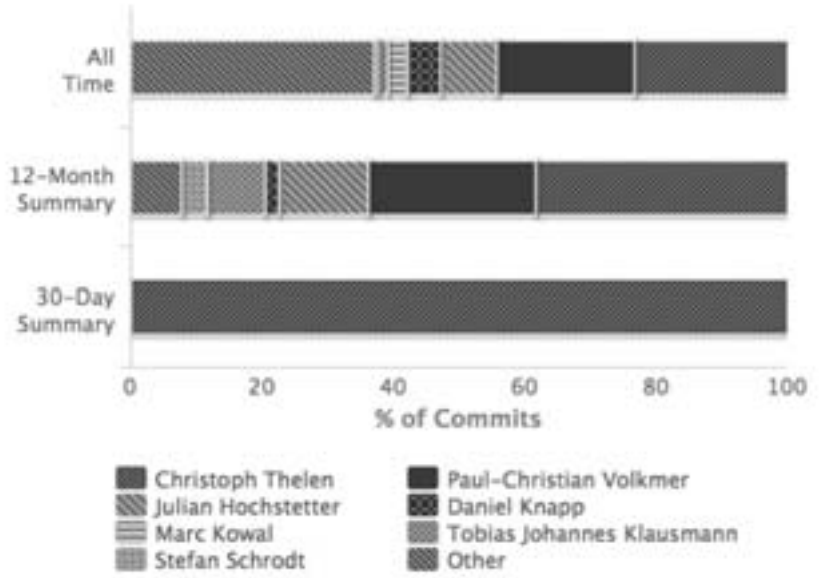

Abbildung 2: Anteil der Commits einzelner Entwickler am Gesamtprojekt.[3]

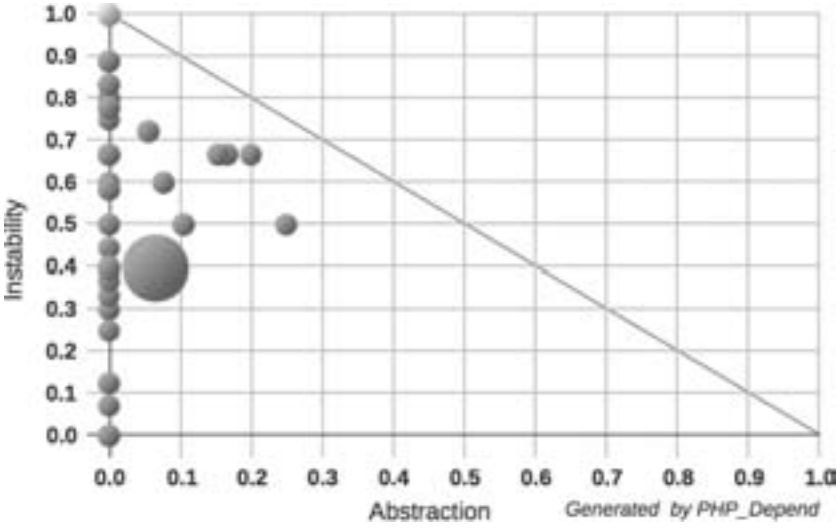

Abbildung 3: „Instability vs. Abstraction"-Diagramm nach [Ma94]

[3] https://www.ohloh.net/p/estudy/contributors/summary (Abruf 28.07.2012)

„Instability vs. Abstraction“ charakterisiert Pakete[4] anhand ihrer Instabilität (I), also wie stark sie an andere Pakete gekoppelt sind, und der Abstraktheit (A), das heißt, wie hoch der Anteil an abstrakten Klassen oder Schnittstellen innerhalb eines Pakets ist (Abb. 3). Berechnet wird die Instabilität anhand der eingehenden und ausgehenden Kopplung. Es wird eine Ideallinie (A+I=1) definiert, welche das beste Verhältnis aus Instabilität und Abstraktheit anzeigt. Je weiter sich die Pakete von der Ideallinie entfernen, desto anfälliger sind sie für Änderungen in anderen Paketen.

Ein wichtiger Aspekt der Nachhaltigkeit der Softwaregüte wird somit ausgedrückt: Werden Änderungen an einem Teil der Software durchgeführt, kann das Konsequenzen für andere Bereiche haben. Diese müssen dann ebenfalls geändert werden. Dadurch erhöht sich der Aufwand für Wartung und Qualitätssicherung.

