DevOps

from pain delivery to continuous delivery

Jackson Santos

Status: **Entwurf**

Revision 2.0

Datum	Verantwortlicher
11.08.2011	NN

Inhaltsverzeichnis

Einleitung Continuous Delivery 1

Zielsetzung 1

Ziele 1

Problematik 1

Lösung: Öfters releasen 1

Wesentliche Vorteile von Continuous Delivery 1

Software vom Fließband 2

Das Continuous Testing Model 3

Kurze Erläuterung der Phasen des CT-Modells 3

Wesentliche Vorteile von Continuous Testing 4

Was steht Continuous Testing im Weg? (Subjektive Betrachtung für den Fall Globalpark) 5

Risiken bei der Einführung von Continuous Testing 6

Fazit zu Continuous Testing 6

Jenseits von Continuous Testing 7

Deployment Pipeline 8

Management von Artefakten 9

Die Phasen der Deployment Pipeline 9

Self-Service Deployment 11

Was kann im Auslieferungsprozess noch optimiert werden? 14

Einleitung Continuous Delivery

Zielsetzung

Continuous Delivery geht vom einfachen Prinzip aus, dass die Software jeder Zeit einsatzbereit ist. Mit Hilfe von agilen Ansätzen wie Continuous Testing und Continuous Deployment wird versucht, den Releaseprozess so zu gestalten, dass die Software per Knopfdruck ausgeliefert wird. Hierdurch soll nicht nur sichergestellt werden, dass das mit dem Release verbundene Risiko minimiert wird, sondern auch, dass das Endprodukt inkrementell an die Bedürfnisse des Kunden oder Stakeholder angepasst werden kann.

In diesem White Paper geht es grundsätzlich darum, zu zeigen, wie die Automatisierung des Build-, Deploy-, Test- und Releaseprozesses mit Hilfe einer Deployment Pipeline erfolgen kann und welche Vorteile bzw. Herausforderungen damit verbunden sind.

Ziele

1. Kontinuierliche Verbesserung der Softwarequalität
2. Optimierung des Build-, Deployment-, Test- und Releaseprozesses
3. Aufbau einer „production-ready software“ Mentalität

Problematik

Da die Releasephase fast immer als „schmerzhaft“ empfunden wird, tendieren wir dann, die Software so selten wie möglich oder ganz am Ende einer aufwändigen Entwicklungsphase zu releasen. Das Problem bei diesem Ansatz ist, dass, je seltener die Software released wird, desto größer ist die Wahrscheinlichkeit, dass „Überraschungen“ in der Testphase bzw. Produktivphase auftreten werden. Im schlimmsten Fall kann es beim Auftreten unbekannter Probleme bereits zu spät sein, sodass wertvolle Features fehlerhaft oder nicht in vollem Umfang ausgeliefert werden.

Lösung: Öfters releasen

Um diese „Überraschungen“ zu vermeiden und Risiken zu minimieren, soll die Software öfters released werden. Mit „releasen“ ist nicht unbedingt die tatsächliche Auslieferung an den Kunden gemeint, sondern auch die Auslieferung eines fertigen Produkts an die QA oder an beteiligte Stakeholders.

Hierbei soll das Testen keine Phase sein, sondern vielmehr ein Teil des Entwicklungsprozesses. Nur wenn wir auftretende Probleme kontinuierlich beseitigen, und zwar vom Anfang an, wird unser Produkt am Ende der Entwicklungsphase robust genug sein, so dass es frühstmöglich an den Kunden gebracht werden kann.

Wesentliche Vorteile von Continuous Delivery

Robustheit: Die Software ist immer releasebar (dies ist zugleich die große Herausforderung von Continuous Delivery).
Strategischer Vorteil: Die Entscheidung „wann releasen“ ist nicht (immer) von Test- oder Releasephase abhängig.
Erfolgsmessung: Der reale Projekterfolg kann nur gemessen werden, wenn die Software released und echtes Feedback eingeholt wurde.

