FernUniversität in Hagen 01919 Seminar Programmiersysteme

Continuous Integration und Jenkins

Seminararbeit im Studiengang 'Bachelor Informatik'

vorgelegt von Daniel Wolfschmidt

Betreuer: Daniela Keller

Ablieferungstermin: 3. Juli 2018

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

Inhaltsverzeichnis

1	Einl	eitung	4			
2	Con	tinuous Integration	5			
	2.1	Begriffsklärung				
		2.1.1 Abgrenzung zu anderen Begriffen				
	2.2	Ablauf von Continuous Integration	7			
	2.3	Gründe Continuous Integration einzusetzen	11			
	2.4	Mögliche Verbesserungen				
3	Tool	Tools zur Unterstützung von CI				
	3.1	Kommerziell vs. Kostenlos	13			
	3.2	Hosted vs. On-Premise	14			
	3.3	Bekannte Vertreter	14			
		3.3.1 Jenkins	15			
		3.3.2 TeamCity				
		3.3.3 TFS	17			
		3.3.4 Travis-CI	18			
4	Jenk	cins	19			
	4.1	Geschichte	19			
	4.2	Möglichkeiten des Betriebs	19			
		4.2.1 Installation direkt im Betriebssystem				
		4.2.2 Vorprovisionierte Container				
	4.3	Funktionsumfang				
	4.4	Erweiterungsmöglichkeiten	21			
5	Anw	Anwendungsbeispiele 2				
	5.1	Verteilte Teams in unterschiedlichen Zeitzonen	22			
	5.2	Regulatorisch relevante Software und Nachvollziehbarkeit	22			
Lit	teratu	ır	23			
Αŀ	okürz	ungsverzeichnis	25			

Abbildungsverzeichnis

2.1	Übersicht über verschiedene Begriffe [Nim16]	8
2.2	Schematischer Ablauf von CI	8
3.1	Jenkins Weboberfläche [Jen17]	15
3.2	Teamcity Weboberfläche [Jet18]	16
3.3	TFS Weboberfläche [Mic17]	17
3.4	Travis Weboberfläche [Tra18a]	18
4.1	Unterstützte Betriebssystem auf der Jenkins Seite[Jen18a]	20

Tabellenverzeichnis

3.1	Jenkins Fakten	15
3.2	TeamCity Fakten [Jet18]	16
3.3	TFS Fakten [Mic17]	17
3.4	Travis Fakten [Tra18b]	18

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 Microservices¹. 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.

¹Dabei 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]

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 Merge²-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

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

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-Pyramide³ 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 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 CI⁴ 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. Des weiteren setzt 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.

³Dabei 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]

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

2.1.1 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 (CD⁵) 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 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 Vorund Nachteilen.

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

 $^{^5}$ Ab hier werde ich CD als Abkürzung für "Continuous Delivery" verwenden. Diese Abkürzung ist auch im Abkürzungsverzeichnis zu finden.

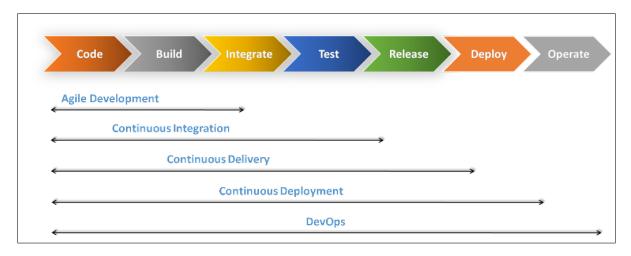


Abbildung 2.1: Übersicht über verschiedene Begriffe [Nim16]

Basis ist ein sogenanntes Source-Control-Management-System (SCM⁶). 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.

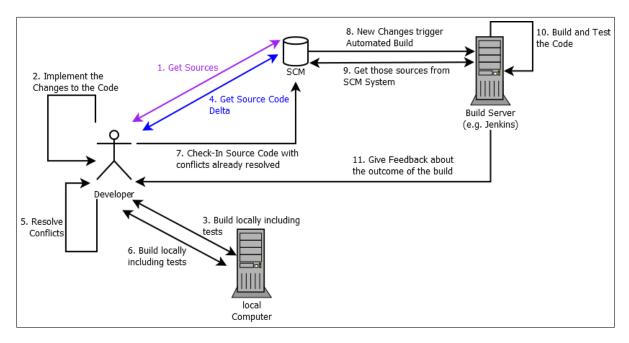


Abbildung 2.2: Schematischer Ablauf von CI

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

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 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 Pattern⁷ 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 ?? 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 Bibliotheken, 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.