Mehr ins Detail geht die Metriken-Pyramide (Abb. 4), welche drei strukturelle Aspekte der Software beleuchtet: Größe und Komplexität im linken Teil, Kopplung im rechten und Vererbung in der Spitze der Pyramide. Der linke Teil besteht zunächst erneut aus einer quantitativen Analyse, die allerdings viel tiefer geht. Hier findet eine Inventur der Software statt: Anzahl der Pakete oder Namensbereiche (NOP), Anzahl der Klassen (NOC), Anzahl der Methoden (NOM), Gesamtzahl der Code-Zeilen (LOC) und schließlich die zyklomatische Komplexität nach McCabe (CYCLO). Im rechten Teil wird die Anzahl der Methodenaufrufe (CALLS) und Kopplungen zu anderen Klassen (FANOUT) gezählt. Die Spitze bildet schließlich die durchschnittliche Anzahl von abgeleiteten Klassen (ANDC) und die durchschnittliche Höhe der Klassenhierarchie (AHH). Die ermittelten Werte werden im inneren Teil der Pyramide abgelegt.

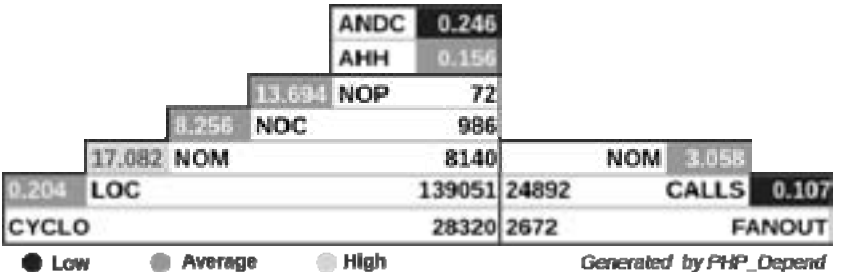

Abbildung 4: Metriken-Pyramide nach [LM06]: Jeweils zwei übereinander stehende Werte in der Mitte der Pyramide werden dividiert und das Ergebnis an der Kante des jeweils oberen Wertes niedergeschrieben. Zusammen mit der farblichen Klassifizierung sorgen diese Zahlen für projektübergreifend vergleichbare Werte.

Die Zählungen allein sind noch nichts Besonderes; zusätzlich wird an den Kanten der Pyramide das Verhältnis zweier aufeinanderfolgender Werte festgehalten, wobei jeweils der untere Wert der Dividend des nächsthöheren ist [LM06]. In Abb. 4 ist zum Beispiel

[4] Robert C. Martin spricht von „Kategorien“ und bezieht sich auf die Modellierungssprache von Grady Booch.

die zyklomatische Komplexität pro Zeile Code (CYCLO/LOC = 0,204), die Anzahl Methoden pro Klasse (NOM/NOC = 8,256) oder die Anzahl externer Kopplungen pro Methodenaufruf (FANOUT/CALLS = 0,107) zu sehen. Die Durchschnittswerte sind gemäß den Referenzwerten aus [LM06] farblich klassifiziert (Low/Average/High) und erlauben so die Vergleichbarkeit mit anderen Projekten, selbst wenn diese größer oder kleiner sind. Insgesamt gibt die Metriken-Pyramide einen Einblick in die Qualität der Code-Organisation.

3 Kontinuierliches Feedback

Für die Entwickler ist ständiges und individuelles Feedback wichtig, wenn sie schnell auf Probleme reagieren sollen. Fehler sollten daher möglichst schon bei oder kurz nach ihrer Entstehung entdeckt werden. Je mehr Zeit zwischen dem Einführen und dem Erkennen eines Fehlers liegt, desto schwieriger kann später auch dessen Korrektur sein, da sich die Entwickler möglicherweise bereits mit völlig anderen Aufgaben beschäftigen. Daher werden mehrere Feedback-Stufen definiert, an denen sich die Entwickler orientieren können (Abb. 5). Jede Stufe gibt den Entwicklern eine zusätzliche Absicherung, dass eventuell auftretende Probleme sofort aufgedeckt werden. Dabei sind die ersten Stufen noch auf dem Entwickler-PC angesiedelt, sodass Fehler bereits dort behoben werden können. Sie werden damit erst gar nicht an die übrigen Entwickler verteilt.

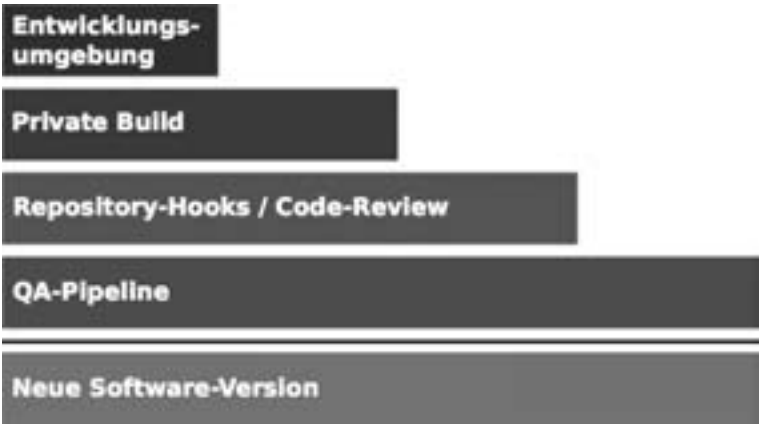

Abbildung 5: Feedback-Phasen im Entwicklungsprozess

Feedback wird auf verschiedenen Wegen von der automatisierten Qualitätssicherung zunächst nur an die beteiligten Entwickler, später im Entwicklungsprozess dann an das gesamte Team geliefert. Jede Stufe ist als Miniatur-Meilenstein in der Entwicklung anzusehen, sodass eine Code-Änderung stets überwacht wird, während sie den gesamten Entwicklungsprozess durchläuft.