Software vom Fließband

Die Einführung von Continuous Integration ist sicherlich für die Steigerung der Qualität und der Produktivität bei vielen Softwareprojekten verantwortlich. CI gibt uns die Sicherheit, dass sich der neue Code, den wir geschrieben haben, mit anderen Modulen und Klassen einwandfrei integrieren lässt. Sollte unsere letzte Code-Änderung einen gravierenden Fehler verursacht haben, werden wir sofort informiert und können somit den Fehler schnellstmöglich beheben.

Obwohl die Einführung von CI durch Systeme wie Jenkins oder CruiseControl sehr leicht gemacht wird, ist bei vielen Unternehmen jedoch keine Umgebung vorhanden, die eine kontinuierliche aber auch konsequente Softwareintegration unterstützt. CI-Systeme werden in der Praxis häufig als Buildmanagementsysteme eingesetzt. Wobei hiermit nur die Code-Kompilierung und das Linken des kompilierten Codes an Bibliotheken gemeint ist. Wichtige Aspekte wie der Einsatz von dedizierten Testrechnern sowie eine kontinuierliche Testentwicklung werden häufig vernachlässigt.

Die fehlende Planung einer Teststrategie kann beim Einsatz von CI in der Praxis folgende Probleme mit sich bringen:

- Tester und Support warten zu lange auf „green builds“, da die Ausführung vorhandener Tests zu lange dauert.
- Dem Development-Team werden weiterhin Bugs gemeldet, obwohl das Projekt seit Monaten abgeschlossen wurde.
- Erst am Ende der Entwicklungsphase wird erkannt, dass die System-Architektur die nicht-funktionalen Anforderungen (Stichwort: Performanz) nicht erfüllen kann, da eine kontinuierliche Entwicklung der Tests nicht stattfindet.

All das kann zu einer Software führen, die trotz kontinuierliche Integration nicht robuster bzw. nicht performanter wird. Nicht vorhandene Zusicherungen im Laufe der Entwicklungszyklen müssen am Ende der Entwicklungsphase durch aufwändige und kostenintensive manuelle Tests kompensiert werden (zum Teil auch durch manuelle Regressionstests).

Ein möglicher Ansatz für die Optimierung des Integrationsprozesses ist die (fast) vollständige Automatisierung des Testprozesses. Dieser Ansatz wird mit dem Begriff „Continuous Testing“ beschrieben (vgl. Abbildung 1). Es handelt sich um einen ganzheitlichen Testmanagement-Ansatz, wobei der Fokus bei der Feedback-Generierung liegt.

