FERNUNIVERSITÄT HAGEN

INSTITUT FÜR

Continuous Integration und Jenkins

Seminararbeit
im Studiengang 'Bachelor Informatik'

vorgelegt von
Daniel Wolfschmidt

Betreuer: Daniela Keller

Ablieferungstermin: 1. Juli 2018

Daniel Wolfschmidt
Fließbachstraße 18
91052 Erlangen
mailto:DanielWolfschmidt@gmx.de
Matrikelnr.: 9601244

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In den Jahrzehnten der Historie von Softwarentwicklung gab es immer wieder neue Erkenntnisse und neue State-of-the-Art Methoden der Entwicklung von Software. So gab es lange Zeit große, monolithische Desktopapplikationen, welche nur als großes Ganzes funktionierten. Mittlerweile geht der Trend hin zu Microservices[1]. Diese, bis auf die Schnittstellenbeschreibung, unabhängige Entwicklung der einzelenen Komponenten von Software, erlaubt eine wesentlich schnellere Entwicklung von Software.
Auch die Einstellung der Nutzer hat sich verändert. Durch das schnelle Entwickeln von Patches und Updates, ist der Anwender zum Betatester von Software avanciert. Er ist es nicht nur gewohnt, dass in schneller Abfolge neue Änderungen an der Software veröffentlicht werden, sondern er erwartet es regelrecht.
Ein weiterer Aspekt sind die neuen Vorgehensmodelle im (Software-)Projektmanagement. In der Vergangenheit war es gang und gäbe, das Wasserfall Modell zu verwenden. Dabei wird der Test in einer späten Projektphase durchgeführt, und die Entwicklungsphase dauert sehr lange, bis das Gesamtprodukt fertig entwicklet ist. Heutzutage bedient man sich agiler Modelle wie z.B. Scrum. Hierbei wird in regelmäßigen Abständen eine überschaubare Verbesserung des Produkts veröffentlicht. Dies unterstützt die oben beschriebene Veränderung in modernen Software Architekturen.
In diesen Zeiten immer kürzerer Entwicklungszyklen gewinnt die Entwicklung von Konzepten zur Sicherung der Code Qualität zunehmend an Bedeutung. Eines dieser Konzepte, das ich in der hier vorliegenden Seminararbeit näher beleuchten möchte, ist **Continuous Integration**.
Software soll schnell entwickelt und getestet werden. Dies ist nur durch eine weitreichende Automatisierung von Build-, Integrations- und Testschritten möglich. Mehrere kommerzielle und kostenlose Tools zur Unterstützung von Continuous Integration existieren am Markt, wobei diese Arbeit **Jenkins** genauer vorstellt.

[1] Dabei handelt es sich um eine Zusammenstellung unabhängiger Prozesse, die durch eine sehr leichtgewichtige Kommunikationsschicht verbunden sind. Ein Beispiel zur Kommunikation ist HTTP(S). Weitergehende Informationen z.B. unter [Fow14]

2 Continuous Integration

Dieses Kapitel beschäftigt sich mit einem der beiden Hauptthemen, nämlich Continuous Integration. Dabei möchte ich mich diesem Thema zunächst durch eine genaue Begriffsbestimmung nähern, wobei auch eine Abgrenzung zu anderen, ähnlichen Begriffen eine wichtige Rolle spielt. Im weiteren Verlauf sollen dann noch der Einsatz von Tools zur Unterstützung sowie die Gründe zum Einsatz dieser Methodik näher beleuchtet werden.

2.1 Begriffsklärung

Den Einstieg soll eine kurze Beschreibung von Martin Fowler bilden, er gilt als der gesitige Vater von Continuous Integration und wird mit diesem Artikel in vielen anderen Abhandlungen zu dem Thema zitiert:

> *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* [Fow06]