Idealerweise werden Probleme bereits während der Arbeit in der Entwicklungsumgebung (IDE) markiert, sodass sie schon mit ihrer Entstehung verhindert werden können.

Typischerweise betrifft das Verstöße gegen die im Projekt vereinbarten Programmierrichtlinien und Fehlschläge der Unit-Tests.

Im zweiten Schritt hilft der „Private Build“, mögliche Integrationsfehler zu beheben: Der gesamte QA-Prozess kann auf dem eigenen Entwickler-PC ausgeführt und nachvollzogen werden. Dies geschieht kurz vor dem Check-in neuer Änderungen in das Entwickler-Repository.

Sollen die Code-Änderungen mit anderen Entwicklern ausgetauscht werden, müssen sie den im Projekt vereinbarten Vorgaben entsprechen. Sollten Verstöße zuvor in der IDE ignoriert worden sein, werden die Änderungen spätestens an dieser Stelle abgewiesen. Somit wird verhindert, dass von den Programmiervorgaben im Laufe der Zeit immer weiter abgewichen wird.

Haben die Änderungen schließlich ihren Weg in das Entwickler-Repository gefunden, wird die QA-Pipeline angestoßen. Diese bildet die letzte Instanz der Qualitätssicherung. Die Änderungen durchlaufen dabei mehrere automatisierte „QA-Schleusen“. Ist die Pipeline vollständig durchlaufen, entsteht ein Gesamtabbild der Anwendung, das für den produktiven Einsatz bereit ist.

4 Entwicklungsumgebung

Abseits der stark kommerzialisierten Sprachen wie Java oder C# gibt es nur eine sehr eingeschränkte Auswahl an Werkzeugen. Mit *Eclipse* gibt es zwar für die meisten Sprachen eine passende IDE, jedoch sind viele Operationen nicht verfügbar, die zum Beispiel in der *Java-Eclipse* oder auch in *Visual Studio* selbstverständlich sind. Dazu gehören in der Regel Standardaufgaben wie automatisierte Refactorings, Unit-Tests oder Code-Formatierungen. Unter dem Gesichtspunkt der Nachhaltigkeit kann das ein Problem sein, da in großen Projekten die Produktivität darunter leidet.

Die Kollaborationsplattform *eCollab* ist in der Sprache PHP geschrieben und diese gehört zu einem Kreis, in dem die Tool-Landschaft eher eingeschränkt ist. Für *eCollab* ist *Eclipse* die IDE der Wahl, doch um die Anbindung an die Infrastruktur zu ermöglichen, muss sie erst noch durch Plugins erweitert werden. Dazu gehört zunächst die Unterstützung von *Git*-Repositorys mit dem Plugin *EGit*. Für PHP gibt es mit den *PHP Developer Tools* ein Paket mit zahlreichen Erweiterungen, die unter anderem das Ausführen von Unit-Tests und die Prüfung auf Programmiervorgaben erst ermöglichen. Alle Plugins müssen aber nach ihrer Installation einzeln konfiguriert werden, was ein nicht zu unterschätzender Aufwand ist. Um diesen zu vermeiden, wurde die gesamte Entwicklungsumgebung virtualisiert. Allen Entwicklern steht das gleiche System mit der gleichen Software und Konfiguration in einem sofort funktionstüchtigen Zustand zur Verfügung. Diese Einheitlichkeit erleichtert letztlich auch die Kommunikation und Zusammenarbeit im Team.

5 Private Build

Jeder Entwickler muss dafür sorgen, dass immer ein „grüner“ Build-Status vorhanden ist, sprich die Software erfolgreich kompiliert und getestet werden konnte. Dies kann dazu führen, dass als Vorsichtsmaßnahme eine Integration erst dann durchgeführt wird, wenn etwa ein Feature komplett fertiggestellt wurde. Das widerspricht allerdings dem Grundgedanken der Continuous Integration (CI), da dadurch die Gefahr erhöht wird, dass die eigenen Änderungen nicht mehr mit denen der anderen Entwickler harmonieren [DMG07]. Ein fehlerhafter Build ist im Grunde aber auch nichts Schlechtes, zeigt er doch, dass ein Fehler in der Software aufgedeckt wurde, bevor dieser im späteren Verlauf der Entwicklung Probleme verursachen kann.