Das Continuous Testing Model

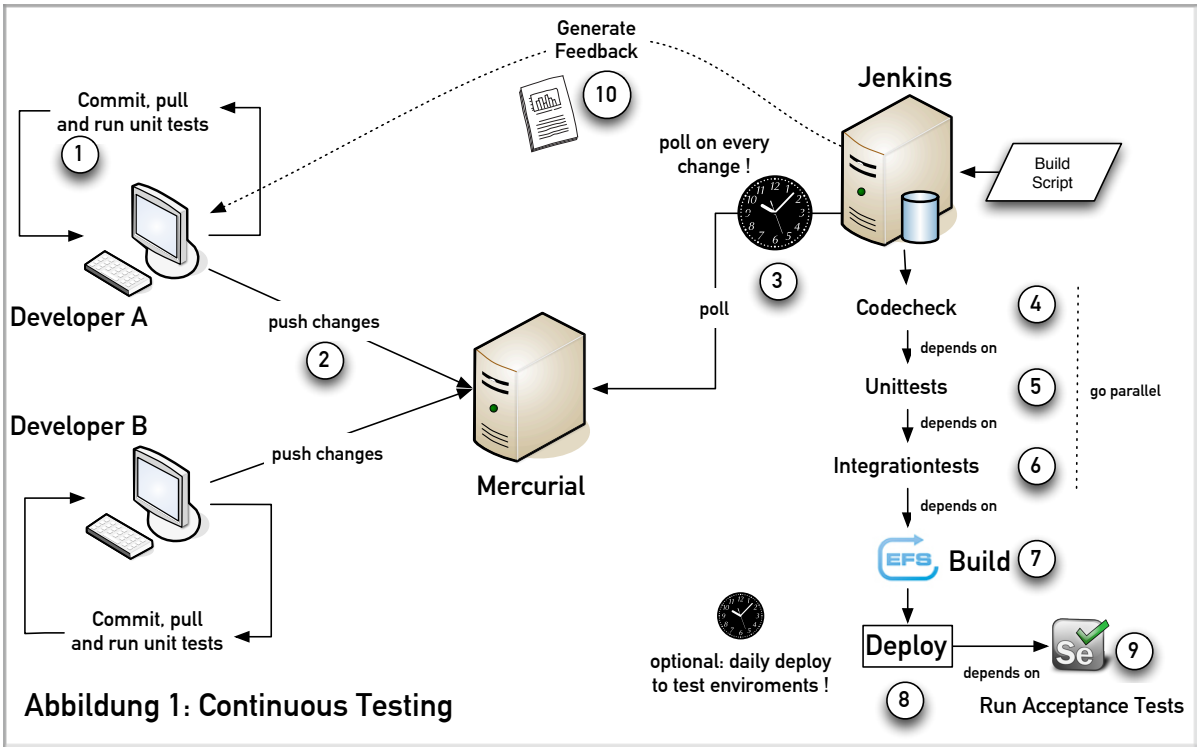

Abbildung 1: Continuous Testing

Kurze Erläuterung der Phasen des CT-Modells

1 - Änderungen im lokalen Repository werden committed und die letzte Änderungen vom Remote-Repository gepullt - und damit auch die letzten Unittests -. Die Unittests sollen erst lokal ausgeführt werden. Durch Parallelisierung bei der Testausführung und Einsatz von dedizierten Testrechnern, soll dieser Vorgang nur wenige Sekunde dauern.

2 - Nach erfolgreicher Testausführung werden die Änderungen zum Remote-Repository gepusht.

3 - Der CI-Server überprüft alle x Sekunden, ob neue Revisionen eingecheckt wurden. Wenn das der Fall ist, wird das System „neu gebaut". Dafür werden die Schritte 4 und 5 auf einem dedizierten Testrechner parallel ausgeführt. Diese zwei Schritte bilden die sog. „*Commit Stage*" des Build-Prozesses und sollen als erstes ausgeführt werden. Durch Parallelisierung soll dieser Vorgang nur wenige Minuten dauern (in der Praxis liegt die maximale Grenze bei etwa 5 bis 10 Minute).

4 - Der statische Codecheck wird ausgeführt (*Commit Stage)*.

5 - Unitests werden ausgeführt (*Commit Stage)*.

6 - Integrationtests werden ausgeführt (optional in *Commit Stage)*.

7 - Ein „grüner Build" wird erstellt und somit auch ein potentielles Release. Ab diesem Moment, müssen QA und Support in der Lage sein, den Build in die gewünschte Zielumgebung zu deployen und zwar unabhängig davon wie die Ergebnisse der automatisierten und manuellen Akzeptanz-Tests aussehen werden (warum? siehe in diesem Abschnitt).

8 - Der erzeugte Build wird **automatisch (!)** in die Akzeptanz-Zielumgebung deployed.

9 - Akzeptanz-Tests (automatisierte GUI-, **Performance(!)**- und Smoketests) werden ausgeführt. Diese Phase wird „*Acceptance Stage*" genannt und dauert in der Regel länger als die „*Commit Stage*".

10 - Ein Bericht mit den Ergebnissen des Build- und Testprozesses wird erstellt und kann vom Entwickler oder Projektverantwortlichen als wichtiger Feedback-Mechanismus verwendet werden.

Abbildung 2: Extreme Feedback Device

Sobald der Buildprozess wegen eines schwerwiegenden Fehlers unterbrochen wird, bekommen die Entwickler, die für die letzte Revision zuständig sind, eine entsprechende Warnmail. Im Idealfall begleitet der Qualitätsverantwortlicher den gesamten Prozess und verschickt persönliche Erinnerungsemails, da diese meistens schwer zu ignorieren sind. Zusätzlich dazu, können sog. Extreme Feedback Devices in Form von physischen Objekten eingesetzt werden. Diese leuchten, blinken oder geben verschiedene Geräusche aus je nachdem wie der Build-Status geworden ist (vgl. Abbildung 2).