Es geht hier also um das kollaborative Arbeiten in einem Team, insbesondere das Integrieren von Code in eine gemeinsame Code-Basis. Das heißt ferner, dass es eine Methodik ist die für einen einzelnen Entwickler kaum Bedeutung hat. Das ist auch einleuchtend, denn seinen eigenen Code in eben diesen zu integrieren, macht kaum Sinn.
Ferner sollte dies sehr oft passieren, am besten mehrmals täglich. Auch das ist eine sehr sinnvolle Daumenregel. Wenn man zu lange seinen Code nicht in die gemeinsame Code-Basis integriert, so besteht die Gefahr, dass man zu weit weg ist davon und dadurch immense Aufwände entstehen. Dann wird aus dem Integrieren einer eigentlich keinen Änderung eine umfassende Merge[2]-Session. Der dritte Teil der vorliegenden Beschreibung geht darauf ein, wie man den Nachweis erbringen kann, dass die Integration erfolgreich war. Dafür soll es automatisierte (und dadurch auch standardisierte) Builds geben. Diese Builds erzeugen zunächst aus dem menschenlesbaren Code den von Maschinen ausführbaren Code sowie dazu gehörige Tests in verschiedenem Detailgrad. Martin Fowler gibt in seienr Beschreibung auch an, dass die Tests mit zu diesem automatisierten Builds gehören. Hierbei muss man auf eine sinnvolle Testtiefe achten. Wenn man die Test-Pyramide[3] zu Rate zieht, muss man darauf achten, dass alle Tests fertig sind bevor der nächste Entwickler sein Änderungen in die Code-Basis integriert. Deshalb ist es eventuell am Besten, wenn man während des initialen Builds

[2] Mergen bezeichnet das Vergleichen mehrerer Änderungen an einer Quelldatei und das Zusammenführen („mergen“) dieser Änderungen. vgl.:`https://de.wikipedia.org/wiki/Merge`

[3] Dabei handelt es sich um eine Darstellung der unterschiedlichen Testtypen in hierarchischer Form, wobei von unten nach oben die Geschwindigkeit abnimmt und die Kosten zunehmen. vgl.:[Fow12]

nur Unittests ausführt, und ein möglicherweise nur einmal am Tag laufender Build dann detailierter testet.
Um diese Beschreibung der Kernpunkte von Continuous Integration auf eine breitere Basis zu stellen, möchte ich noch eine zweite Quelle nutzen, um von einem anderen Blickwinkel auf das Thema zu blicken.

> *The practice of continuous integration represents a fundamental shift in the process of building software. It takes integration, commonly an infrequent and painful exercise, and makes it a simple, core part of a developer's daily activities. Integrating continuously makes integration a part of the natural rhythm of coding, an integral part of the test-code-refactor cycle. Continuous integration is about progressing steadily forward by taking small steps.*
> [Rog04]

Der Autor dieses Konferenzbeitrags ist R. Owen Rogers. Er arbeitet bei Thoughtworks, derselben Firma bei der auch Martin Fowler arbeitet. Er stammt von einer Konferenz aus dem Jahr 2004, also zeitlich zwischen der initialen Version über CI[4] seiner aktuellen Version aus dem Jahr 2006.
Es wird dabei eher der Fokus auf die Auswirkungen von Continuous Integration auf die Softwareentwicklung und der Einfluss auf die Qualität von Software gelegt. Er geht dabei vor allem darauf ein, dass das häufige Integrieren der zentrale Teile dieses Konzepts ist. Das deckt sich mit der oben vorgestellten Sichtweise von Martin Fowler. Desweiteren setz er den Ansatz in den Kontext von „test-code-refactor", und geht damit auch auf einen anderen bereits vorgestellten Aspekt ein, nämlich das Überprüfen des Erfolgs der Integration. Der Anteil, dass es eine Praktik ist, die hauptsächlich in Teams sinn macht, ist eher implizit enthalten in diesem Text und wird nicht extra erwähnt.
Er deckt sich damit mit der Sichtweise von Martin Fowler, und beleuchtet das Thema einfach nur aus einem anderen, eher anwendungsbezogenen Blickwinkel. Das ist auch nachvollziehbar, da es sich hier nicht um eine theoretische Abhandlung handelt sondern einen Konferenzbeitrag, der an Anwender dieser Technik gerichtet war.

