

2 CI in 20 Minuten

Erinnern Sie sich noch an die Integrationshölle, die das Softwareteam zu Beginn des vorausgegangenen Kapitels durchschreiten musste? Wie Sie richtig vermuten, könnte CI viele der beschriebenen Probleme lösen.

In diesem Kapitel lernen Sie daher zunächst das CI-Konzept in einer knappen Definition kennen: Was also *ist* CI? Wir fragen aber auch: Was ist CI *nicht*? Anschließend kehren wir zu »unserem« Softwareteam in einem Vorher-Nachher-Vergleich zurück und überprüfen, wie CI dieses Team im Alltag unterstützen kann.

Nach der Lektüre dieses Kapitels kennen Sie also die Grundzüge der CI und haben einen ersten Eindruck von deren Vorteilen, aber auch von den notwendigen Voraussetzungen. Sie schaffen sich damit eine gute Grundlage für die darauffolgenden zwei Kapitel, die dann auf Vorteile und Voraussetzungen im Detail eingehen werden.

2.1 Was ist CI?

Zahlreiche Softwareteams dürften unabhängig voneinander die Idee des kontinuierlichen Integrierens »erfunden« haben. Populär geworden ist sie unter dem Namen »Continuous Integration« jedoch erst als eine der Praktiken der Extremprogrammierung (*eXtreme Programming*, kurz: XP), wie beispielsweise durch Kent Beck beschrieben [Beck99].

Begriffsbildend dürfte letztendlich Martin Fowlers Artikel »Continuous Integration« gewesen sein, in dem er das Kernkonzept folgendermaßen beschreibt:

CI nach Martin Fowler

»Die Kontinuierliche Integration ist eine Softwareentwicklungspraktik, bei der Teammitglieder ihre Arbeit häufig integrieren. Üblicherweise integriert jede Person im Team mindestens einmal täglich was zu mehreren Integrationen am Tag führt. Jede Integration wird durch einen vollautomatisierten Build (und Test) geprüft, um Fehler so

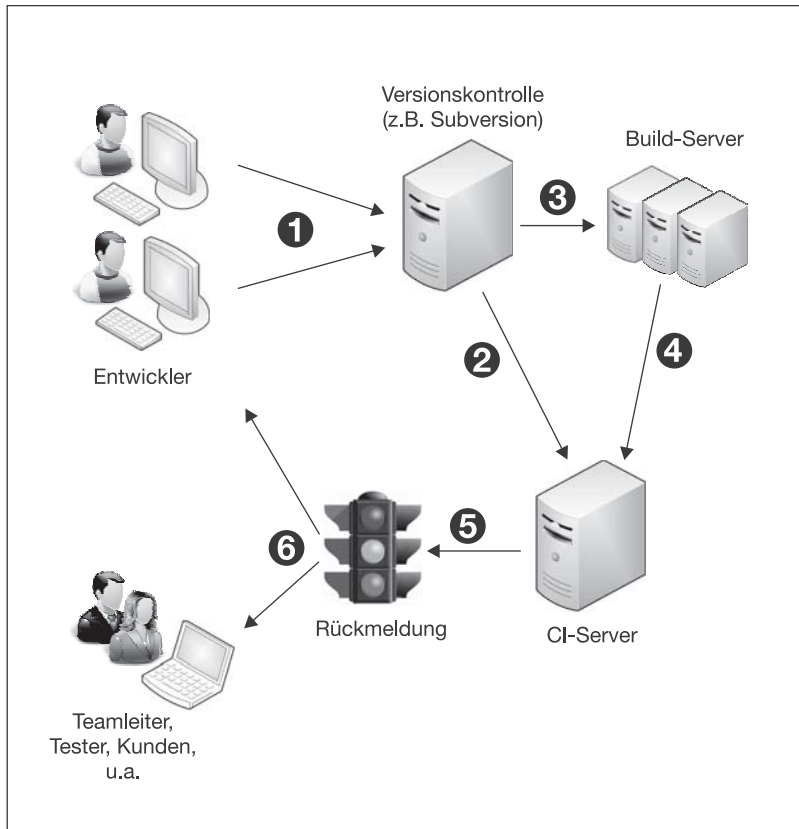
schnell wie möglich aufzudecken« ([Fowler00], aus dem englischen Original übersetzt).