 $^{^7}$ 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]

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 FDA⁸ 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. Das verkürzt den Restprozess bis zum Release und steigert auch das Vertrauen in den aktuellen Stand, da dieser regelmäßig und mehrfach getestet ist. Wenn aus dem Markt nun ein besonders schwerwiegender Fehler gemeldet wird, kann man kurzfristig einen (Patch-)Release ansetzen und durchführen.

5. Vorsichtigere Entwickler, wenn sie wissen dass eine Kontrollinstanz existiert

Auch die Einstellung der Entwickler ändert sich. Alleine durch das Wissen, dass es eine zentrale Kontrollinstanz gibt, gehen sie bewusster und vorsichtiger mit Codeänderungen um. Jeder im System kann sehen, aufgrund welches Check-Ins der Build auf einmal nicht mehr funktioniert. Schon allein weil man vor den Kollegen nicht als schlechter Entwickler identifiziert werden möchte, achtet man mehr auf seine Check-Ins, baut und testet lokal. Aufgrund dieses vorsichtigeren Ansatzes werden auch die Checkins vom Umfang her kleiner. Das System ist automatisiert und es macht von dieser Seite keinen Unterschied ob man viel oder wenig ändert. Kleine Änderugnen lassen sich jedoch leichter korrigieren bei einem Fehler und auch leichter kontrollieren im Handling. Das führt insgesamt zu einer besseren Entwicklungskultur im Unternehmen und zu besserer Performance der Mitarbeiter.

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

2.4 Mögliche Verbesserungen

Das Konzept CI wurde bereits 2006 von Martin Fowler vorgestellt. Seitdem gab es bereits mehrere Ansätze der Weiterentwicklung. Ich möchte hierbei einige vorstellen, sowohl auf Seiten des Prozesses als auch, solche die durch neue Funktionen von Tools ermöglicht wurden:

• Erweiterung des Prozesses

Aufgrund der immer kürzeren Entwicklungszyklen wird es in manchen Bereichen, wie z.B. Appentwicklung für mobile Geräte, unasuweichlich möglichst viel der Arbeit zu automatisieren. Deshalb wurde bereits die Erweiterung des Konzepts zu "Continuous Delivery" bzw. "Continuous Deployment" entwickelt. Hierfür gibt es auch Toolunterstützung von z.B. Jenkins.

• Verbesserung des vorhandenen Prozesses

Eine weitere mögliche Verbesserung setzt viel früher im Prozess an. In dem hier vorgestellten klassischen CI Prozess, fügt ein Entwickler seine Änderungen in das SCM System ein und anschließend wird die Qualität durch einen automatisierten Build ermittelt. Das kann aber gerade im Fall eines mangelhaften CheckIns zu Problemen führen, da andere Entwickler diesen korrupten Stand aus dem SCM holen und eventuell in ihre Änderungen einbauen.

Deshalb gibt es bereits vorhandene Konzepte die Änderungen zu prüfen, bevor diese in das SCM gelangen. Je nach verwendetem Tooling heißen diese "Gated-Checkin" (TFS") oder "Pretested Commits" (Jenkins).

 $^{^{9}}$ Microsoft Team Foundation Server, kommerzielles Tool zur Unterstützung von CI

Tools zur Unterstützung von CI

Es gibt eine sehr große Menge an angebotenen Tools am Markt, um CI zu unterstützen. In diesem Kapitel möchte ich vor allem ein paar dieser Tools kurz vorstellen und anhand einiger Merkmale kategorisieren.