Zusammenfassend bleibt zu sagen dass mit Continuous Integration die Zusammenarbeit eines Entwicklerteams an einer gemeinsamen Code-Basis verbessert werden soll. Dies soll geschehen durch kontinuierliches Zusammenführen der Änderungen aller Beteiligten und das automatisierte Prüfen des Ergebnisses.

[4] Ab hier werde ich CI als Abkürzung für „Continuous Integration" verwenden. Diese Abkürzung ist auch im Abkürzungsverzeichnis zu finden.

Abgrenzung zu anderen Begriffen

Es gibt einige Begriffe die Continuous Integration sehr ähnlich sind. Dieser Abschnitt soll genauer umreißen, wo der Unterschied ist und was sie vielleicht gemeinsam haben.

- **Continuous Delivery**
  Auch hier hat Martin Fowler eine zusammenfassende Definition geliefert:

  > *You achieve continuous delivery by continuously integrating the software done by the development team, building executables, and running automated tests on those executables to detect problems. Furthermore you push the executables into increasingly production-like environments to ensure the software will work in production. To do this you use a DeploymentPipeline.*
  > [Fow13]

  Bei Continuous Delivery (CD[5]) wird der Gedanke von Continuous Integration aufgegriffen, und weiterentwickelt. Während Continuous Integration sich komplett in der Entwicklung bewegt, umfasst Continuous Delivery auch Schritte bis hin zum Kunden. Es werden weitere Schritte wie das Paketieren als Deliverable (z.B. Erstellen eines Setups) und das Deployment (z.B. Bereitstellen als Download oder Einstellen in einen AppStore) betrachtet. Das Ziel dessen ist, dass das Ergebnis der Entwicklung zu jedem Zeitpunkt zum Kunden geschickt werden könnte.

- **Continuous Deployment**
  Hier beziehe ich mich auf den Abstract eines Konferenzbeitrages von Helena Holmström Olsson

  > *The concept of continuous deployment, i.e. the ability to deliver software functionality frequently to customers [...]*
  > [OAB12]

  Continuous Deployment bringt das CI-Konzept zwei Schritte weiter. Nicht nur wird hier wie bei Continuous Delivery in „Production-like“ Umgebungen installiert sondern sehr häufig (potentiell mit jedem Build) in die Produktion gegeben. Der Unterschied ist hier erstmal nur marginal, aber prinzipiell kann man sagen, dass bei Continuous Delivery festgelegt wird, wann

[5] Ab hier werde ich CD als Abkürzung für „Continuous Delivery“ verwenden. Diese Abkürzung ist auch im Abkürzungsverzeichnis zu finden.

man in die Produktion geht, und bei Continuous Deployment dies laufend passiert. [Pau15]. Das heißt aber auch, dass dieses Konzept das am weitesten automatisierte ist mit allen Vor- und Nachteilen.

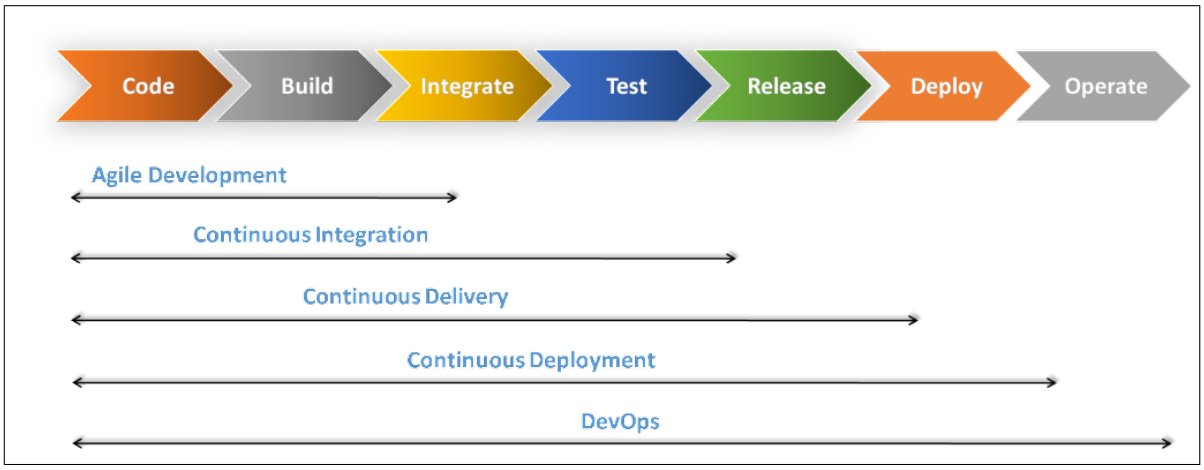

Abbildung 1: Übersicht über verschiedene Begriffe [Nim16]