Wichtig bei diesem Ansatz ist, dass der Build-Prozess vollkommen transparent gemacht wird. Jeder, der im Prozess beteiligt ist - im normalen Fall das ganze Development-Team -, soll in der Lage sein, Fehler zu beheben und zwar unabhängig davon, ob der Fehler mit dem Build-Skript zusammenhängt oder von der letzen Code-Änderung verursacht wurde. „Keep builds in the green“ soll in diesem Zusammenhang das oberste Gebot sein.

Die technische Umsetzung des Continuous-Testing-Models, sowie die Begriffe „Commit Stage“ und „Acceptance Stage“ werden im Unterkapitel „Deployment Pipeline“ näher erläutert.

Wesentliche Vorteile von Continuous Testing

- **Risikominimierung**
  - i. Fehler können früher erkannt und behoben werden.
  - ii. Der „Gesundheitszustand“ der Software ist messbar und für alle sichtbar.
  - iii. Risiken bei getroffenen Annahmen („ich denke, dass es jetzt gefixt ist“) werden minimiert.
- **Minimierung repetitiver Prozesse durch Prozessautomatisierung**
  - i. Durch die Minimierung wiederholender Prozesse werden Zeit, Kosten und Arbeitsaufwand gespart.
  - ii. Entwickler haben mehr Freiheit für kreative Aufgaben, da wenig manuell gemacht werden muss.
  - iii. Der Prozess ist stark standardisiert und dadurch weniger fehleranfällig.
- **Production-ready Software**
  - i. Hat die Software alle Phasen des Build- und Testprozesses überstanden, so ist die Wahrscheinlichkeit, dass Fehler in der Produktivumgebung auftreten, sehr gering. Hierdurch wird gewährleistet, dass die Software früher an den Kunden gebracht und echtes Feedback eingeholt werden kann.
- **Prozesstransparenz**
  - i. Vielen Projektverantwortlichen stehen keine Informationen über den aktuellen Stand der Software zur Verfügung. Das kann dazu führen, dass die Entscheidungsfindung, um z.B. eine Optimierungsaufgabe durchzuführen, extrem erschwert wird. Continuous Testing kann Informationen über Build-Status und Qualitätsmetriken „just-in-time“ bereitstellen, sodass jeder den aktuellen „Gesundheitszustands“ der Software kennt und bei der Entscheidungsfindung unterstützen kann.

Was steht Continuous Testing im Weg? (Subjektive Betrachtung für den Fall Globalpark)

- **Geringe Testabdeckung:** Das Thema ist im Fall von Globalpark ein generelles Problem. Fehler können nur gefunden werden, wenn auch Tests dafür existieren. Die Folge einer geringen Testabdeckung ist, wie im vorigen Absatz erwähnt, dass gleiche Tests mehrmals wiederholt werden müssen. Nebeneffekte wie z.B. Folgefehler werden dadurch nicht (immer) ausgeschlossen. Mögliche Maßnahmen für die Optimierung der Testabdeckung werden im Folgenden aufgelistet:

  i. Wichtige Codebereiche identifizieren und Integrationstests nachziehen.
  ii. Testaufgaben können über Kanban umgesetzt werden.
  iii. Obligatorische Unit- und **Performancetests** für neue Features (Definition of Done).
  iv. Durchdachte Branching-Strategie (TOFU Skala - siehe Vortrag „Branching Concepts“ von Jackson).