3.1 Kommerziell vs. Kostenlos

Grundsätzlich kann man zwischen kostenlosen (meist Open Source) und proprietären, kommerziellen Tools unterscheiden. Es gibt sowohl Gründe für die eine wie auch die andere Art. Ich möchte hier einige Aspekte aufführen, welche eine Toolentscheidung in die eine oder andere Richtung lenken:

Technologie

Zunächst ist zu erwähnen, dass bestimmte Rahmenbedingungen für Unternehmen beziehungsweise Projekte vorgegeben sind. Als Beispiele möchte ich hier bestimmte zu verwendende Technologien erwähnen. Es kann unumgänglich sein ein bestimmtes kommerzielles Produkt zu verwenden, weil nur für diese Technologie zertifizierte Tools erlaubt sind, was meist mit hohem finanziellem Aufwand verbunden ist. Kostenlose Projekte können dies nicht für jede Version leisten und lassen sich deshalb nicht zertifizieren.

Usability

Funktional sind kostenlose Tools meist sehr weit entwickelt. Sie sind nicht selten sogar funktional besser als kommerzielle Tools, weil die Entwicklergemeinde selbst die Funktionalität nach vorne treibt und sogar neueste Entwicklungen einfließen. Ein Manko in vielen dieser Tools ist jedoch die Usability. Nur die größten der kostenlosen Tools schaffen es aufgrund des riesigen Entwicklerhintergrunds, auch die Usability im Auge zu behalten. Ein Beispiel ist Jenkins, bei dem lange Zeit UE/UX das größte Manko war und erst seit Einführung von "Blue Ocean" 2016 ist auch der Usability Aspekt in diesem sehr großen Projekt präsent.

Regulatorische Einflüsse

Ein weiterer Aspekt, der Aufmerksamkeit bedarf sind regulatorische Rahmenbedingungen.

Rechtliche Bedingungen

In der Softwareentwicklung gerade in sehr großen Unternehmen, sind die Kosten für ein System eher zu vernachlässigen. Viel gewichtiger sind rechtliche Absicherungen die man sich durch den Einsatz eines Tools eines finanzkräftigen Unternehmens einkauft. Falls ein Fehler

auf einen Bug des CI Systems zurückzuführen ist, und das Unternehmen mit einem prozentualen Anteil des hohen Jahresumsatzes haften muss, so besteht prinzipiell die Chance etwas zurückzuholen. Bei einem kostenlosen Tool existiert dieser finanuzielle Hintergrund nicht. Deshalb ist eine Tendenz zu erkennen, je größer das Unternehmen ist, desto wahrscheinlicher ist der Einsatz eines kommerziellen Tools.

3.2 Hosted vs. On-Premise

Auch dabei, ob man sich selbst um die Infrastruktur des Servers kümmert, oder ein fertiges Produkt mietet, kann man unterscheiden. Dabei gibt es mehrere Aspekte, die beachtet werden müssen, wobei ich einige herausgreifen möchte:

• Finanzielle Aspekte

Von finanzieller Seite, muss man klar unterscheiden. Eigene Infrastuktur kostet erstmal Anschaffungskosten, verursacht dann aber auch während des Betriebs laufende Kosten wie Strom, Wartung und Arbeitszeit (was indirekt auch Kosten sind). Auf der anderen Seite bekommt man bei On-Premise Angeboten alles aus einer Hand, trägt kein finanzielles Risiko für Hardwareausfall, und muss dafür einen monatlichen Beitrag zahlen. Für eher kleine Projekte ist es wohl eher von Vorteil On-Premise zu verwenden, in großen Unternehmen und Projekten, wo eine Person sich nur um Build Automatisierung kümmern kann, ist dann wohl eher der eigene Server besser.

· Nachvollziehbarkeit und Reproduzierbarkeit

Ein anderer Aspekt ist vor allem für regulatorische Umgebungen wichtig. Hierbei muss es möglich sein, nachzuvollziehen was alles Auswirkungen auf das Ergebnis des Builds hatte. Dazu gehört auch die Build Umgebung selbst. Diese ist jedoch bei vielen On-Premise Angeboten nicht eindeutig zu identifizieren, da diese neue Features einfach live schalten. Auch ist es unmöglich einen Build mit einem bestimmten Stand der Build Automatisierung nochmals laufen zu lassen nach längerer Zeit. Man hat keine Chance, den alten Build Server wiederherzustellen.

