2 Continuous Delivery: Was und wie?

2.1 Was ist Continuous Delivery?

Diese Frage ist nicht so einfach zu beantworten. Die Erfinder des Begriffs geben in [6] keine echte Definition. Martin Fowler [7] stellt in den Mittelpunkt, dass die Software jederzeit in Produktion ausgeliefert werden kann. Dazu ist eine Automatisierung der Prozesse zur Installation der Software notwendig und Feedback über die Qualität der Software. Wikipedia hingegen spricht von einer Optimierung und Automatisierung des Software-Release-Prozesses [8].

Letztendlich geht es bei Continuous Delivery darum, den Prozess bis zum Release der Software zu betrachten und zu optimieren. Genau dieser Prozess wird oft bei der Entwicklung ausgeblendet.

2.2 Warum Software-Releases so kompliziert sind

Software-Releases sind eine Herausforderung – jede IT-Abteilung hat wohl schon ein Wochenende in der Firma verbracht, um ein Release in Produktion zu bringen. Oft enden solche Aktionen damit, dass die Software irgendwie in Produktion gebracht wird – weil ab einem bestimmten Punkt in dem Prozess der Weg zurück zur alten Version noch gefährlicher und schwieriger ist, als der Weg nach vorne. An die Installation des Release schließt sich häufig eine lange Phase an, in der das Release stabilisiert werden muss.

Continuous Integration macht Hoffnung.

Heutzutage ist das Release in Produktion ein Problem. Vor nicht allzu langer Zeit setzten die Probleme schon wesentlich früher an: Die einzelnen Teams arbeiteten an ihren Modulen und vor dem Release mussten die verschiedenen Versionen zunächst integriert werden. Wenn die Module das erste Mal zusammen genutzt werden sollten, kompilierte das System oft noch nicht einmal. Tage oder gar Wochen später waren dann erst alle Änderungen integriert und kompilierten erfolgreich. Dann erst konnten die Deployments beginnen. Diese Prob-

leme sind heute meistens gelöst: Alle Teams arbeiten an einem gemeinsamen Versionsstand, der ständig automatisiert gemeinsam kompiliert und getestet wird. Dieses Vorgehen nennt man Continuous Integration. Die dafür notwendige Infrastruktur wird Kapitel 4 noch detailliert darstellen. Dass die Probleme dieser Phase gelöst sind, macht Hoffnung, dass die Probleme in den anderen Phasen hin zur Produktion ebenfalls lösbar sind.

Langsame und risikoreiche Prozesse

Die Prozesse in den späteren Phasen sind oft sehr komplex und aufwendig. Durch manuelle Prozesse sind sie außerdem langwierig und fehleranfällig. Das betrifft das Release in Produktion, aber auch die Phasen davor – also beispielsweise die verschiedenen Tests. Schließlich können gerade bei einem manuellen Prozess, der zudem nur ein paar Mal im Jahr ausgeführt wird, viele Fehler passieren – und das trägt zum Risiko bei.

Wegen des hohen Risikos und Aufwands werden Releases nicht besonders häufig in Produktion gebracht. Dadurch dauern die Prozesse noch länger, weil es keinen Übungseffekt gibt. Ebenso ist es schwierig, die Prozesse zu optimieren.

Schnell geht auch.

Auf der anderen Seite gibt es immer Möglichkeiten, im Notfall ein Release sehr schnell in Produktion zu bringen – wenn beispielsweise dringend ein Fehler behoben werden muss. Diese Prozesse umgehen aber jegliche Tests und damit die Sicherheitsnetze, die im normalen Prozess genutzt werden. Letztendlich ist das eine Risikomaximierung – die Tests werden ja aus einem Grund durchgeführt.

Also ist der normale Weg in die Produktion langsam und risikoreich – und im Notfall ist der Weg schnell, aber dafür noch risikoreicher.

2.3 Werte von Continuous Delivery

Mit der Motivation und den Ansätzen aus der Continuous Integration wollen wir den Weg für Releases in die Produktion optimieren.

Ein wesentliches Prinzip von Continuous Integration ist »If it hurts do it more often and bring the pain forward« – etwa: »Wenn es weh tut, tu es öfter und verlagere den Schmerz nach vorne.« Was sich wie Masochismus anhört, ist in Wirklichkeit ein Ansatz zur Problemlösung. Statt die Probleme bei den Releases zu umgehen, indem man möglichst wenig Releases in Produktion bringt, sollen die Prozesse so oft und so früh wie möglich durchgeführt werden, um sie möglichst schnell zu optimieren – in Bezug auf Geschwindigkeit und auf die Zuverlässigkeit. Continuous Delivery zwingt so die Organisation, sich zu ändern und eine andere Arbeitsweise zu adaptieren.