- **Kulturelles Problem:** Es existiert noch keine *beständige* Kultur. Committen bedeutet längst nicht fertig sein. Erst wenn eine robuste Software in die Produktivumgebung gebracht wurde, ist das Development-Team endlich fertig. Solange die Software nicht ausgeliefert wurde, ist die reale Leistung des Entwicklers nicht erbracht worden. Mögliche Maßnahmen für dieses Problem werden im folgenden aufgelistet:

  i. Aufbau einer „production-ready software“ Mentalität. Jedes Check-In kann ein **potentielles Release** werden.
  ii. Kein Commit von fehlerhaften Features (Unittests erst lokal ausführen)
  iii. Fehlerhafter Build soll schnellstmöglich behoben werden (jeder ist hierbei Owner des Buildprozesses)

- **Wartung des CI-Systems:** Die aufwändige Wartung eines CI-Systems kann einer der Hauptgründe sein, warum Continuous Testing nicht vollständig eingeführt wird. Allerdings wird in der Praxis schnell erkannt, dass die Wartung und Kontrolle eines robusten CI-Systems bequemer und kostengünstiger ist als die Überwachung manueller Prozesse. Je komplexer eine Software wird, desto sinnvoller sollte der Einsatz von Continuous Testing sein. Bei komplexen Softwareprojekten wird jedoch häufig auf Continuous Testing verzichtet und zwar mit der Begründung „Die Wartung des CI-Systems sei zu Aufwändig“.

- **Zu viele Änderungen im Entwicklungsprozess:** Ein weiterer Grund, warum auf Continuous Testing immer wieder verzichtet wird, sind die Prozessänderungen an sich. Es wird meistens argumentiert, dass die Einführung eines solchen Models zu viele Änderungen mit sich bringt und aufgrund des hohen Aufwands nicht gewährleistet werden kann.

  Allerdings kann die Einführung von Continuous Testing nur erfolgreich ablaufen, wenn sie in ***kleinen Schritten*** stattfindet. Die Optimierung der Form, in der wir Software für die Produktivumgebung bereitstellen, sollte daher nicht als „Big-Bang-Ansatz“[1] durchgeführt werden. Identifizierung und Realisierung anhand von Quickwins sind hier der Weg zum Erfolg.

- **Einchecken von instabilen Änderungen:** Eine der größten Schwierigkeit bei der Einführung von Continuous Testing sind die ständige „rote Builds“. Der Grund dafür sind häufig die nicht stattfindende lokale Testausführung sowie die nicht stattfindende Trennung von Stable- und Integration-Lines. Mögliche Maßnahmen für diese Problematik werden im folgenden aufgelistet:

  i. Unittests nachziehen und erst lokal ausführen (größte Herausforderung von Continuous Testing)
  ii. Transparenz schaffen (welche und wie viele Unittests haben wir, wo kann ich sie finden?)
  iii. Kosten eines in der Produktivumgebung auftretenden Fehlers sichtbar machen (was kostet Globalpark ein fehlender Unittest nachdem die Software bereits ausgeliefert wurde?)

[1] Stichtag: bzgl. Software- oder Prozesseinführung

- **Anschaffungskosten (Hardware, Software) und Einarbeitung:** Selbst wenn bei vielen Unternehmen behauptet wird, dass Continuous Testing stattfindet, ist dies in der Regel nicht der Fall. Die mit der Einführung eines CI-Systems verbundene Anschaffungskosten werden von Qualitätsverantwortlichen als zu hoch eingeschätzt, sodass Continuous Testing nur zum Teil vorhanden ist. Die Anschaffungskosten für neue CI- und dedizierte Testrechner sowie die für die Einarbeitung in das Thema Continuous Testing sollen in der Tat nicht unterschätzt werden. Mittelfristig allerdings, kann diese Investition durch einen positiven „return on investment“ begründet werden.

Risiken bei der Einführung von Continuous Testing

i. Unvertretbarer Aufwand für die Wartung vorhandener Tests.

ii. Versehentlicher Aufbau einer „passiven Mentalität“ was Qualität angeht („es wird eher getestet“).

iii. Zu starker Fokus auf Automatisierung und Vernachlässigung der Akzeptanzkriterien.

Fazit zu Continuous Testing

Die Einführung von Continuous Testing und den damit verbundene Aufwand dürfen wir sicherlich nicht unterschätzen. Die Wirtschaftlichkeit jedoch, die durch den Einsatz eines solchen Models erzielt wird, darf nicht vernachlässig werden. Mit Hilfe einer durchdachten Einführungsstrategie soll jedes Development-Team in der Lage sein, den Einführungsprozess in kleinen Schritten zu verwirklichen.