· Datenschutz und Datensicherheit

Ein nicht zu vernachlässigender Aspekt ist die Sicherheit der Daten und Unternehmensgeheimnisse. Wenn man die Daten aus der Hand gibt, um aus Source Code wirklich ausführbare Programme zu machen, so gibt man auch sein wertvollstes Gut aus der Hand. Man muss dem Unternehmen, das die Services anbietet schon besonders trauen, beziehungsweise sollte es finanzstark sein, um bei Verstößen gegen Schutzbedürfnisse entsprechende Entschädigungen zu zahlen. Die sicherste Methode ist, den Code nicht aus der Hand zu geben und selbst einen Build Server zu hosten. Wenn das nicht möglich ist, sollte auf einen sehr vertrauenswürdigen Partner geachtet werden.

3.3 Bekannte Vertreter

Im Folgenden eine kleine Auswahl an Tools, die jedoch keinen Anspruch auf Vollständigkeit erhebt. Ich habe mich auf bekannte Vertreter beschränkt.

3.3.1 Jenkins

Bei Jenkins handelt es sich um ein OpenSource Produkt. Es wurde 2004 gestartet, war zwischenzeitlich im Besitz von Oracle unter dem Namen Hudson und wurde dann geforkt¹⁰ unter dem Namen Jenkins, unter dem es heute bekannt ist.

Kategorie	Wert
Firma	"The Jenkins Project". Dies ist eine rechtliche Dachorganisation und hält die Marke "Jenkins"
Kosten	Es ist ein Open Source Projekt. Es fallen keine Kosten für die Software selbst an. Nur die Server auf denen Jenkins betrieben wird müssen bei manchen Angeboten (z.B. Cloud) bezahlt werden.
Version	2.121.1 (LTS), 2.129 (Weekly)
SCM Systeme	GIT, TFVC, PVCS, Mercurial, Subversion,
Erweiterbar	Ja, via Plugin System

Tabelle 3.1: Jenkins Fakten

Ein Beispiel für das Userinterface von Jenkins ist in Abbildung 3.1 zu sehen.

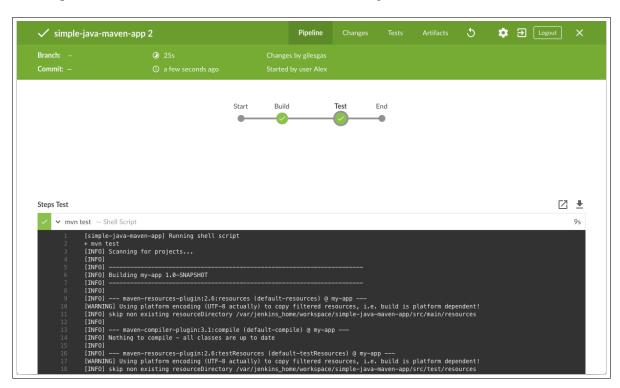


Abbildung 3.1: Jenkins Weboberfläche [Jen17]

¹⁰Forken ist eine besondere Art des Abspaltens von Entwicklungszweigen, bei dem ein Projekt in mehreren Folgeprojekten resultiert

3.3.2 TeamCity

TeamCity ist ein kommerzielles Produkt, das vor allem aus der .NET Welt kommt. Es ist in Java geschrieben und wurde bereits 2006 veröffentlicht. Jetbrains ist eine Firma, die viele Entwicklertools anbietet, um die Produktivität zu erhöhen.

Kategorie	Wert
Firma	JetBrains s.r.o., registered seat at Na hřebenech II 1718/10, 14700 Prague 4, Czech Republic
Kosten	Teamcity ist ein proprietäres Produkt. Es wird pro Build Agent lizenziert. Der Preis liegt bei etwa 300€ pro Build Agent. Open Source Projekte können das Produkt kostenlos nutzen.
Version	2018.1
SCM Systeme	GIT, TFVC, Mercurial, Subversion,
Erweiterbar	Ja, via Plugin System. Jedoch deutlich weniger Plugins vorhanden als bei Jenkins. Man kann auch selbst Plugins entwickeln.

Tabelle 3.2: TeamCity Fakten [Jet18]

Ein Beispiel für das Userinterface von TeamCity ist in Abbildung 3.2 zu sehen.