Überraschend ist der Ansatz eigentlich nicht: Wie schon erwähnt, kann jede IT-Organisation in kurzer Zeit einen Fix in Produktion bringen – und dabei werden meistens nur ein Teil der sonst üblichen Tests und Sicherheitsvorkehrungen ausgeführt. Das ist möglich, weil die Änderung nur klein ist und daher ein geringes Risiko hat. Hier zeigt sich ein anderer Ansatz für die Risikominimierung: Statt einer Absicherung über komplexe Prozesse und seltene Releases, kann man auch kleine Änderungen in Produktion bringen. Dieser Ansatz ist genau derselbe wie bei Continuous Integration (CI): Bei CI werden die Änderungen an der Software durch die einzelnen Entwickler und das Team ständig integriert und so häufig kleine Änderungen integriert, statt die Teams und Entwickler tage- oder wochenlang getrennt arbeiten zu lassen und die Änderungen am Ende zusammenzuführen – was meistens zu erheblichen Problemen führt, manchmal so sehr, dass die Software noch nicht einmal kompiliert werden kann.

Continuous Delivery ist aber mehr als »schnell und klein«. Continuous Delivery liegen verschiedene Werte zugrunde. Aus diesen Werten lassen sich die konkreten technischen Maßnahmen ableiten.

Regelmäßigkeit

Regelmäßigkeit bedeutet, Prozesse öfter durchzuführen. Alle Prozesse, die zur Auslieferung von Software notwendig sind, sollten regelmäßig durchgeführt werden – und nicht nur, wenn ein Release in Produktion gebracht werden muss. Beispielsweise ist es notwendig, Test- und Staging-Umgebungen aufzubauen. Die Testumgebungen können für fachliche oder technische Tests genutzt werden. Die Staging-Umgebung kann vom Endkunden genutzt werden, um die Features eines neuen Release auszuprobieren und zu evaluieren. Durch die Bereitstellung dieser Umgebungen kann der Prozess für den Aufbau einer Umgebung zu einem regelmäßigen Prozess werden, der nicht erst ausgeführt wird, wenn die Produktionsumgebung aufgebaut wird. Um diese Vielzahl von Umgebungen ohne allzu großen Aufwand zu erstellen, müssen die Prozesse weitgehend automatisiert werden. Regelmäßigkeit führt meistens zur Automatisierung. Ähnliches gilt für Tests: Es ist nicht sinnvoll, die notwendigen Tests bis kurz vor das Release zu verschieben – sie sollten lieber regelmäßig ausgeführt werden. Auch in diesem Fall erzwingt der Ansatz praktisch eine Automatisierung, um die Aufwände in Zaum zu halten. Regelmäßigkeit führt auch zu einer hohen Zuverlässigkeit – was ständig durchgeführt wird, kann zuverlässig wiederholt und durchgeführt werden.

Nachvollziehbarkeit

Alle Änderungen an der auszuliefernden Software und der dafür notwendigen Infrastruktur müssen nachvollziehbar sein. Es muss möglich sein, jeden Stand der Software und Infrastruktur zu rekonstruieren. Das führt zu einer Versionierung, die nicht nur die Software, sondern auch die notwendigen Umgebungen erfasst. Idealerweise ist es möglich, jeden Stand der Software zusammen mit der für den Betrieb notwendigen Umgebung in der richtigen Konfiguration zu erzeugen. Dadurch können alle Änderungen an der Software und an den Umgebungen nachvollzogen werden. Ebenso ist es einfach möglich, für die Fehleranalyse ein passendes System aufzubauen. Und schließlich können so Änderungen dokumentiert oder auditiert werden.

Eine mögliche Lösung des Problems ist es, dass Produktions- und Staging-Umgebung nur für bestimmte Personen zugänglich sind. Dadurch sollen Änderungen »kurz zwischendurch« vermieden werden, die nicht dokumentiert werden und nicht mehr nachvollziehbar sind. Außerdem sprechen Sicherheitsanforderungen und Datenschutz gegen den Zugriff auf Produktionsumgebungen.

Mit Continuous Delivery sind Eingriffe an einer Umgebung nur möglich, wenn ein Installationsskript geändert wird. Die Änderungen an den Skripten sind nachvollziehbar, wenn sie in einer Versionskontrolle liegen. Die Entwickler der Skripte haben auch keinen Zugriff auf die Produktionsdaten, so dass es mit Datenschutz ebenfalls keine Probleme gibt.

Regression

Um das Risiko bei der Auslieferung der Software zu minimieren, muss die Software getestet werden. Natürlich muss dabei die korrekte Funktion neuer Features sichergestellt werden. Aber viel Aufwand entsteht, um Regressionen zu vermeiden – also Fehler in eigentlich schon getesteten Softwareteilen, die durch Modifikationen eingeführt worden sind. Dazu müssen eigentlich alle Tests bei jeder Modifikation noch einmal ausgeführt werden – schließlich kann eine Modifikation an einer Stelle des Systems irgendwo anders einen Fehler erzeugen. Dazu sind automatisierte Tests notwendig, da sonst der Aufwand für die Ausführung der Tests viel zu hoch wird. Sollte ein Fehler es dennoch bis in die Produktion schaffe, so kann er immer noch durch Monitoring entdeckt werden. Idealerweise gibt es die Möglichkeit, auf dem Produktionssystem möglichst einfach eine ältere Version ohne den Fehler zu installieren (Rollback) oder einen Fix schnell in Produktion zu bringen (Roll forward). Eigentlich geht es also um eine Art

Frühwarnsystem, das über verschiedene Phasen wie Test und Produktion Maßnahmen ergreift, um Regressionen zu entdecken und zu lösen.