Im Kern geht es also darum, die Integration aller Codeänderungen nicht mehr als eine einmalige Phase am Ende eines Entwicklungsprojekts zu betrachten. Vielmehr soll diese »Endmontage« einer Software viel häufiger erfolgen, typischerweise sogar mehrfach am Tag. Im Idealfall wird die Integration damit zum »Nicht-Ereignis«, da sie ohnehin ständig stattfindet. Vorteil: Wer mehrmals täglich den Ernstfall probt, hat auch keine Angst mehr davor.

Der Entwicklungsprozess
mit CI

Wie sieht also der Entwicklungsprozess in einem minimalen CI-System aus? Betrachten wir dazu den Ablauf der kontinuierlichen Integration (CI) in Abbildung 2-1:

Abb. 2-1
Ablauf der
kontinuierlichen
Integration (CI)



Neuer Code entsteht zunächst auf den Rechnern der Entwickler. Nach eingehender lokaler Überprüfung durch den Entwickler werden alle Codeänderungen an ein zentrales Versionskontrollsystem übertragen (1). Dieser Vorgang wird vom CI-Server bemerkt (2), der daraufhin

einen neuen Build auf Build-Servern veranlasst (3). Nach Abschluss des Builds werden die Ergebnisse (Programme, Testberichte, Dokumentationen usw.) auf den CI-Server zur Auswertung und Archivierung übertragen (4). Über geeignete Kommunikationskanäle (E-Mail, RSS, Issue-Tracker usw.) meldet der CI-Server den Ausgang des Builds an die Entwickler zurück (6). Diese bekommen so eine zeitnahe positive Bestätigung ihrer Arbeitsergebnisse – oder aber einen Hinweis auf neu aufgetretene Fehler. In diesem Falle kann der Entwickler sofort mit der Verbesserung seines Codes beginnen: Der CI-Zyklus beginnt von vorne. Darüber hinaus können auch andere interessierte Parteien den aktuellen Stand der Softwareentwicklung verfolgen, z.B. Teamleiter, Tester, Kunden usw.

CI-Systeme haben in der Praxis zahlreiche Schnittstellen zu weiteren IT-Systemen (z.B. zentrales Benutzerverzeichnis, Test- und Produktionsserver, Software-Repositories). In kleineren Arbeitsgruppen werden die benötigten Dienste oftmals auf derselben Hardware ausgeführt, in großen Unternehmen werden hingegen eigens dafür vorgesehene Server eingesetzt. Das Funktionsprinzip ist jedoch das gleiche und setzt hinsichtlich des Entwicklungsprozesses gewisse Rahmenbedingungen voraus. Genau betrachtet können diese fast vollständig aus der Anforderung des häufigen Bauens (und Testens) abgeleitet werden. Fowler zählt in seinem Artikel folgende zehn Praktiken als Voraussetzung für wirkungsvolles CI auf:

Voraussetzungen für CI

1. *Gemeinsame Codebasis:*
Alle Daten, die in ein Produkt eingehen, werden an einem gemeinsamen Ort verwaltet, typischerweise in einem Versionskontrollsystem.
2. *Automatisierter Build:*
Das Produkt muss vollautomatisch aus seinen Grundbestandteilen übersetzt und zusammengebaut werden können.
3. *Selbsttestender Build:*
Entstandene Produkte werden während des Builds automatisch auf korrekte Funktionsweise überprüft.
4. *Häufige Integration:*
Entwickler integrieren ihre Arbeitsergebnisse mindestens einmal pro Tag. Anders ausgedrückt: Sie checken mindestens einmal am Tag ihren Stand in das Versionskontrollsystem ein.
5. *Builds (und Tests) nach jeder Änderung:*
Nach jeder Änderung wird vollautomatisch gebaut und getestet. Sind die Änderungen klein, lassen sich so neue Fehler sehr schnell in den korrespondierenden Änderungen auffinden.