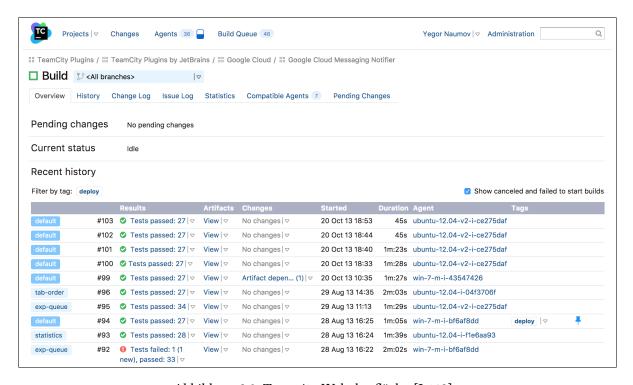


Abbildung 3.2: Teamcity Weboberfläche [Jet18]

3.3.3 TFS

Der Teamfoundation Server ist die kommerzielle CI Lösung von Microsoft. Historisch kommt es aus dem .NET Umfeld, entwickelt sich aber in den letzten Jahren immer weiter in Richtung Open Source und CrossPlatform. So ist zum Beispiel die letzte Version der TFS Build Automatisierung lauffähig auf vielen verschiedenen Platformen wie z.B. Windows, Linux, MacOS.

Auch sind große Teile der Build Automatisierung OpenSource und auf GitHub verfügbar.

Kategorie	Wert
Firma	Microsoft
Kosten	TFS ist ein kommerzielles Produkt. Es wird pro Nutzer lizenziert, wobei rudimentäre Aufgaben mit der "Stakeholder Lizenz" kostenlos sind. Sobald man auch die Build Funktionalität nutzen will, braucht man eine bezahlte Lizenz.
Version	2018
SCM Systeme	GIT, TFVC. Allgemein sehr limitiert, denkbar ist eine manuelle Anbindung, was aber dann von den Tools nicht besonders gut unterstützt wird.
Erweiterbar	Ja, an vielen Stellen. Sowohl auf Server- als auch auf Clientseite erweiterbar.

Tabelle 3.3: TFS Fakten [Mic17]

Ein Beispiel für das Userinterface von TFS ist in Abbildung 3.3 zu sehen.

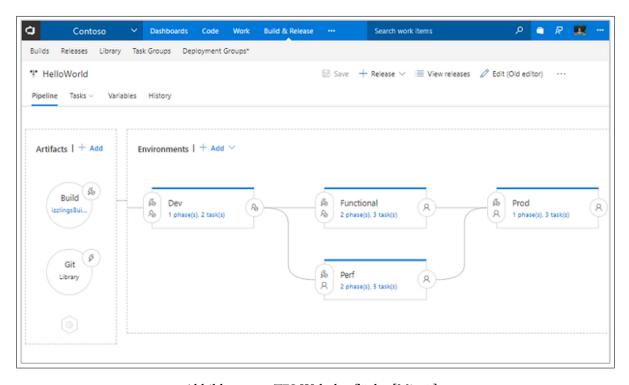


Abbildung 3.3: TFS Weboberfläche [Mic17]

3.3.4 Travis-CI

Travis-CI ist ein reiner On-Premise OnlineService. Er erlaubt es automatisch mit jedem GitHib push einen Build zu starten und zu testen.

Kategorie	Wert
Firma	TRAVIS CI, GMBH
Kosten	Travis-CI ist OpenSource. Es ist kostenlos für OpenSource Projekte. Für alle anderen ist es kostenpflichtig.
Version	2.2.1
SCM Systeme	Nur GitHub
Erweiterbar	Man kann per yaml den Build Agent so konfigurieren, dass automatisch Compiler usw. installiert werden, aber direkte Erweiterbarkeit gibt es nicht.

Tabelle 3.4: Travis Fakten [Tra18b]

Ein Beispiel für das Userinterface von Travis ist in ?? zu sehen.