2.4 Vorteile von Continuous Delivery

Continuous Delivery bietet zahlreiche Vorteile. Je nach Szenario können die Vorteile unterschiedlich wichtig sein – und damit auch die Nutzung von Continuous Delivery beeinflussen.

2.4.1 Continuous Delivery für Time-to-Market

Continuous Delivery verringert die Zeit, die benötigt wird, um Änderungen in Produktion zu bringen. Dadurch ergibt sich auf der Geschäftsseite ein wesentlicher Vorteil: Es ist viel einfacher, auf Änderungen am Markt zu reagieren. Also verbessert Continuous Delivery das Time-to-Market.

Aber die Vorteile gehen weiter: Moderne Ansätze wie Lean Startup [1] propagieren einen Ansatz, der von der erhöhten Geschwindigkeit noch mehr profitiert. Im Wesentlichen geht es darum, am Markt Produkte zu positionieren und die Marktchancen auszuwerten und dabei möglichst wenig Aufwand zu investieren. Ganz wie bei wissenschaftlichen Experimenten wird definiert, wie der Erfolg des Produkts am Markt gemessen werden kann. Dann wird das Experiment durchgeführt und am Ende der Erfolg oder Misserfolg gemessen.

Ein Beispiel

Nehmen wir als konkretes Beispiel: In einem Webshop soll die Möglichkeit geschaffen werden, Bestellungen zu einem bestimmten Termin auszuliefern. Als erstes Experiment kann das Feature beworben werden. Als Indikator für den Erfolg dieses Experiments kann beispielsweise die Anzahl der Klicks auf einen Link in der Werbung genutzt werden. Zu diesem Zeitpunkt ist noch keine Software entwickelt – das Feature ist also noch nicht implementiert. Wenn das Experiment zu keinem erfolgversprechenden Ergebnis geführt hat, ist das Feature wohl nicht sinnvoll und es können andere Features priorisiert werden – ohne dass viel Aufwand investiert worden ist.

Feature implementieren und ausliefern

Wenn das Experiment erfolgreich war, wird das Feature implementiert und ausgeliefert. Auch dieser Schritt kann als Experiment durchgeführt werden: Metriken können helfen, um den Erfolg des Fea-

tures zu kontrollieren. Beispielsweise kann die Anzahl der Bestellungen mit einem festen Liefertermin gemessen werden.

Auf zum nächsten Feature!

Bei der Analyse der Metriken stellt sich heraus, dass die Anzahl der Bestellungen hoch genug ist – und interessanterweise die meisten Bestellungen nicht direkt zum Kunden, sondern zu einer dritten Person geschickt werden. Weitere Messungen ergeben, dass es sich offensichtlich um Geburtstagsgeschenke handelt. Basierend auf diesen Informationen kann nun das Feature weiter ausgebaut werden – beispielsweise mit einem Geburtstagskalender und Empfehlungen für passende Geschenke. Auch dazu müssen natürlich entsprechende Features geplant, implementiert, ausgeliefert und schließlich der Erfolg gemessen werden. Oder vielleicht gibt es auch Möglichkeiten, die Marktchancen dieser Features ganz ohne Implementierung zu evaluieren – durch Werbung, Kundeninterviews, Umfragen oder andere Ansätze.

Continuous Delivery führt zu Wettbewerbsvorteilen.

Continuous Delivery ermöglicht es, die notwendigen Änderungen an der Software schneller auszuliefern. Dadurch kann eine Firma schneller verschiedene Ideen ausprobieren und das Geschäftsmodell weiterentwickeln. Das führt zu einem Wettbewerbsvorteil: Da mehr Ideen ausgewertet werden können, ist es einfacher, die richtigen Ideen herauszufiltern – und zwar nicht aufgrund subjektiver Abschätzungen über die Markchancen, sondern anhand objektiv gemessener Daten.

Ohne Continuous Delivery

Ohne Continuous Delivery wäre das Feature für die festen Liefertermine geplant und beim nächsten Release ausgeliefert worden – das kann durchaus einige Monate dauern. Vorab hätte man wohl kaum gewagt, das Feature bereits zu bewerben – denn die lange Zeit, bis das Feature ausgeliefert wird, macht solche Werbung sinnlos. Wenn das Feature kein Erfolg geworden wäre, hätte man hohe Kosten bei der Implementierung gehabt, ohne dadurch einen Nutzen zu erzielen. Das Messen des Erfolgs wäre sicher auch in einem klassischen Modell möglich, aber die Reaktion würde deutlich länger dauern. Weiterentwicklungen wie die Unterstützung für Geburtstage wären noch später am Markt, denn dafür muss die Software noch einmal ausgeliefert werden und der langwierige Release-Prozess hätte noch ein zweites Mal durchlaufen werden müssen. Außerdem ist es fraglich, ob der Erfolg des Features ausreichend detailliert gemessen wird, um das Potenzial für die Features rund um Geburtstage zu erkennen.