Um dennoch frühestmöglich die Sicherheit zu erhalten, dass die eigenen Änderungen keine Probleme bereiten, gibt es den Private Build: Dieser erlaubt das Bauen und Testen der Software auf dem eigenen Entwicklungsrechner. Voraussetzung hierfür ist, dass alle Testskripte als Teil der Anwendung mitgeliefert werden und von jedem Entwickler ausgeführt werden können. Die Tests können somit auch ohne zentrale QA-Infrastruktur verwendet werden, was besonders für die schnelle Erkennung von selbst verursachten Fehlern nützlich ist.

Einige CI-Werkzeuge bieten deswegen auch die Möglichkeit, einen inoffiziellen Versuchs-Build durchzuführen [Wi11]. Dieser versteht sich als Private Build innerhalb der zentralen QA-Umgebung. Hier wird der gesamte Integrations-Vorgang simuliert und dem Entwickler später das Ergebnis mitgeteilt, ohne dass die anderen Projektbeteiligten etwas davon erfahren. Ginge das Ergebnis an alle Beteiligten, wäre der „Private“-Aspekt verloren: Experimentieren wäre nicht mehr möglich, da jeder Testlauf offiziell wird und dem Entwickler möglicherweise der „Gesichtsverlust“ droht.

Der Private Build wird nach dem Aktualisieren des eigenen Repositorys mit den Änderungen anderer Entwickler durchgeführt. Die Tests sorgen dafür, dass zum Beispiel Erweiterungen von öffentlichen Schnittstellen zu keinen unvorhergesehenen Fehlern führen. Treten keine Probleme auf, können die eigenen Änderungen in das von allen Entwicklern genutzte Repository eingestellt werden.

6 Repository-Workflow

In den letzten Jahren hat sich die Entwickler-Community weg bewegt von den klassischen Werkzeugen der Versionsverwaltung, vertreten durch *CVS* und *Subversion*, und wechselt nun zur neuen Generation: den verteilten Versionsverwaltungssytemen (DVCS) [Sp12, GDB11]. Prominentestes Werkzeug dieser Gattung ist das vom Schöpfer des Linux-Kernels entwickelte Open-Source-Werkzeug *Git* [Sp12].

Ein Unterschied zu den klassischen Tools ist, dass jeder Entwickler im Vergleich zu früher nicht mehr bloß eine „Arbeitskopie“ besitzt, sondern das gesamte Repository mit der kompletten Versionsgeschichte. Damit entfällt im Prinzip die Notwendigkeit eines zentralen Servers, mit dem alle Entwickler ihre Arbeitskopie synchronisieren müssen.

Das verteilte Modell zeichnet sich vor allem durch seine Flexibilität aus, denn es bietet viele verschiedene Einsatzmöglichkeiten; unter anderem kann damit auch das alte, zentrale Modell realisiert werden. In der Open-Source-Szene setzt sich zunehmend das Fork-und-Pull-Modell durch, welches vor allem durch den Hosting-Anbieter *GitHub* für viele greifbar wurde.

Software wird bei einem DVCS grundsätzlich „geforkt“, das heißt, die gesamten Quellen des Projektes inklusive seiner Versionsgeschichte werden kopiert und sind dadurch im Prinzip als eigenständiges Projekt zu betrachten. Software lebt aber von der Zusammenarbeit der Entwickler, deswegen wird durch einen „Pull-Request“ den ursprünglichen Entwicklern des Projektes mitgeteilt, dass in einem Fork diverse Änderungen vorliegen, die in das Ursprungsprojekt übernommen werden können. Es obliegt den Empfängern des Pull-Requests, diesen zu akzeptieren und die Änderungen zu übernehmen, Korrekturen zu fordern oder die Anfrage gänzlich abzuweisen. Typischerweise dient ein Pull-Request als Diskussionsplattform, in der alle Beteiligten die Änderungen diskutieren können, sollte es Klärungsbedarf geben.

Dieses Modell lässt sich für die nachhaltige Entwicklung abwandeln und mit dem klassischen zentralen Modell verbinden. Der Vorteil des Fork-und-Pull-Modells liegt zunächst in dem integrierten Code-Review: Bei Bedarf können sich die Entwickler einen Pull-Request zusenden, der dann von einer oder mehreren Personen abgenommen wird. Ansonsten kann die Entwicklung gemäß dem zentralen Modell erfolgen, indem alle Entwickler ihre Änderungen in ein gemeinsames Repository einfließen lassen. Hier besteht eine weitere Möglichkeit: Die Änderungen können zunächst in einen speziellen Branch oder in ein Zwischenrepository eingestellt werden, von wo sie erst nach einer automatisierten Qualitätssicherung in den Hauptzweig oder das Hauptrepository gelangen [GDB11] (Abb. 6).