2.2 Ablauf von Continuous Integration

Da in der Beschreibung von CI die Rede ist vom Integrieren von Code Änderungen, muss es auch irgendwie eine gemeinsame Basis geben, in die diese Änderungen einfließen. Eine solche gemeinsame Basis ist ein sogenanntes Source-Control-Management-System (SCM[6]). Dabei handelt es sich um ein System, in dem Änderungen an einer Datei, oder der Struktur der Dateien, nachvollziehbar gemacht werden, und man verschiedene Stände abrufen kann.(vgl. [Fow06]) Es gibt dazu verschiedene Konzepte, entweder eine zentrale Stelle an der alle Dateien inklusive Historie verwaltet werden, oder verteilte Systeme, bei der es keine ausgezeichnete zentrale Instanz gibt.

1. **Get Sources**
   Die Arbeit des Entwicklers basiert auf dem aktuellen Stand der Quellen aus dem SCM. Deshalb holt er sich zunächst diesen auf seinen lokalen Computer, um seine Arbeit zu beginnen.

2. **Implement the Changes to the Code**
   Der Entwickler folgt diesem Prozess aus einem bestimmten Grund, nächlich entweder ein neues Feature zu implementieren oder bekannte Fehler in der Software zu beheben. Dies geschieht in diesem Schritt. Der Entwickler führt die ihm übertragenen Aufgaben aus. Dabei wird bei

[6]Ab hier werde ich SCM als Abkürzung für „Source Control Management“ verwenden. Diese Abkürzung ist auch im Abkürzungsverzeichnis zu finden.