Continuous Delivery und Lean Startup

Dank Continuous Delivery können also die Optimierungszyklen viel schneller durchlaufen werden, weil Features praktisch jederzeit in Produktion gebracht werden können. Das ermöglicht Ansätze wie Lean Startup. Das hat Auswirkungen auf die Geschäftsseite: Sie muss

schneller neue Features definieren und sie muss nicht mehr langfristig planen, sondern kann auf die Ergebnisse der aktuellen Experimente reagieren. Das ist in Start-ups besonders einfach, aber auch in klassischen Organisationen können solche Strukturen aufgebaut werden. Der Lean-Startup-Ansatz hat leider einen irreführenden Namen: Er beschreibt einen Ansatz, bei dem neue Produkte durch eine Serie von Experimenten am Markt positioniert werden – und dieser Ansatz ist natürlich auch in klassischen Unternehmen umsetzbar, nicht nur in Start-ups. Er kann auch genutzt werden, wenn Produkte klassisch ausgeliefert werden müssen – beispielsweise auf Datenträgern, mit anderen komplexen Installationsprozeduren oder als Teil eines anderen Produkts wie einer Maschine. Dann muss die Installation der Software vereinfacht oder idealerweise automatisiert werden. Außerdem muss ein Kundenkreis identifiziert werden, der gerne neue Versionen der Software ausprobieren will und dazu Feedback geben kann – also klassische Beta-Tester oder Power-User.

Auswirkungen auf den Entwicklungsprozess

Continuous Delivery hat Auswirkungen auf den Softwareentwicklungsprozess: Wenn einzelne Features in Produktion gebracht werden sollen, muss der Prozess das unterstützen. Einige Prozesse nutzen Iterationen von einem oder mehreren Wochen Länge. Am Ende jeder Iteration wird ein neues Release mit mehreren Features ausgeliefert. Das ist kein idealer Ansatz für Continuous Delivery, denn so können Features nicht alleine durch die Pipeline gebracht werden. Auch der Lean-Startup-Ansatz wird so erschwert: Wenn mehrere Features gleichzeitig ausgerollt werden, ist es nicht offensichtlich, welche Änderung die Messwerte beeinflusst. Nehmen wir an, dass die Auslieferung zu einem festen Liefertermin parallel mit einer Änderung der Versandpreise einher geht – welche der beiden Maßnahmen mehr Einfluss auf die höheren Verkaufszahlen hatte, ist nicht zweifelsfrei zu klären.

Also sind Prozesse wie Scrum, XP (Extreme Programming) und natürlich der Wasserfall hinderlich, denn die Prozesse liefern immer mehrere Features gemeinsam aus. Kanban [2] hingegen fokussiert darauf, ein einzelnes Feature durch die verschiedenen Phasen schließlich in Produktion zu bringen. Das passt ideal zu Continuous Delivery. Natürlich kann man die anderen Prozesse auch so modifizieren, dass sie die Auslieferung einzelner Features unterstützen – dann sind die Prozesse aber abgewandelt und zumindest nicht mehr nach Lehrbuch umgesetzt. Eine weitere Möglichkeit ist es, Features zunächst zu deaktivieren, um so zwar mehrere Features in einem Release gemeinsam auszuliefern, aber einzeln messbar zu machen.

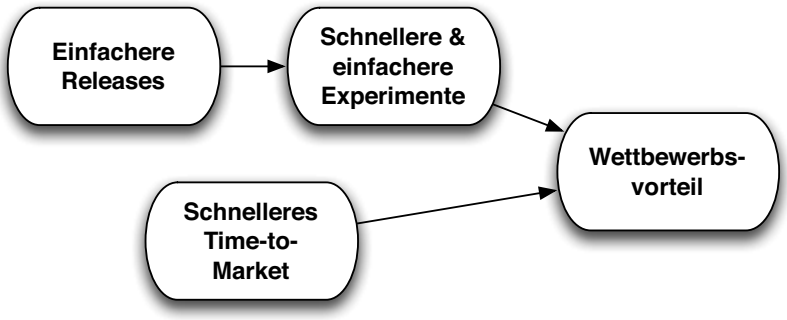


Abb. 2–1
Gründe für Continuous Delivery in einem Start-up-Umfeld

Schließlich bedeutet dieser Ansatz auch, dass Teams mehrere unterschiedliche Funktionen integrieren. Neben Entwicklung und Betrieb für die Features kämen noch Geschäftsrollen beispielsweise für Marketing in Frage. Durch die verringerten organisatorischen Hürden kann das Feedback aus dem Geschäftsbereich noch schneller in Experimente umgesetzt werden.