Jenseits von Continuous Testing

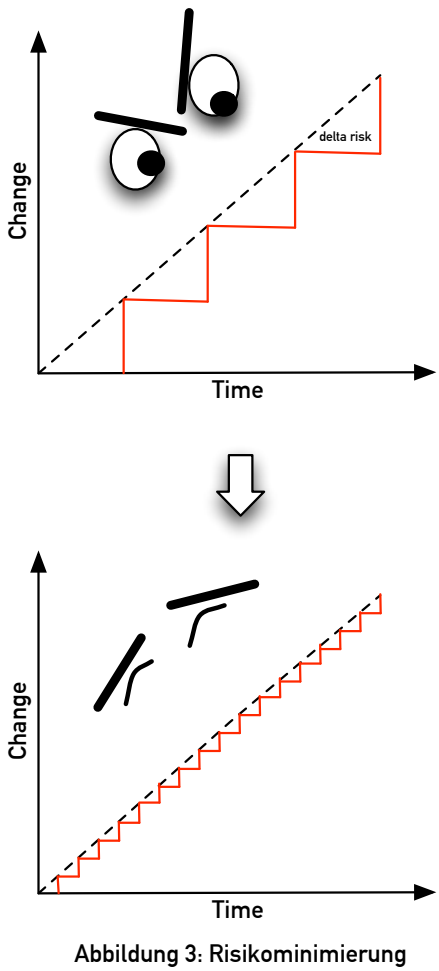

Abbildung 3: Risikominimierung

Das Ziel von Continuous Delivery ist, die Möglichkeit zu schaffen, Software per Knopfdruck bereitzustellen. Aber wie können wir unseren Releaseprozess so anpassen, dass die Software jederzeit einsatzbereit ist? Welche sind die technische Anforderungen, die bei der Einführung von Continuous Delivery zu beachten sind?

Durch den Einsatz von Continuous Testing wird die Basis für das geschaffen, was in der Praxis „Continuous Deployment“ genannt wird. Continuous Deployment beschreibt die kontinuierliche Auslieferung von Software an die QA, an Kunden oder an Stakeholder. Der wesentliche Vorteil von Continuous Deployment ist die Risikominimierung, die durch regelmäßiges Releasen erzielt wird (vgl. Abbildung 3).

Durch regelmäßiges Releasen kann das Development-Team auftretende Probleme besser nachvollziehen (die Themen sind ja aktuell) und Fehler schneller beheben.

Aber nicht nur das Risiko wird durch häufiges Releasen minimiert, sondern auch der Feedback-Zyklus zwischen Development, QA und Stakeholder wird dadurch beschleunigt. Mit Hilfe von Ansätzen aus der Lean-Startup-Bewegung wie z.B. „Minimum Viable Product“ wird das Endprodukt nicht mit allen möglichen Features (Business Value) an den Kunden (bzw. QA oder Stakeholder) ausgeliefert, sondern nur mit den Features, die im letzten Entwicklungszyklus **vollständig**[2] umgesetzt wurden. Durch das Einholen von echtem Feedback wird nun sichergestellt, ob die umgesetzten Features den Anforderungen entsprechen.

Damit wir regelmäßig releasen können, müssen wir zunächst sicherstellen, dass die Software deployable und somit jeder Zeit releasebar ist. Um dies zu erreichen, soll jeder einzelne Build als ein potentieller Release-Kandidat behandelt werden. Dadurch, dass der Build und somit ein neuer Release-Kandidat bei jedem neuen Commit „entsteht“, muss jede Änderung in der Software erst durch einen komplexen Delivery-Prozess validiert und freigegeben, werden bevor der Commit in die Zielumgebung ausgerollt wird.

Der größter Unterschied zwischen dem Delivery- und unserem herkömmlichen Entwicklungsprozess besteht darin, dass die Freigabe zwischen den einzelnen Phasen des Prozesses (Commit, Build, Deploy, Test und sogar Release) zum großen Teil automatisiert erfolgt.