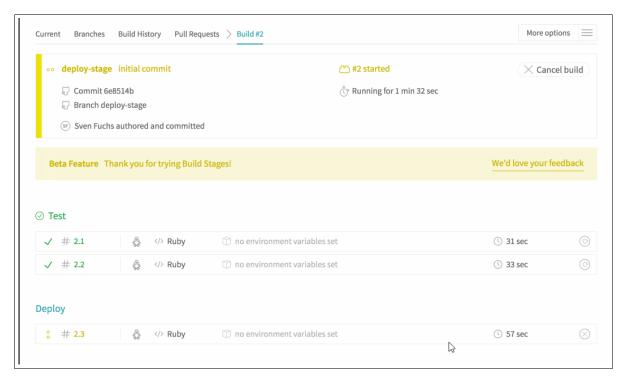


Abbildung 3.4: Travis Weboberfläche [Tra18a]

Jenkins

Bei Jenkins handelt es sich um einen der bekanntesten Vertreter der OpenSource CI Tools. Es hat eine sehr breite Anwender Basis und dementsprechend auch viele nützliche Blogeinträge und Hands-On Berichte.

4.1 Geschichte

2004 startete Kohsuke Kawaguchi damit, Jenkins zu implementieren. Damals noch unter dem Namen "Hudson" und als privates Projekt. Eigentlich wollte er nur ein Tool für sich selbst entwickeln, das ihm half, qualitativ hochwertigen Code in das SCM einzufügen. (Foreword of [Sma11]).

Das war im Jahr 2004, und er führte das Projekt neben seiner Arbeit für SUN als privates Projekt bis 2008 weiter. Zu diesem Zeitpunkt entdeckte sein Arbeitgeber das Potential dieses Tools, und er wurde gefragt, ob er nicht seine komplette Arbeitszeit für das Tool verwenden möchte, um es auf eine professionellere Ebene zu heben. Er willigte ein und bis zum Jahr 2010 erreichte Hudson einen Marktanteil von 70%.[Sma11, S. 3]

In der Zwischenzeit hatte Oracle die Firma SUN gekauft und damit auch Hudson. Ende 2010 kam es zum Zerwürfnis zwischen den OpenSource begeisterten Kernentwicklern und Oracle, in welche Richtung das Projekt ausgerichtet werden sollte. Die Meinungsverschiedenheiten, konnten nicht bewältigt werden, so dass im Januar 2011 die ursprünglichen Hudson Entwickler unter der Führung von Kohsuke Kawaguchi einen Fork bei GitHub erstellten unter dem Namen "Jenkins". Auch die meisten der bisherigen Hudson Nutzer blieben dem Projekt treu und so wechselten 75% der Hudson Nutzer zu Jenkins. [Sma11, S. 3–4]

4.2 Möglichkeiten des Betriebs

Anfänglich war das Betreiben einer Jenkins Installation ziemlich eindeutig vorgegeben. Man musste gewisse Voraussetzungen erfüllen, wie z.B. Java und ein SCM Client und installierte dann lokal auf einem Rechner. Mittlerweile gibt es jedoch, aufgrund neuer Technologien, auch andere Ansätze, von denen ich hier drei vorstellen möchte.

4.2.1 Installation direkt im Betriebssystem

Der klassische Weg einen Jenkins CI Server zu betreiben ist die Installation direkt in ein Betriebssystem.

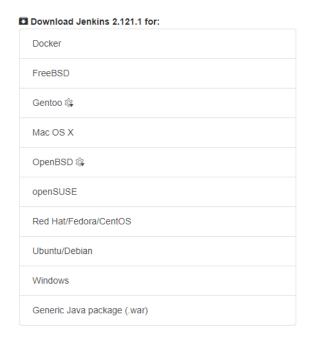


Abbildung 4.1: Unterstützte Betriebssystem auf der Jenkins Seite[Jen18a]

Dazu benötigt man vor allem Java, wobei die einzige momentan unterstützte Version Java 8 ist.[Jen18d] Desweiteren mindestens 256MB RAM (1GB empfohlen) und 1GB (50GB empfohlen) Massenspeicher.[Jen18b]

Windows

Die Jenkins Seite bietet direkt einen Windows Installer für Jenkins an. Dieser installiert Jenkins als Windows Service, der keine Nutzer Interaktion benötigt und direkt beim Systemstart mit gestartet wird. Auch die Agents die die Builds ausführen können direkt als Windows Service gestartet werden. [Jen18c]

Außerdem wird die Installation von "UnixUtils"¹¹ empfohlen, da Jenkins zunächst für Unixbasierte Plattformen entwickelt wurde und an machen Stellen deshalb Unix Tools voraussetzt.