Selber ausprobieren und experimentieren

- Informiere dich über Lean Startup und Kanban. Woher kommt Kanban ursprünglich?
- Suche dir ein bekanntes Projekt oder Feature in einem Projekt aus:
- Was könnte ein minimales Produkt sein? Das minimale Produkt soll einen Hinweis auf die Marktchancen des eigentlich geplanten Produkts geben.
- Kann man das Produkt auch ohne Software evaluieren? Kann man es beispielsweise bewerben? Oder potenzielle Nutzer interviewen?
- Wie misst man den Erfolg des Features? Gibt es beispielsweise einen Umsatzeinfluss, eine Anzahl Klicks oder andere Werte, die man messen könnte?
- Welchen Vorlauf haben Marketing und Vertrieb typischerweise für die Planung eines Produkts oder Features? Wie passt das zum Lean-Startup-Gedanken?

2.4.2 Continuous Delivery zur Risikominimierung

Hinter der Nutzung von Continuous Delivery, wie sie im letzten Abschnitt dargestellt worden ist, steht ein Geschäftsmodell. Bei klassischen Unternehmen hängt das Geschäft aber oft von langfristiger Planung ab. Ein Ansatz wie Lean Startup lässt sich dann nicht umsetzen. Ebenso gibt es viele Unternehmen, bei denen Time-to-Market nicht entscheidend ist. Nicht alle Märkte sind diesbezüglich sehr kompetitiv. Das kann sich natürlich ändern, wenn solche Firmen plötzlich mit

Konkurrenten konfrontiert werden, die dazu in der Lage sind, mit einem Lean-Startup-Modell an den Markt zu gehen.

In vielen Szenarien fällt Time-to-Market als Motivation für die Einführung von Continuous Delivery aus. Dennoch können die Techniken sinnvoll sein. Denn Continuous Delivery bietet weitere Vorteile:

- Manuelle Release-Prozesse kosten viel Aufwand. Es ist keine Seltenheit, dass ganze IT-Abteilungen für jedes Release jeweils ein ganzes Wochenende blockiert werden. Und auch nach dem Release fallen meistens noch umfangreiche Nacharbeiten an.
- Außerdem ist das Risiko hoch: Das Ausrollen der Software hängt von vielen manuellen Änderungen ab. Dabei können Fehler geschehen. Werden die Fehler nicht rechtzeitig analysiert und behoben, kann das für das Unternehmen weitreichende Konsequenzen haben.

Die Leittragenden sind in den IT-Abteilungen zu finden: Entwickler und Systemadministratoren, die Wochenenden und Nächte arbeiten, um Releases in Produktion zu bringen und Fehler zu beheben. Dabei sind sie wegen der hohen Risiken oft auch hohem Stress ausgesetzt. Und die Risiken sind nicht zu unterschätzen: Knight Capital beispielsweise hat aufgrund eines fehlerhaften Software-Rollouts 440 Mio. $ verloren [4]. Die Folge war die Insolvenz der Firma. Aus solchen Szenarien lassen sich eine Menge Fragen ableiten [5].

Continuous Delivery kann eine Lösung für solche Situationen sein: Ein wesentlicher Aspekt von Continuous Delivery sind die höhere Zuverlässigkeit und die Qualität im Release-Prozess. Dadurch können Entwickler und Systemadministratoren im wahrsten Sinne des Wortes ruhig schlafen. Dafür sind verschiedene Faktoren relevant:

- Durch die höhere Automatisierung des Release-Prozesses werden die Ergebnisse besser reproduzierbar. Wenn also die Software in einer Test- oder Staging-Umgebung deployt und getestet worden ist, wird in der Produktion exakt dasselbe Ergebnis erzielt, weil die Umgebung vollständig identisch ist. So können Fehlerquellen weitgehend eliminiert werden und das Risiko sinkt.
- Es wird auch viel einfacher, Software zu testen, da die Tests weitgehend automatisiert sind. Dadurch wird die Qualität weiter gesteigert, da die Tests öfter durchlaufen werden können.
- Wenn öfter deployt wird, sinkt das Risiko ebenfalls. Denn pro Deployment werden weniger Änderungen in Produktion gebracht.

> Weniger Änderungen bedeuten aber auch weniger Risiko, dass sich irgendwo ein Fehler eingeschlichen hat.

In gewisser Weise ist die Situation paradox: In der klassischen IT wird versucht, Releases möglichst selten in Produktion zu bringen, da sie mit einem hohen Risiko verbunden sind. Bei jedem Release kann sich ein Fehler mit potenziell desaströsen Konsequenzen einschleichen. Weniger Releases sollten also weniger Probleme zur Folge haben.