6. *Schnelle Build-Zyklen:*
Zwischen dem Einchecken einer Änderung und der Rückmeldung durch das CI-System sollte möglichst wenig Zeit vergehen. Idealerweise liegt diese Dauer im Minutenbereich.
7. *Tests in gespiegelter Produktionsumgebung:*
Tests sollten in einer möglichst realitätsnahen Umgebung stattfinden. In der Konsequenz muss also auch der Aufbau dieser Testumgebungen automatisiert werden.
8. *Einfacher Zugriff auf Build-Ergebnisse:*
Der letzte Stand des Produkts sollte für alle Beteiligten einfach zugänglich sein. Dies gilt nicht nur für Entwickler, sondern auch für Tester, Kundenberater, Projektleiter usw.
9. *Automatisierte Berichte:*
Die Ergebnisse der Builds müssen vollautomatisch »gebrauchsfertig« aufbereitet werden. Zum einen beinhaltet dies das Verschicken von Benachrichtigungen, zum anderen auch die Visualisierung von archivierten Informationen aus vergangenen Builds.
10. *Automatisierte Verteilung:*
Ein CI-System sollte nicht nur den Softwareerstellungsprozess automatisieren, sondern auch die Verteilung zu den jeweiligen Anwendern bzw. das Ausbringen auf Test-, Demonstrations- und Produktionsserver.

Wenn Sie sich die Liste der zehn Praktiken betrachten, fragen Sie sich vermutlich, ob hier nicht zu viel vorausgesetzt wird? Muss nicht zu viel umgestellt werden? Lohnt sich dieser Aufwand?

Um es vorwegzunehmen: Es lohnt sich auf jeden Fall! Jede dieser Praktiken ist bereits für sich isoliert gesehen wertvoll und wird vielleicht sogar bereits in Ihrer Arbeitsgruppe praktiziert. Erst in der Summe hingegen ermöglichen sie einen runden Entwicklungsprozess, der auf hoher Frequenz kontinuierlich wechselt zwischen Entwickeln, Einchecken, Bauen, Testen, Berichten und erneut Entwickeln, Einchecken, Bauen, ...

2.2 Was ist CI nicht?

Wie wir im vorausgegangenen Abschnitt gesehen haben, beinhaltet CI eine weitestgehende Automatisierung typischer Arbeitsschritte der Softwareentwicklung, z. B. Kompilieren, Testen, Archivieren, Verteilen, Berichten (denken muss man glücklicherweise noch selber). Kein Wunder also, dass kaum ein anderer Dienst mit so vielen anderen IT-Systemen in Kontakt steht wie ein CI-Server: vom Versionskontrollsys-

tem bis zum E-Mail-Server, von der Benutzerverwaltung bis zum Produktionsserver.

Im Zusammenspiel dieser IT-Systeme übernimmt der CI-Server sozusagen die Rolle des Dirigenten im »Build-Orchester«. Sicher – man kann auch ohne CI brauchbare Software erstellen. Und genauso würden die Berliner Philharmoniker auch ohne Dirigent locker vom Blatt musizieren können. Ungleich besser wird das Ergebnis jedoch unter der Leitung eines Sir Simon Rattle. Rattle selbst greift dabei weder zur Bratsche noch zur Oboe. Stattdessen überlässt er dies den jeweiligen Virtuosen. Trotzdem ist sein »Mehrwert« so groß, dass er als Dirigent auf Konzertplakaten zuerst aufgeführt wird. Analog gilt für Continuous Integration:

Das Build-Orchester

- CI ist *keine Programmiersprache* wie C/C++, Python oder Java. Sie steuert aber die entsprechenden Compiler, um Quelltexte zu übersetzen.
- CI ist *kein Build-Werkzeug* wie Make, Ant oder Maven. Sie ruft aber typischerweise solche Werkzeuge auf.
- CI ist *kein Versionskontrollsystem* wie Subversion, CVS, Git oder Perforce. Sie kommuniziert aber mit ihnen, um von Codeänderungen der Entwickler zu erfahren.
- CI ist *kein Test-Framework* wie JUnit, TestNG oder Selenium. Sie ruft aber solche Frameworks auf, um übersetzte Programme zu testen.
- CI ist *kein Werkzeug für statische Codeanalyse* wie PMD, Checkstyle oder FindBugs. Sehr wohl aber werden diese Werkzeuge in einem CI-Build aufgerufen und deren Ergebnisse ausgewertet.
- CI ist *kein Repository* für erzeugte Artefakte wie Nexus oder Artifactory. Sie kann aber Produkte eines CI-Builds dort ablegen und so anderen IT-Systemen und Entwicklern bereitstellen.
- CI ist *kein einzelnes Produkt*, sondern integriert vielmehr eine Vielzahl von Technologien für die jeweiligen Arbeitsschritte der Softwareerstellung (Compiler, Build-Werkzeuge, Testverfahren usw.)
- CI ist *keine markengeschützte Methode*, für die sich Zertifikate sammeln lassen. In der agilen Softwareentwicklung ist CI jedoch nicht wegzudenken und auch in anderen Methoden sinnvoll einsetzbar.

Und: CI ist kein Allheilmittel gegen schlechten Code, fehlerhafte Software und enge Projektzeitpläne. Sie hilft aber, Probleme schneller zu entdecken und früher zu beheben. Dies spart Zeit in nachgelagerten Phasen und vergrößert den zeitlichen Puffer für Unvorhergesehenes (»Alle mal herhören: Unser Kunde hat gerade angerufen ...«).

Nachdem wir uns vom CI-Ablauf und der eingesetzten Technik ein Bild gemacht haben, wollen wir in den nächsten zwei Abschnitten CI an einem Vorher-Nacher-Beispiel betrachten.

2.3 Software entwickeln ohne CI

Zu Beginn des vorangegangenen Kapitels hatten wir ein Softwareteam in der heißen Phase der Release-Integration besucht. Erinnern Sie sich noch an die Tiefpunkte, mit denen dieses Team ohne Continuous Integration zu kämpfen hatte? Hier ein »Worst of«:

Integration als besondere Aktivität

»Am kommenden Tag soll Ihre Abteilung die längst überfällige neue Version Ihrer Software ausliefern ...« – Die Integration wurde bis auf den letzten Drücker hinausgeschoben, weil sie nicht als Teil des Entwicklungsprozesses betrachtet oder im Eifer der Codierung verdrängt wurde (»Der Kunde bezahlt uns fürs Programmieren und nicht fürs Dateien zusammenkopieren.«).

Späte Integration

»Die Entwickler werfen also eilig auf dem Abteilungsserver ihre Codeänderungen ab, die sie in den letzten Wochen auf ihren Rechnern erbrütet haben.« – Code wurde über Tage und Wochen in Silos lokal entwickelt. Schnittstellen und Abhängigkeiten diffundieren dabei zwangsläufig auseinander, wenn nicht immenser Abstimmungsaufwand betrieben wird (»Oh je – alle in den Teamraum! Wir müssen ganz dringend noch ein Meeting machen.«).