Die technische Umsetzung von Continuous Testing[3] und somit auch von Continuous Deployment[3] wird anhand einer Deployment Pipeline beschrieben (vgl. nächstes Kapitel).

---

[2] Unter vollständig versteht sich: Getestet und fehlerfrei. Performance- und Loadtests gehören ebenfalls dazu und sollten nicht erst am Ende des Projektes ausgeführt werden.

[3] Continuous Testing + Continuous Deployment = Continuous Delivery => Delivery Process

Deployment Pipeline

Die Deployment Pipeline[4] beschreibt (auf einem hohen Abstraktionsniveau) die Automatisierung des Build-, Deployment-, Test- und Releaseprozesses. D.h. die Automatisierung (fast) aller Prozesse, die notwendig sind, um die Software vom Repository bis zum Kunden, Stakeholder oder QA zu bringen.

Der Input einer Deployment Pipeline ist eine bestimmte Revision im Versionsverwaltungssystem. Für jede Änderung in der Software wird ein neuer Build erstellt, der mehr oder weniger wie eine Heldenfigur alle Testphasen und Herausforderungen des Prozesses überstehen muss, um seine Robustheit zu beweisen. Am Ende des Prozesses wartet eine Prinzessin namens „Releasefähigkeit".

Jeder Build, der den Prozess durchläuft, wird aus unterschiedlichen Perspektiven (Code, GUI, Performance, usw.) evaluiert, wobei die Initialisierung des Prozesses anhand unseres CI-Systems angestoßen wird (und zwar bei jedem Commit (!)).

Folgende Abbildung verschafft einen Überblick, wie eine neue Revision aus unterschiedlichen Perspektiven evaluiert wird.