Continuous Delivery setzt hingegen auf häufige Releases. So gehen bei einem Release weniger Änderungen live und die Wahrscheinlichkeit für das Auftreten von Fehlern sinkt. Voraussetzung sind automatisierte und zuverlässige Prozesse. Sonst führen die häufigen Releases zu einer Überlastung bei jenen, die manuelle Prozesse durchführen, und außerdem steigt das Risiko, da sich in manuelle Prozesse leicht Fehler einschleichen können. Statt einer niedrigen Release-Frequenz werden also Prozesse automatisiert, damit das Risiko eines Release sinkt. Dabei hilft natürlich, dass jedes Release wegen der höheren Release-Frequenz weniger Änderungen umfasst und so inhärent ein niedrigeres Risiko hat.

Die Motivation für Continuous Delivery unterscheidet sich also deutlich von der Lean-Startup-Idee: Es geht um die Zuverlässigkeit und bessere technische Qualität bei den Releases – nicht um Time-to-Market. Und die Nutznießer sind die IT-Abteilungen – nicht die Geschäftsbereiche.

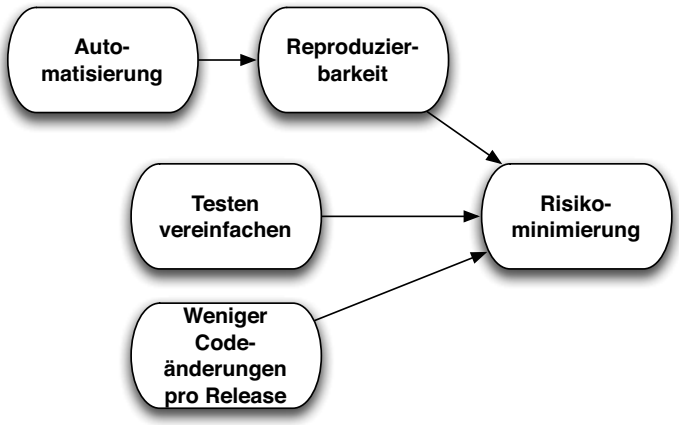


Abb. 2–2
Gründe für Continuous Delivery in einem Enterprise-Umfeld

Da die Vorteile andere sind, können auch andere Kompromisse eingegangen werden: Beispielsweise lohnt sich die Investition in eine Continuous-Delivery-Pipeline oft sogar, wenn sie nicht bis in Produktion geht – also die Produktion immer noch manuell aufgebaut werden muss. Schließlich wird für jedes Release die Produktion nur einmal

aufgebaut, aber es werden mehrere Umgebungen für die verschiedenen Tests benötigt. Ist Time-to-Market der Treiber für Continuous Delivery, wäre gerade die Produktion entscheidend.

Selber ausprobieren und experimentieren

Betrachte dein aktuelles Projekt:

- Wo tauchen typischerweise Probleme bei der Installation auf?
- Könnten die Probleme durch Automatisierung gelöst werden?
- Wo sollten die aktuellen Ansätze vereinfacht werden, um eine Automatisierung und Optimierung zu ermöglichen? Betrachte den Aufwand und den Nutzen.
- Wie werden Produktionssysteme und Testsysteme aktuell aufgebaut? Durch dasselbe Team? Wäre es denkbar, die Automatisierung auf beide Bereiche anzuwenden oder nur für einen?
- Für welche Systeme wäre eine Automatisierung sinnvoll? Wie oft werden die Systeme aufgebaut?

2.4.3 Schnelleres Feedback und Lean

Wenn ein Entwickler eine Änderung am Code vornimmt, bekommt er Feedback durch seine eigenen Tests, Integrationstests, Performance-Tests und schließlich aus der Produktion. Wenn nur einmal im Quartal Änderungen in Produktion gebracht werden, können zwischen der Änderung und dem Feedback aus der Produktion mehrere Monate vergehen. Ähnliches kann für Akzeptanztests oder Performance-Tests gelten. Tritt ein Fehler auf, muss der Entwickler sich überlegen, was er da überhaupt damals implementiert hatte und was das Problem sein könnte.

Bei Continuous Delivery verkürzen sich die Feedback-Zyklen: Jedes Mal, wenn Code die Pipeline durchläuft, bekommen der Entwickler und das ganze Team Feedback. Automatisierte Akzeptanztests und automatisierte Kapazitätstests können nach jeder Änderung ausgeführt werden. So können der Entwickler und das Team schnell Probleme erkennen und beseitigen. Die Geschwindigkeit des Feedbacks kann weiter erhöht werden, indem schnelle Tests wie Unit-Tests bevorzugt werden und zunächst in die Breite und dann erst in die Tiefe getestet wird. So wird zunächst sichergestellt, dass alle Features mindestens für einfache Fälle funktionieren – den sogenannten »Happy Path«. Grundlegende Fehler lassen sich so einfacher und schneller finden. Ebenso können Tests, die erfahrungsgemäß öfter mal fehlschlagen, am Anfang ausgeführt werden.

Ebenso entspricht Continuous Delivery dem Lean-Gedanken. Bei Lean wird alles, wofür der Kunde nicht zahlen würde, als Waste angesehen. Eine Änderung am Code ist Waste, bis sie in Produktion gebracht worden ist, denn erst dann wird der Kunde für die Änderungen zahlen wollen. Außerdem setzt Continuous Delivery auf kürzere Durchlaufzeiten für schnelles Feedback – ein weiteres Lean-Konzept.