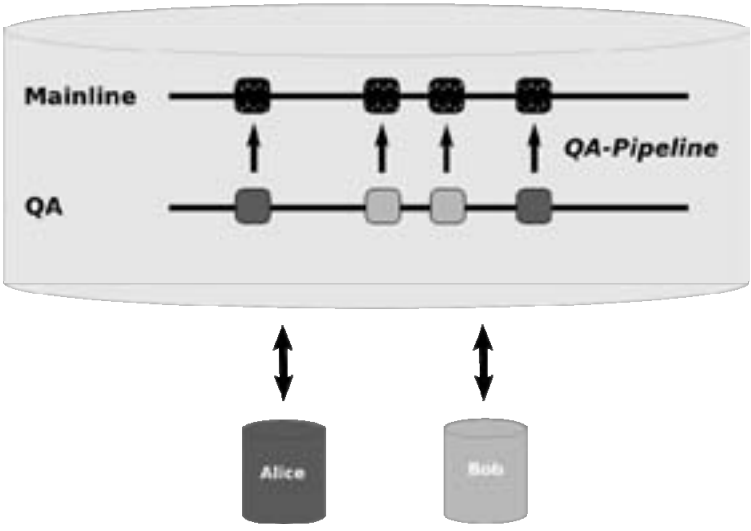

Abbildung 6: Entwickler arbeiten mit einem zentralen Repository, das sich in einen Entwicklungs- und automatisierten Produktionszweig aufteilt.

Dieses Verfahren bietet den Vorteil, dass jederzeit eine qualitätsgesicherte Version der Software vorliegt. Voraussetzung ist, dass der QA-Prozess automatisch ausgeführt wird, sobald eine neue Änderung eingestellt wurde, ohne dass die Entwickler hier manuell eingreifen müssen. Damit ist gewährleistet, dass der Prozess kontinuierlich für jede Änderung neu gestartet wird. Des Weiteren muss sichergestellt werden, dass erfolgreich getestete Änderungen automatisch in den Hauptzweig (*Mainline* in Abb. 6) eingebracht werden und diese somit allen Entwicklern als funktionstüchtige Basis zur Verfügung stehen.

Als Open-Source-Variante von *GitHub* steht *Gitorious* zur Verfügung, welches ebenfalls das Fork-und-Pull-Modell auf einfache Weise verfügbar macht. Entwickler-Teams können über diese Plattform ihre Repositories verwalten und Änderungen über Pull-Requests („Merge-Request" in der *Gitorious*-Terminologie) austauschen. Es steht des Weiteren immer auch der direkte Weg über die *Git*-Werkzeuge zum Austausch der Quellen zur Verfügung.

Es gilt zu beachten, dass die Grundprinzipien der Continuous Integration durch die verteilte Versionsverwaltung nicht untergraben werden: Änderungen sollen mindestens einmal täglich – spätestens am Ende des „Programmiertages" – in das zentrale Repository eingestellt werden, damit diese den übrigen Entwicklern zur Verfügung stehen. Änderungen, die nicht ihren Weg in das zentrale Repository finden, können nicht automatisiert getestet werden. Hier ist auch die sogenannte „Promiscuous Integration" zu vermeiden, also das Austauschen von Quellen untereinander – vorbei am zentralen Repository und damit der Qualitätssicherung [Fo09].

Hooks

Jedes Projekt definiert typischerweise einen „Wertekanon". Darunter sind zum einen die aus *Extreme Programming* bekannten, eher abstrakten Werte wie Mut, Respekt oder Einfachheit, zum anderen aber auch konkrete Elemente in Form von Programmierrichtlinien zu verstehen, zu denen sich alle Entwickler bekennen. Ziel solcher Richtlinien ist es, den Quelltext nicht nur einheitlich, sondern vor allem optimal lesbar zu gestalten. Des Weiteren sollen mögliche Probleme durch das Einfordern eines bestimmten Stils gänzlich ausgeschlossen werden. Am besten gelingt dies, wenn die Richtlinien lebendig, also direkt in den Entwicklungsablauf integriert sind. Dadurch wird die Einhaltung der Richtlinien zum Automatismus.

Alle gängigen Versionierungstools bieten Hooks an, womit die Abläufe innerhalb des Repositorys durch eigene Skripte angepasst werden können. Hooks sind dabei an Ereignisse geknüpft: So gibt es bei *Git* etwa den Commit-Hook. Hooks erscheinen immer in einer Pre- und einer Post-Variante. Erstere kann dazu genutzt werden, den Vorgang unter Umständen abzuweisen; bei der Post-Variante ist der Commit zu diesem Zeitpunkt bereits geschehen.

Ein Pre-Commit-Hook eignet sich dafür, die Änderungen innerhalb eines Commits auf Verstöße gegen die Programmiervorgaben zu überprüfen. Werden Verstöße festgestellt, kann der gesamte Commit deswegen abgebrochen werden. Dem Entwickler wird an-

schließend mitgeteilt, welche Unstimmigkeiten entdeckt worden sind, sodass die Probleme behoben werden können. Somit gelangt am Ende kein Quelltext in das Repository, der gegen die im Projekt definierten Vorgaben verstößt [Th11].