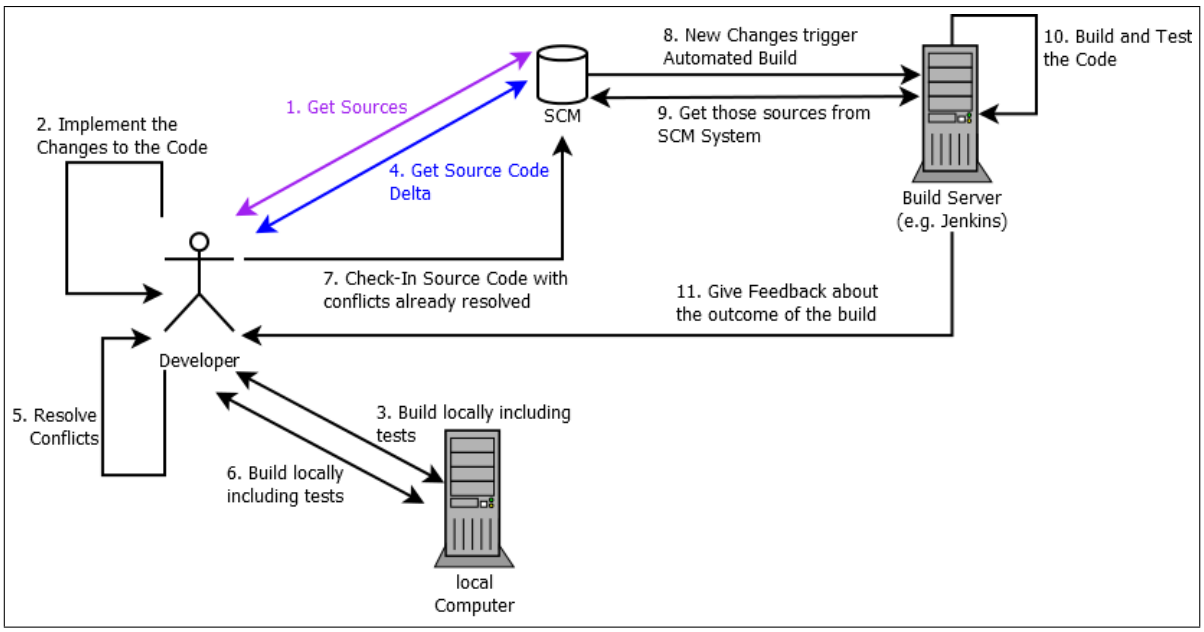

Abbildung 2: Schematischer Ablauf von CI

CI besonders Wert auf Tests gelegt. Dies ist mittlerweile zum Quasistandard in der Sotwareentwicklung geworden, und folgt vom schematischen Aufbau her der Testpyramide, wie sie in ([Fow12]) beschrieben wird.

3. **Build locally including tests**
   Der vorhergehende Schritt hat sich komplett auf das Implementieren von Code bzw. Codeänderungen sowie das Implementieren und Ändern von Tests beschränkt. Diese müssen auch noch auch Fehlerfreiheit überprüft werden, bevor man sie in das SCM einfügt. Die erste Kontrollinstanz ist nun der automatisierte Build auf der Entwicklermaschine. Darunter versteht man das Kompilieren und Linken der Quellen sowie das Ausführen der zuvor geschriebenen automatisierten Tests. Hierbei beschränkt man sich zumeist auf UnitTests.

4. **Get Source Code Delta**
   Während der Entwickler seinen Auftrag ausgeführt hat, haben einige seiner Kollegen bereits ihre Arbeit vollendet. Deshalb muss er nun den aktuellen („top-level“) Stand holen, weil es sonst zum Beispiel passieren könnte, dass er Änderungen überschreibt bzw. es Konflikte gibt beim Hinzufügen, die nicht automatisch aufgelöst werden können.

5. **Resolve Conflicts**
   Falls der Entwickler nun in die Situation gekommen ist, dass seine lokalen Änderungen mit Änderungen, die bereits im SCM sind in Konflikt stehen, so muss er diese manuell auflösen. Es gibt einige Konflikte die grundsätzlich auch toolunterstützt automatisch auflösbar sind (z.B.

Änderung an derselben Datei aber in unterschiedlichen Zeilen), jedoch muss bei manchen Konflikten einfach ein Mensch entscheiden, was der richtige Schritt ist.

6. **Build locally including tests**
   Der Entwickler muss nun überprüfen ob seine Änderungen noch funktionieren und alle Tests weiterhin fehlerfrei sind. Dies betrifft nun nicht nur seinen eigenen Code sondern auch den der anderen Entwickler. Es ist denkbar, dass seine Änderungen Auswirkungen an anderen Stellen haben. Dabei ist auch er selbst in der Verantwortung, dass der Code-Stand den er später dem SCM hinzufügen möchte funktioniert. Dies geschieht auf seinem lokalen Rechner.

7. **Check-In Source Code with conflicts already resolved**
   Nachdem lokal der Code erfolgreich kompilierbar ist, und alle (Unit-)Tests fehlerfrei sind, kann der Entwickler seine Änderungen dem SCM hinzufügen.

8. **New Changes trigger Automated Build**
   Je nach SCM und Build Server Kombination gibt es mehrere denkbare Ansätze automatisiert nach jedem Check-In von Code einen Build zu triggern. Diese basieren prinzipiell auf einem Observer Pattern[7] entweder mit Push-Notification (Jede Änderung benachrichtigt automatisch alle Subscriber) oder Pull-Notification (Regelmäßiges Nachfragen der Subscriber, ob sich etwas geändert hat). Egal welche der Methoden zum Einsatz kommt, nach dem Hinzufügen der Änderungen wird ein Build gestartet.

9. **Get those sources from SCM System**
   Der Build Server wurde bisher nur benachrichtigt, es ist aber bisher noch kein Inhalt übertragen worden. Das geschieht in diesem Schritt. Der Einfachheit halber habe ich dabei in Abbildung 2 nur einen einzigen Build Server eingefügt. Prinzipiell gibt es eine Orchestrierung-Instanz und mehrere Build Server. Diese zentrale Instanz koordiniert dabei die Aufgaben und die Server führen diese aus. Im vorliegenden Schaubild ist sowohl die koordinierende Instanz als auch die ausführende auf demselben Server. Die Übertragung findet zur ausführenden Instanz statt, da diese auch den Code kompiliert und testet.