Tests als »Code zweiter Klasse«

»Die (spärlich) vorhandenen Tests schlagen fehl, manchmal aber auch nicht.« – Die Tests deckten nur geringe Teile des zu prüfenden Codes ab. Zusätzlich trugen unzuverlässige Tests das Übrige bei, um das Vertrauen in Testberichte zu erschüttern. Sicheres Alarmsignal hierfür sind Dialoge nach folgendem Schema: Projektleiter: »Ist das Feature komplett?« Entwickler: »Die automatischen Tests laufen eigentlich durch, aber die Tester müssten sich das trotzdem nochmals gründlich anschauen ...«

Keine Builds auf neutralem Grund

»Bei mir läuft's aber! – Ich habe dort nichts geändert!« – Jeder Entwickler arbeitet auf seinem Rechner in seinem persönlichen lokalen Ökosystem aus Werkzeugen, Einstellungen und ausgecheckten Quelltexten. Kein Wunder, dass auf unterschiedlichen Rechnern unterschiedliche Ergebnisse beobachtet werden können. Wer aber hat Recht, wenn es keine verbindliche und neutrale Instanz gibt, die hier entscheidet?

Lange Build- und Test-Zyklen

»Am späten Abend laufen die Tests endlich durch.« – Lange Build-Zeiten bedeuten weniger Chancen pro Tag, um geänderten Code zu bauen und zu testen. Das Resultat ist Programmierung im Stile von »Commit and Run«: Kurz vor Arbeitsende einchecken, dann schnell den Arbeitsplatz verlassen und am nächsten Morgen in den Trümmern

nachschauen, wie der nächtliche Build mit den Änderungen klargekommen ist. So griffig dieses Konzept sein mag, so stark limitiert es die Anzahl an Rückmeldungen zum Entwickler – in diesem Fall auf eine pro Tag.

»Als langjähriger Mitarbeiter mit Kopfmonopol kennt der Meister als Einziger die geheimen Schritte, die notwendig sind, um eine Distribution zu erstellen – und ja, nur er hat die notwendigen Werkzeuge dazu auf seinem Rechner installiert.« – Teile des Erstellungsprozesses sind nicht dokumentiert, werden manuell ausgeführt und können nur in bestimmten Rechnerumgebungen stattfinden. Folgerichtig tauchen im Ablauf der Softwareerstellung immer wieder Flüchtigkeitsfehler durch menschliche Nachlässigkeit auf. Im besten Falle sind sie nur lästig, im schlimmsten Falle aber fatal – denn durch die Abhängigkeit von wenigen Köpfen und Rechnern hat man gleich mehrere K.o.-Punkte (*single points of failure*) geschaffen.

Geringe Automatisierung
und Dokumentation
kritischer Schritte

»Das Team hat Bauchschmerzen, denn keiner weiß wirklich genau, welche Änderungen in dieses Release eingeflossen sind.« – Von Hand erzeugte Dokumentation kann selten mit dem Tempo der Änderungen in einem Softwareprojekt mithalten. Im Ergebnis erhält man Dokumentation, die erfreulicherweise zu 80% auf dem neuesten Stand ist. Leider weiß man hingegen nicht, welche 20% der Dokumentation inzwischen veraltet sind ...

Geringe Automatisierung
der Dokumentation

2.4 Software entwickeln mit CI

Besuchen wir »unser« Softwareteam ein paar Monate später. Der hohe Leidensdruck in den Integrationsphasen führte zur Installation eines CI-Servers, der nun stetig das Zusammenspiel aller beteiligten Personen und IT-Systeme koordiniert.

Seit der Vollautomatisierung der Integration erstellt der CI-Server inzwischen vollautomatisch aus den Quelltexten des Versionsmanagementsystems das fertige Produkt – genau so, wie es der Kunde letztendlich erhält: Ein Installationsprogramm, eine Liste mit den Änderungen seit der letzten Version, eine aktuelle Dokumentation der externen Schnittstellen sowie ein Prüfprotokoll des vorgeschriebenen Testlaufs. Die Integration ist zum »Nicht-Ereignis« geworden.