Selber ausprobieren und experimentieren

Betrachte dein aktuelles Projekt:

- Wie viel Zeit vergeht zwischen einer Code-Änderung und
  - Feedback von einem Continuous Integration Server?
  - Feedback aus einem Akzeptanztest?
  - Feedback von einem Performance-/Kapazitätstest?
  - der Auslieferung in Produktion?

2.5 Aufbau und Struktur einer Continuous-Delivery-Pipeline

Wie schon erwähnt, erweitert Continuous Delivery das Vorgehen von Continuous Integration auf die weiteren Phasen. Einen Überblick über die Phasen bietet Abbildung 2–3.

Abb. 2–3
Phasen der Continuous-Delivery-Pipeline

Dieser Abschnitt führt die Struktur einer Continuous-Delivery-Umgebung ein. Sie orientiert sich an Humble et al. [6] und besteht aus den folgenden Phasen:

- Die Commit-Phase beschreibt jene Aktivitäten, die typischerweise von einer Continuous-Integration-Infrastruktur abgedeckt werden. Dazu zählen der Build-Prozess, Unit-Tests und statische Code-Analyse. Diesen Teil der Pipeline beschreibt Kapitel 4 detailliert.
- Der nächste Schritt sind Akzeptanztests, die in Kapitel 5 im Mittelpunkt stehen. Genau genommen geht es dabei um automatisierte Tests: Entweder werden die Interaktionen mit der GUI automatisiert, um so das System zu testen, oder die Anforderungen werden in natürlicher Sprache so hinterlegt, dass sie als automatisierte Tests genutzt werden können. Spätestens ab dieser Phase ist es notwendig, dass Umgebungen aufgebaut werden, auf denen die

Anwendungen laufen können. Daher widmet sich das Kapitel 3 der Frage, wie solche Umgebungen aufgebaut werden können.

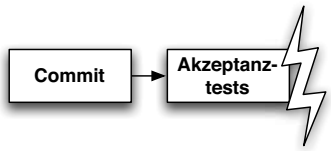
- Bei den Kapazitätstests (Kap. 6) wird sichergestellt, dass die Software die erwartete Last abarbeiten kann. Dazu sollte ein automatisierter Test dienen, der eindeutig ergibt, ob die Software ausreichend leistungsfähig ist oder nicht. Es geht dabei nicht nur um die Performance, sondern auch um die Skalierbarkeit. Der Test kann also auch auf einer Umgebung stattfinden, die nicht der Produktionsumgebung entspricht – sie muss allerdings zuverlässige Aussagen über das Verhalten in Produktion liefern. Je nach konkretem Einsatzkontext können auch andere nichtfunktionale Anforderungen automatisiert getestet werden, wie beispielsweise Sicherheit.
- Beim explorativen Test (Kap. 7) wird die Anwendung nicht nach einem strikten Testplan auf den Prüfstand gestellt, sondern Domänenexperten testen die Anwendung mit einem Fokus auf neue Features und unvorhergesehenes Verhalten. Es müssen also auch bei Continuous Delivery nicht alle Tests automatisiert werden. Vielmehr wird durch die automatisierten Tests genügend Luft für das explorative Testen geschaffen, da Routinetests nicht mehr manuell abgearbeitet werden müssen.
- Die Einführung in die Produktion (Kap. 8) umfasst nur noch die Installation der Anwendung in einer weiteren Umgebung und ist daher recht risikoarm. Es gibt verschiedene Ansätze, um die Risiken bei der Einführung in die Produktion weiter zu minimieren.
- Auch während des Betriebs der Anwendung ergeben sich weitere Herausforderungen – vor allem im Bereich Monitoring und Überwachung von Log-Dateien. Diesen Herausforderungen widmet sich Kapitel 9.

Grundsätzlich werden Releases in die einzelnen Phasen promotet: Sie durchlaufen die einzelnen Phasen nacheinander. Es ist denkbar, dass ein Release zwar noch in die Akzeptanztestphase kommt und die Tests dort erfolgreich besteht. Bei den Kapazitätstests stellt sich aber heraus, dass die Software den Anforderungen bezüglich des Lastverhaltens nicht gerecht wird. Dann wird das Release nie in die weiteren Phasen wie den explorativen Test oder gar die Produktion promotet. So werden zunehmend mehr Anforderungen an die Software erfüllt, bis sie schließlich in Produktion geht.

Nehmen wie beispielsweise an, dass die Software einen fachlichen Fehler enthält. Der würde spätestens beim Akzeptanztest auffallen, denn dort wird die fachliche Funktionalität der Anwendung getestet.