Weitere Ansatzpunkte für Skripte ergeben sich beim Empfang von Änderungen aus einem anderen Repository. *Git* nennt sie Receive-Hooks und sie eignen sich dafür, weitere Aktionen vor oder nach dem Empfang auszuführen. Sie können dazu genutzt werden, um bei soeben eingetroffenen Änderungen den Qualitätssicherungsprozess anzustoßen.

7 QA-Pipeline

Die automatisierte Qualitätssicherung nimmt einen zentralen Platz ein. Hier wird entschieden, ob Änderungen in den Hauptzweig übernommen werden oder nicht (Abb. 6). Die Entscheidung stützt sich dabei auf eine ganze Reihe von Tests, über deren Ausführung alle Entwickler Schritt für Schritt informiert werden.

Die Durchführung wird von der QA-Pipeline übernommen: Durch sie werden alle Aspekte der Anwendung überprüft und dadurch wichtiges Feedback gewonnen, das den Entwicklern übermittelt wird. Die QA-Pipeline, detailliert beschrieben von Humble und Farley, teilt sich dabei in mehrere Phasen auf, die jeweils für einen bestimmten Aspekt der Qualitätssicherung vorgesehen sind [HF10]. Sie beginnt mit Unit-, Integrations- und Akzeptanztests. Diese überprüfen die Funktionalität der Anwendung über mehrere Abstraktionsschichten hinweg – vom Code bis zum Feature aus Kundensicht. Darauf folgen Phasen für die nicht-funktionalen Anforderungen, etwa Performance oder Sicherheitsaspekte. Weitere Phasen sind für die Prüfung der Code-Qualität und das Testen eines möglichen Deployments zuständig (Abb. 7).

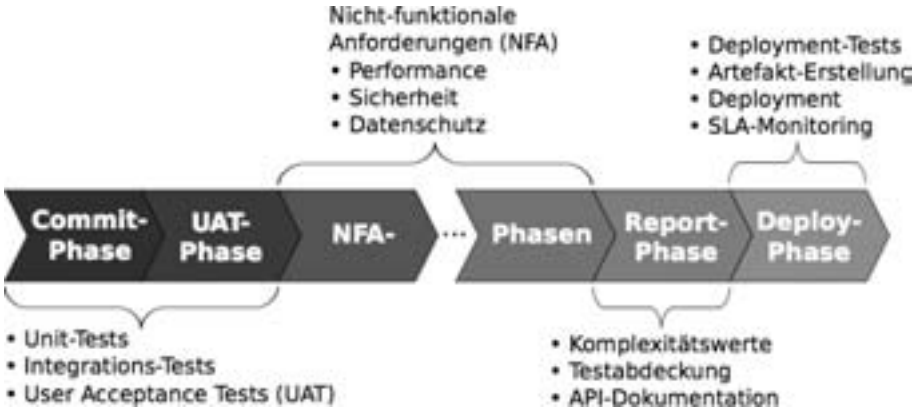

Abbildung 7: Schema einer QA-Pipeline [QT12]

Auf jeder Stufe erhalten alle Entwickler Rückmeldung, ob eine Änderung Fehler verursacht hat. Wird ein Fehler entdeckt, wird die gesamte Pipeline abgebrochen und die entsprechenden Phasen als fehlerhaft markiert. Die Probleme sollten anschließend so schnell wie möglich behoben werden, da wegen eines dauerhaften Fehlerzustands der gesamte Feedback-Aspekt verloren geht. Die Granularität der Fehlererkennung nimmt mit jeder Stufe ab, weswegen es wichtig ist, einen Fehler möglichst früh aufzudecken. Andernfalls kann die Fehlerquelle unter Umständen nicht genau ermittelt werden, wodurch sich die Fehlerbehebung erschwert.

Das Konzept „QA-Pipeline“ ist entstanden, damit die zuvor in der Regel getrennten Build- und Deployment-Abläufe zusammengeführt werden konnten. Es ist also das Bindeglied, durch das aus einer Code-Änderung die fertige, vollständig qualitätsgesicherte und produktionsreife Anwendung entsteht. Für die Kollaborationsplattform *eCollab* ist die Pipeline mit dem Continuous-Integration-Server *Jenkins* umgesetzt worden, da dieser dank zahlreicher Plugins ein äußerst flexibles Werkzeug darstellt.[5]

Welche Phasen letztendlich in der Pipeline definiert werden, hängt von den konkreten Anforderungen der Software ab. Hier können also bewusst Schwerpunkte auf ausgewählte QA-Aspekte gelegt werden. Die Pipeline verbietet nicht, dass etwa im nächtlichen Betrieb weitere Aspekte geprüft werden. Das können zum Beispiel länger laufende Performance-Tests sein. Dabei gilt es jedoch zu beachten, dass die zusätzlichen Tests nicht entscheidend für den Betrieb der Anwendung sein dürfen, da die Pipeline genau dies sicherstellen soll.