Linux

Unter Linux kann man Jenkins ähnlich wie bei Windows als ständig laufenden Prozess installieren. Dazu erstellt man am besten einen eigenen Service Nutzer und erstellt einen deamon um Jenkins direkt beim Systemstart zu starten. [Jen18b]

Andere Systeme auf denen Java läuft

Für viele andere Systeme, und als Alternative für die bereits genannten direkten Serviceinstallationen in Linux und Windows, bietet Jenkins ein generisches Java Paket an (Siehe letzte Option in ??). Diese WAR Datei kann als extra Prozess oder unterhalb eines Webcontainers wie Apache Tomcat gestartet werden [Jen18b]

4.2.2 Vorprovisionierte Container

Das Jenkins Projekt bietet auch vorprovisionierte Container an. In ?? ist der Download des Docker Containers zu sehen. Bei einem Docker Container handelt es sich um ein eigenständiges Paket, das

¹¹Das sind Tools die Unix-like Funktionalität nachbilden, Download unter http://unxutils.sourceforge.net/, empfohlen hier: https://wiki.jenkins.io/display/JENKINS/Installing+Jenkins

eine Applikation und alle dazu nötigen Ressourcen enthält. Es nutzt das darunterliegende Betriebssystem um auf Systemressourcen zuzugreifen. Man kann es in etwa mit Apps auf dem Handy vergleichen.

Docker existiert bereits seit längerer Zeit für alle nix basierten Betriebssysteme, und seit Windows 10 Professional mit aktieiertem Hyper-V auch auf Windows verfügbar. Die Installation von Jenkins entfällt, man muss nur den Container starten und hat einen funktionierenden Jenkins Server.

4.3 Funktionsumfang

4.4 Erweiterungsmöglichkeiten

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

Anwendungsbeispiele

- 5.1 Verteilte Teams in unterschiedlichen Zeitzonen
- 5.2 Regulatorisch relevante Software und Nachvollziehbarkeit

Literatur

Bücher

[Sma11] John Ferguson Smart. Jenkins: The Definitive Guide: Continuous Integration for the Masses. "O'Reilly Media, Inc.", 2011. ISBN: 978-1-449-30535-2.

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/.
- [Jen17] Jenkins. Build a Java app with Maven. Example config page for Jenkins Java Build with Maven. 2017. URL: https://jenkins.io/doc/tutorials/build-a-java-app-with-maven/.
- [Mic17] Microsoft. Team Foundation Server 2018: Anmerkungen zu dieser Version. Release Notes TFS2018. 2017. URL: https://docs.microsoft.com/de-de/visualstudio/releasenotes/tfs2018-relnotes.
- [Jen18a] Jenkins. Getting started with Jenkins. Download Page of Jenkins. 2018. URL: https://jenkins.io/download/.
- [Jen18b] Jenkins. Installing Jenkins. Different installation instructions for Jenkins. 2018. URL: https://jenkins.io/doc/book/installing/.
- [Jen18c] Jenkins. Installing Jenkins as a Windows service. Wiki entry about Jenkins on Windows and running as a Service. 2018. URL: https://wiki.jenkins.io/display/JENKINS/Installing+Jenkins+as+a+Windows+service.
- [Jen18d] Jenkins. Java requirements. Site explaining all Java related constraints of Jenkins. 2018. URL: https://jenkins.io/doc/administration/requirements/java/.
- [Jet18] Jetbrains. *TeamCity*. Marketing page of Teamcity. 2018. URL: https://www.jetbrains.com/teamcity/.
- [Tra18a] Travis-CI. Build Stages. Documentation Wiki of Travis-CI. 2018. URL: https://docs.travis-ci.com/user/build-stages/.
- [Tra18b] Travis-CI. Travis-CI Documentation. Documentation Wiki of Travis-CI. 2018. URL: https://docs.travis-ci.com/.

Abkürzungsverzeichnis

CI Continuous Integration

TFS Team Foundation Server

CD Continuous Delivery

SCM Source Control Management

SCM Source Control Management