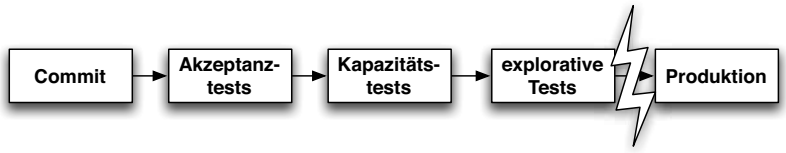
Also würde die Pipeline abgebrochen – siehe Abbildung 2–4. Weitere Tests sind dann nicht mehr sinnvoll.

Abb. 2–4
Pipeline bricht beim Akzeptanztest ab.



Die Entwickler beheben diesen Fehler und die Software wird erneut gebaut. Dieses Mal übersteht sie auch den Akzeptanztest. Aber in einer neuen Funktionalität, für die es keinen automatisierten Akzeptanztest gibt, ist ein Fehler. Dieser Fehler kann nur bei den explorativen Tests auffallen. Also bricht die Pipeline diesmal bei den explorativen Tests ab und auch diese Software geht nicht in Produktion – siehe Abbildung 2–5. So wird aber auf jeden Fall nicht die Zeit der Tester mit Software verschwendet, die Fehler enthält, die schon automatisierte Tests finden können, oder die den Kapazitätsanforderungen nicht gerecht wird.

Abb. 2–5
Pipeline bricht bei den manuellen Tests ab.



Prinzipiell können mehrere Releases parallel in der Pipeline bearbeitet werden. Dazu ist es natürlich notwendig, dass die Pipeline mehrere Releases parallel zulässt – wenn die Tests jeweils in festen Umgebungen laufen, ist das nicht möglich, da die Umgebung dann von einem Test belegt wird und ein paralleler Test eines zweiten Release nicht möglich ist.

Allerdings werden bei Continuous Delivery kaum Releases parallel bearbeitet. Ein Projekt sollte genau einen Stand in der Versionsverwaltung haben, der jeweils durch die Pipeline promotet wird. Es kann höchstens passieren, dass so schnell Änderungen an der Software vorgenommen werden, dass ein neues Release schon in die Pipeline geschickt wird, bevor das vorherige Release die Pipeline verlassen hat. Vielleicht gibt es noch Ausnahmen für Hotfixes – aber ein Ziel von Continuous Delivery ist gerade, alle Releases gleich zu behandeln.

Das Beispiel

Im Buch wird eine Beispielanwendung genutzt. Es handelt sich dabei um eine Kundenregistrierung – inspiriert durch das Beispiel der Raffzahn Online Commerce GmbH (siehe Abschnitt 1.2). Fachlich ist dieses Beispiel bewusst sehr einfach gehalten. Im Wesentlichen werden von einem Kunden Vorname, Name und E-Mail-Adresse erfasst. Die Registrierungen werden validiert: Die E-Mail-Adresse muss syntaktisch korrekt sein und für eine E-Mail-Adresse ist nur eine Registrierung erlaubt. Außerdem lässt sich eine Registrierung anhand der E-Mail-Adresse suchen und kann schließlich gelöscht werden.

Da die Anwendung nicht sonderlich komplex ist, ist sie auch recht leicht zu verstehen, so dass sich der Leser auf die verschiedenen Aspekte von Continuous Delivery konzentrieren kann, die mit der Anwendung verdeutlicht werden.

Technisch ist die Anwendung mit Java und dem Framework Spring Boot [5] implementiert. Dadurch ist es möglich, die Anwendung einschließlich Weboberfläche ohne Installation eines Web- oder Application-Servers zu starten. Das macht gerade das Testen einfacher, da keine Infrastruktur installiert werden muss. Die Anwendung kann aber auch in einem Application- oder Webserver wie Apache Tomcat betrieben werden, wenn das notwendig sein soll. Die Daten werden in HSQLDB abgelegt. Das ist eine In-Memory-Datenbank, die innerhalb des Java-Prozesses läuft. Auch diese Maßnahme reduziert die technische Komplexität der Anwendung.

Der Quellcode des Beispiels kann unter *http://github.com/ewolff/user-registration-V2* heruntergeladen werden. Ein wichtiger Hinweis: Der Beispielcode enthält Dienste, die unter root-Rechten laufen und auf die durch das Netz zugegriffen werden kann. Das ist für Produktionsumgebungen wegen der sich daraus ergebenden Sicherheitsprobleme nicht tragbar. Allerdings soll der Beispielcode auch nur zum Experimentieren dienen. Dafür ist der einfache Aufbau der Beispiele nützlich.

2.6 Links & Literatur

[1] **Eric Ries:** The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Crown Business, 2011, ISBN 978-0-67092-160-7

[2] **David J. Anderson:** Kanban: Evolutionäres Change Management für IT-Organisationen, dpunkt.verlag, 2011, ISBN 978-3-89864-730-4

[3] *http://de.wikipedia.org/wiki/5-Why-Methode*

[4] *http://www.sec.gov/litigation/admin/2013/34-70694.pdf*

[5] *http://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/*

[6] **Jez Humble, David Farley:** Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9

[7] *http://martinfowler.com/bliki/ContinuousDelivery.html*

[8] *http://en.wikipedia.org/wiki/Continuous_delivery*