7.1 Commit und User Acceptance Tests

Diese Phasen bilden den Start der Pipeline. Zunächst wird auf Unit-Tests gesetzt, um die Grundfunktionalität der Anwendung auf Code-Ebene zu überprüfen. Die erste Phase muss so schnell wie möglich durchlaufen werden, damit den Entwickler ein zeitnahes Feedback zu seiner Änderung erreicht. Im weiteren Verlauf wird die Abstraktionsebene immer weiter erhöht: Integrations-Tests überprüfen das Zusammenspiel mehrerer Komponenten, bis schließlich mit den User Acceptance Tests die Features aus Sicht des Kunden überprüft werden [QT12]. Bei Letzterem werden die Use Cases automatisch „durchgeklickt“ und die Ergebnisse mit vordefinierten Erwartungswerten verglichen.

7.2 Nicht-funktionale Anforderungen

Weitere Phasen überprüfen beispielsweise die Performance, die Sicherheit oder auch den Datenschutz. Für Performance-Tests wird die Anwendung gezielt unter Last gesetzt, um Aufschlüsse über das Antwortzeitverhalten bei einem hohen Daten- oder Benutzeraufkommen zu gewinnen. Sicherheitstests werden mit Fuzzing-Werkzeugen ausgeführt, die diverse Angriffsvektoren überprüfen und im Prinzip versuchen, die Anwendung zum Absturz zu bringen. Durch Vulnerability Scans können Sicherheitslücken aufgedeckt werden. Wird bei Performance- oder Sicherheitstests ein bestimmtes Fehler-Level über-

[5] Siehe die Visualisierung der QA-Pipeline unter: https://scm.thm.de/pipeline

schritten, gelten die Phasen als gescheitert. Datenschutz-Aspekte können in Form eines *aktiven Verfahrensverzeichnisses* für jedes neue Software-Release veröffentlicht werden, indem innerhalb der Anwendung durch Oberflächentests überprüft wird, wer in welcher Benutzerrolle personenbezogene Daten anderer Benutzer sehen kann.[6] [QT12]

7.3 Reporting

Für die nachhaltige Software-Entwicklung ist es ratsam, diverse Qualitätsmetriken ständig im Auge zu behalten. Verschlechtern sich etwa die Komplexitätswerte in einem bestimmten Modul, müssen die Entwickler darüber informiert werden. Dies kann so weit gehen, dass die Pipeline abgebrochen wird, wenn ein gewisses Maß überschritten wird [Th11]. Nur so ergibt sich aus den Software-Metriken auch eine Konsequenz.

Am Beispiel der Metrik „Instability vs. Abstraction“ (Abb. 4) kann der QA-Verantwortliche des Projekts eine mittlere Distanz (D-Wert) aller Pakete von der Ideallinie festlegen, die nicht innerhalb einer vorgegebenen Toleranz überschritten werden darf. Je näher die Pakete der Ideallinie kommen, desto eher sind sie erweiterbar, wiederverwendbar und wartbar, desto höher also die Softwaregüte [Ma94]. Alternativ kann der QA-Verantwortliche festlegen, dass sich der mittlere D-Wert von Build zu Build nicht verschlechtern darf, ansonsten wird der aktuelle Build abgebrochen. In beiden Fällen erfährt der Entwickler, welche D-Werte sich durch den abgebrochenen Build verschlechtert hätten. Die Nachhaltigkeit der Softwaregüte kann so konsequent und automatisiert eingefordert werden.

7.4 Deployment

In der letzten Phase der Pipeline wird die endgültige Produktionsreife der Software festgestellt: In einer produktionsnahen Umgebung wird ein Test-Deployment durchgeführt, welches anschließend besonders unter Migrations- und Rollback-Gesichtspunkten überprüft wird. Des Weiteren kann das Test-Deployment als Schaufenster für den Kunden genutzt werden, der sich dadurch ständig ein Bild vom aktuellen Entwicklungsstand machen kann [HF10].

8 Zukünftige Arbeiten

Im Rahmen eines Forschungsprojekts zur Qualitätssicherung in globalen Open-Source-Entwicklungsprojekten soll die Nachhaltigkeit in der Software-Entwicklung mit weiteren Automatismen und Tools unterstützt werden. Unter anderem sollen die folgenden Herausforderungen gelöst werden:

- Nachhaltigkeits-Checks in der QA-Pipeline: Während Build-spezifische Qualitätsmetriken umfassend und detailliert ermittelt werden (Testabdeckung, Richtlinientreue, D-Wert, C.R.A.P.-Faktor [Sa07]), fehlt eine automatisierte Trend-Analyse zur Soft-

[6] Beispiel eines aktiven Verfahrensverzeichnisses: https://ecollab.thm.de/infos/datenschutz.php

waregüte auf der Basis aller Metriken aller Builds. Eine anhaltende Degenerierung einzelner Softwarekomponenten auf Paket-, Klassen- und Methodenebene soll erkannt werden und in konkrete Gegenmaßnahmen münden.