10. **Build and Test the Code**
    Hier wird, genau wie auch auf dem lokalen Rechner des Entwicklers der Code gebaut und getestet. Der große Vorteil gegenüber den Entwicklerrechnern ist, dass es eine klar definierte Instanz ist. Entwicklerrechner sind sehr heterogen vom Aufbau, da im Laufe der Zeit immer mehr „HilfsTools“ die das Arbeiten erleichtern hinzukommen. Der Build Server ist anders, denn er hat ein klar definiertes Set von installierten Programmen und installierten Bibliothe-

[7] Ziel des Observer Patterns ist es eine sog. one-to-many Beziehung zwischen Objekten zu definieren, so dass eine Statusänderung eines Objektes all davon abhängigen Objekte benachrichtigt, bzw. automatisch ändert. vgl. auch [HK02]

ken, die zum Erstellen der Kompilate benutzt werden können. Hier fällt zum ersten Mal auf, wenn der Entwickler sich auf etwas verlässt, das nur auf seiner Maschine vorhanden ist. Dazu zählen auch neue Kompilate. Wenn die Source Dateien nicht explizit in das SCM eingefügt wurden bzw. die Anweisung zum Erstellen des Kompilats aus diesen hinzugefügt wurde, funktioniert zwar der lokale Build aber nicht der Server Build. Dies ist das zentrale Quality Gate im CI Prozess und aufgrund des automatisierten Prozesses und der klar definierten Umgebung die Komponente die das größte Vertrauen in das Ergebnis besitzt.

11. **Give Feedback about the outcome of the build**
    Nachdem der Build fertig ist, muss der Entwickler auch noch auf irgendeine Art und Weise Kenntnis vom Ergebnis erlangen. Entweder wird es auf einer Webseite veröffentlicht, oder er bekommt direkt eine Email mit dem Ergebnis, oder eine Kombination aus beidem. Dies ist wichtig, denn egal wie der Build verlaufen ist, ist es wichtig für die weitere Arbeit. Der schlimmere Fall ist, dass der Build nicht funktioniert hat. Das bedeutet, dass alle anderen Entwickler die sich auf diesen Build stützen blockiert sind in ihrer Arbeit. Es heißt dann also so schnell wie möglich den Grund zu finden und dieses Problem zu beheben. Potentielle Lösung wäre auch, die Änderungen im SCM rückgängig zu machen, damit die anderen Entwickler erst mal ungestört weiter arbeiten können. Im guten Fall, dass der Build erfolgreich war, signalisiert die Benachrichtigung, dass der Entwickler sich nun der nächten Aufgabe widmen kann.

2.3 Gründe Continuous Integration einzusetzen

Die Gründe für den Einsatz von Continuous Integration sind sehr vielfältig. Ich möchte im Folgenden eine kleine Auswahl für Gründe geben:

1. **Qualität steigern**
   Dieser Grund ist ziemlich offensichtlich. Dadurch dass regelmäßig Tests im Build mitlaufen, die ein schnelles Feedback über die Qualität des Codes geben, wird diese auf lange Sicht gesteigert. Man könnte auch gewisse Metriken einführen, die einen Build scheitern lassen, so dass die Entwickler gezwungen sind die Qualität zu erhöhen. Darunter zählt zum Beispiel Code Coverage. Dabei geht es darum wie viel des produktiven Codes von Tests durchlaufen wird und dadurch eine Qualitätsaussage darüber getroffen werden kann.

2. **Management Vorgaben**
   Auch dieser, eher organisatorische, Grund ist vorzubringen. Dadurch, dass Continuous Integration bzw. dessen Weiterentwicklung CD mittlerweile Einzug gehalten hat in weite Teile der Softwareentwicklung, kann auch das Management verlangen, dass dies Eingeführt wird, bzw. als Abteilungs- bzw. Unternehmensziel festlegen. Es sollte jedoch sowieso im eigenen Interesse der Softwareentwicklung sein, solche Praktiken zu verwenden.