Integration als
»Nicht-Ereignis«

Die Entwickler wurden in der Folge geradezu süchtig nach dem »schnellen Erfolgserlebnis« und checken nun ihre Änderungen nach eingehender lokaler Prüfung sehr viel häufiger ein.

Häufige Integration,
häufiges Erfolgserlebnis

Durch zusätzliche Tests konnte die Codeabdeckung erhöht werden. Vor allem aber wurden unzuverlässige Tests überarbeitet, so dass deren Ausgang nun nicht mehr von veränderlichen Einflüssen wie

Tests als »Code erster
Klasse«

Netzwerkbandbreite oder der momentanen Auslastung des E-Mail-Servers abhängt. Und weil auf Testberichte wieder Verlass ist, erhalten fehlgeschlagene Tests sofortige Aufmerksamkeit.

Builds auf neutralem Grund

Immer noch kommt es vor, dass Entwickler auf unterschiedlichen Rechnern zu unterschiedlichen Ergebnissen kommen. Entscheidend ist aber nun, was das CI-System nach Ausführung der Tests befindet. Somit gibt es endlich einen neutralen Schiedsrichter, der – frei von Skrupel und Häme – aufgetretene Probleme und fehlgeschlagene Tests mitteilt.

Kurze Build- und Test-Zyklen

Die Build-Zeit konnte durch geschickte Verteilung und Parallelisierung auf mehrere Rechner auf unter 15 Minuten gedrückt werden. Dadurch wurde der Taktschlag ganz erheblich erhöht und langes »Codieren im Nebel« abgeschafft.

Vollautomatisierung kritischer Schritte

Durch die Automatisierung kann inzwischen selbst der Praktikant eine aktuelle Produktversion zu Demonstrationszwecken für den Vertrieb erstellen. Die erfahrenen Entwickler haben damit auch endlich wieder mehr Zeit, sich um *wirklich* knifflige Probleme zu kümmern oder den Junior-Entwicklern Tricks aus ihrem Erfahrungsschatz beizubringen.

Automatische Erstellung der Dokumentation

Die Schnittstellendokumentation wird endlich nicht mehr von Hand in einer Textverarbeitung geführt, sondern in jedem Build aufs Neue vollautomatisch aus den Quelltexten erzeugt. Seitdem hatte es wesentlich weniger Anrufe erboster Kunden gegeben, die – streng der *veralteten* Dokumentation folgend – Fehlermeldungen beim Aufruf nicht mehr vorhandener Funktionen erhalten hatten.

Gruppenstolz

Im Abteilungsflur fallen drei niedliche Lampen in Form von Gummibären auf. Sie sind in den Ampelfarben Rot, Gelb und Grün aufgestellt und zeigen rund um die Uhr die Testergebnisse des letzten Projekt-Builds an. Zurzeit leuchtet der grüne Bär, was selbst für Branchenfremde als gutes Zeichen interpretierbar ist. Ein dankbarer Entwickler huscht vorbei und stellt vor dem grünen Bären ein kleines Schälchen mit Süßigkeiten ab ...

2.5 Zusammenfassung

In diesem Kapitel haben Sie in kompakter Form den Ablauf und die erforderlichen Rahmenbedingungen für effektive CI kennengelernt. Ein CI-Server koordiniert dabei als Dirigent des »Build-Orchesters« das Zusammenspiel der beteiligten Virtuosen wie Compiler, Build-Werkzeug, Versionskontrollsystem, Test-Framework usw. Im Vorher-Nachher-Vergleich konnten Sie bereits erahnen, welches Potenzial die Einführung von CI in der Softwareentwicklung freisetzen kann.

Vielleicht können Sie es inzwischen kaum erwarten, CI auch in Ihrer Arbeitsgruppe einzuführen. Was wird also benötigt? Im folgenden Kapitel werden wir die erforderlichen Rahmenbedingungen nochmals im Detail beleuchten.