- Integration der Nachhaltigkeits-Checks in global verteilte Entwicklungsprojekte: Der gezeigte Prozess soll beispielhaft für die Weiterentwicklung der weltweit verbreiteten Open-Source-Lernplattform *Moodle* adaptiert werden. Dafür sind zunächst die technischen und strukturellen Voraussetzungen in Form eines Online-QA-Labors zu schaffen. Anschließend muss der QA-Prozess in die bestehenden Prozesse der *Moodle*-Community[7] integriert werden.

9 Zusammenfassung

Für eine nachhaltige Software-Entwicklung ist es wichtig, die Entwickler an zentralen Stellen im Entwicklungsprozess zu entlasten. Für Neulinge soll der Einstieg in das Projekt so einfach wie möglich gestaltet werden, sodass sie innerhalb kürzester Zeit an der Entwicklung teilnehmen können. Gleichzeitig soll sichergestellt werden, dass die Qualität der Software durch die Fluktuation der Entwickler nicht leidet.

Sich immer wiederholende Abläufe sind prädestiniert, automatisiert zu werden. Dazu zählen besonders die Integration der Code-Änderungen und die Qualitätssicherung. Entlang mehrerer Stufen wird den Entwicklern dabei Feedback geliefert, sodass sie im Problemfall sofort reagieren können.

Nachhaltige Software-Entwicklung fokussiert auf die Wahrung der bisher erreichten Softwaregüte trotz hoher Fluktuation der Entwickler. Der Beitrag hat gezeigt, dass mithilfe von Open-Source-Werkzeugen QA-Automatismen implementiert werden können, um eine schleichende Degenerierung der Softwaregüte von Build zu Build zu erkennen und die QA-Verantwortlichen auf den Plan zu rufen.

Danksagung

Die Autoren bedanken sich bei den Gutachtern für die konstruktive Kritik und bei Julian Hochstetter (M.Sc.) und Paul-Christian Volkmer (M.Sc.) für ihr Engagement und ihre Hilfe bei der Recherche und Evaluation der Open-Source-Tools sowie beim Aufbau und Betreiben der Source-Code-Management-Infrastruktur und QA-Pipeline.

[7] http://moodle.org/development/ (Abruf 28.07.2012)

Literaturverzeichnis

[DMG07] Duvall, P. M.; Matyas, S.; Glover, A.: Continuous Integration – Improving Software Quality and Reducing Risk. Addison-Wesley Professional, 2007.

[Fo09] Fowler, M.: FeatureBranch. 2009. http://martinfowler.com/bliki/FeatureBranch.html (Abruf 28.07.2012).

[GDB11] Geisler, M.; Digulla A.; Bucka-Lassen, K.: Mercurial – Schnelle und skalierbare Versionsverwaltung. In: Objekt Spektrum (Jan. 2011), S. 80–85.

[HF10] Humble, J.; Farley, D.: Continuous Delivery – Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010.

[LM06] Lanza, M.; Marinescu, R.: Characterizing the Design. In: Object-Oriented Metrics in Practice. Springer, Berlin Heidelberg, 2006, S. 23–44.

[Ma94] Martin, R. C.: OO Design Quality Metrics – An Analysis of Dependencies. 1994. http://www.objectmentor.com/resources/articles/oodmetrc.pdf (Abruf 28.07.2012).

[QT12] Quibeldey-Cirkel, K.; Thelen, C.: Continuous Deployment. In: Informatik-Spektrum 35.4 (2012), S. 301–305.

[Sa07] Savoia, A.: Pardon My French, But This Code Is C.R.A.P. (2). 2007. http://www.artima.com/weblogs/viewpost.jsp?thread=210575 (Abruf 28.07.2012).

[Sp12] Spinellis, D.: Git. In: IEEE Software 29.3 (2012), S. 100–101.

[Th11] Thelen, C.: Qualitätssicherung von Software-Altsystemen durch automatisierte Verfahren. Master-Thesis. Technische Hochschule Mittelhessen, 2011.

[Wi11] Wiest, S.: Continuous Integration mit Hudson. dpunkt, Heidelberg, 2011.

Open-Source-Tools und Online-Dienste

Eclipse	http://www.eclipse.org
EGit	http://www.eclipse.org/egit
Git	http://git-scm.com
GitHub	https://github.com
Gitorious	http://gitorious.org
Jenkins	http://jenkins-ci.org
Ohloh	http://www.ohloh.net
PDT	http://www.eclipse.org/projects/project.php?id=tools.pdt
Redmine	http://www.redmine.org (Abruf 28.07.2012)