3. **Audit Trail**
   Die Einführung von Praktiken wie Continuous Integration und deren Implementierung als ganzes System helfen immens bei der Softwareentwicklung in regulatorischen Umgebungen. Besonders die FDA[8] macht strikte Vorgaben in Bezug auf die Nachvollziehbarkeit von Änderungen, bzw. dem Einfluss den diese auf ein Produkt haben („Audit Trail“).

4. **Schnellerer und spontanerer Release möglich**
   Unter Zuhilfenahme von CI wird der Prozess der Softwareentwicklung weiter voran getrieben als in einem klassischen Setup. Es wird mit jeder Codeänderung getestet und auch die Komponenten untereinander integriert.

5.

- Schnellerer Release möglich für Patch - Vorsichtigere Entwickler, wenn sie wissen dass da eine Kontrollisntanz ist

2.4 Marktübersicht an Tools

2.4.1 Kommerziell vs. Kostenlos

2.4.2 Hosted vs. On-Premise

2.5 Mögliche Verbesserungen

Gated Checkin and Pull-trigger

[8] Food and Drug Administration, eine US Amerikaische Behörde ähnlich dem deutschen Gesundheitsministeriums

3 Jenkins

3.1 Funktionsumfang

3.2 Erweiterungsmöglichkeiten

In den Gründen wurde AuditTrail erwähnt, hier ein Plugin dazu: [Har14]

4 Anwendungsbeispiele

4.1 Verteilte Teams in unterschiedlichen Zeitzonen

4.2 Regulatorisch relevante Software und Nachvollziehbarkeit

Literatur

Konferenzbeiträge

[HK02] Jan Hannemann und Gregor Kiczales. „Design pattern implementation in Java and AspectJ“. In: *ACM Sigplan Notices*. Bd. 37. 11. ACM. 2002, S. 161–173.

[Rog04] R. Owen Rogers. „Scaling Continuous Integration“. In: *Extreme Programming and Agile Processes in Software Engineering*. Hrsg. von Jutta Eckstein und Hubert Baumeister. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 68–76. ISBN: 978-3-540-24853-8. DOI: 10.1007/978-3-540-24853-8_8.

[OAB12] Helena Holmström Olsson, Hiva Alahyari und Jan Bosch. „Climbing the”Stairway to Heaven-A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software“. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE. 2012, S. 392–399. DOI: 10.1109/SEAA.2012.54.

Internet

[Fow06] Martin Fowler. *Continuous Integration*. Scientific article, 01 Mai 2006. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html.

[Fow12] Martin Fowler. *TestPyramid*. Scientific article, 01 Mai 2012. 2012. URL: https://martinfowler.com/bliki/TestPyramid.html.

[Fow13] Martin Fowler. *ContinuousDelivery*. Scientific article, 30 Mai 2013. 2013. URL: https://martinfowler.com/bliki/ContinuousDelivery.html.

[Fow14] Martin Fowler. *Microservices*. Scientific article, 25 März 2014. 2014. URL: https://www.martinfowler.com/articles/microservices.html.

[Har14] Alan Harder. *Audit Trail Plugin*. Version 2.1 (16-Sept-2014). 2014. URL: https://wiki.jenkins.io/display/JENKINS/Audit+Trail+Plugin.

[Pau15] Harm Pauw. *UNTERSCHIEDE ZWISCHEN CONTINUOUS INTEGRATION, CONTINUOUS DELIVERY UND CONTINUOUS DEPLOYMENT*. Overview and relations. 2015. URL: https://www.scrum.de/unterschiede-zwischen-continuous-integration-continuous-delivery-und-continuous-deployment/.

[Nim16] Vora Nimish. *Continuous Integration (CI) & Continuous Deployment (CD) – Bandwagon of Agile Development*. Marketing Blog entry, 08 November 2016. 2016. URL: https://volansys.com/continuous-integration-continuous-deployment-bandwagon-of-agile-development/.

Abkürzungsverzeichnis

CI Continuous Integration

SCM Source Control Management