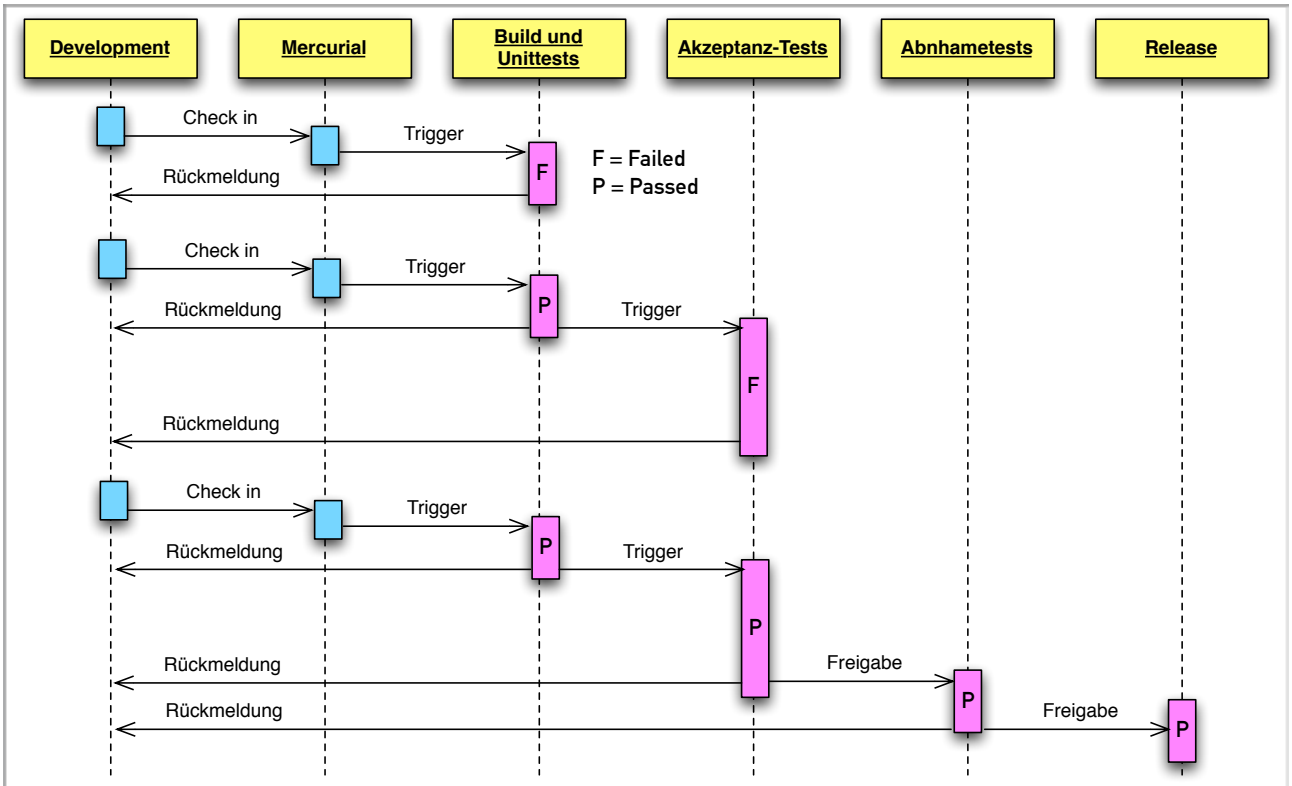

Abbildung 4: Änderung wird aus unterschiedlichen Perspektiven validiert

Das Ziel der Deployment Pipeline ist unfähige Release-Kandidaten bereits in frühen Stadien des Entwicklungsprozesses zu identifizieren und Feedback über die Ursachen schnellstmöglich zu liefern. Dafür wird jeder Build, der eine Phase nicht übersteht, automatisch als „unfähiger Release-Kandidat" gekennzeichnet und nicht an die nächste Phase weitergeleitet. Ausnahmen bilden die Übergänge von Akzeptanz-Tests zu Abnahmetests sowie von Abnahmetests zum Release, da auftretende Fehler subjektiv zu bewerten sind und ggfls. toleriert werden können.

[4] Der Begriff Pipeline kommt vom englischen Wort „Pipe" und bezeichnet den Datenstrom zwischen zwei Prozessoren nach dem FIFO-Prinzip, wobei der Fokus bei der Parallelisierung der Ausführung liegt.

Management von Artefakten

Jedes Mal wenn das System neu gebaut wird, wird der erstellte Build in Form eines Artefakts im CI-System abgelegt. Ein fundamentales Prinzip bei der Umsetzung der Deployment Pipeline ist, dass das Artefakt bei der Initialisierung des Prozesses ein einziges Mal erstellt wird. Das verhindert, dass der Code während des Prozesses neu kompiliert wird und „eingeschlichene" Fehler in späteren Phasen nicht abgefangen werden.

Das Problem bei diesem Ansatz ist, dass jedes Artefakt aufbewahrt wird und zwar unabhängig davon, ob es jemals released oder verwendet wird. Hierfür muss ein Kompromiss gefunden werden, sodass ältere Artefakten des Systems ab einer bestimmten Zeit endgültig gelöscht werden dürfen. Wo die Artefakten letztendlich gespeichert werden, nachdem sie den Delivery-Prozess erfolgreich durchlaufen haben, ist abhängig davon, wie lange sie aufbewahrt werden. Wichtig ist nur, dass es sich hierbei um ein File-System mit regelmäßigen Backups handelt. Dieses wird in der Praxis auch „Artifact Repository" genannt. Siehe bitte Abschnitt „Commit Stage" für mehr Information über das „Artifact Repository".

Eine weitere Technik, die sich in der Praxis bewährt hat, ist das Taggen der Artefakten mit Informationen über die Revision, aus der das Artefakt entstanden ist. Hiermit kann der Zusammenhang zwischen Source-Code, Artefakten und Systemkonfiguration in allen Phasen des Delivery-Prozesses hergestellt und sichtbar gemacht werden.

Die Phasen der Deployment Pipeline

Um die Essenz einer Deployment Pipeline besser darzustellen, werden die einzelnen Phasen der Pipeline anhand eines Beispiels[5] näher erläutert (vgl. Abbildung 5).

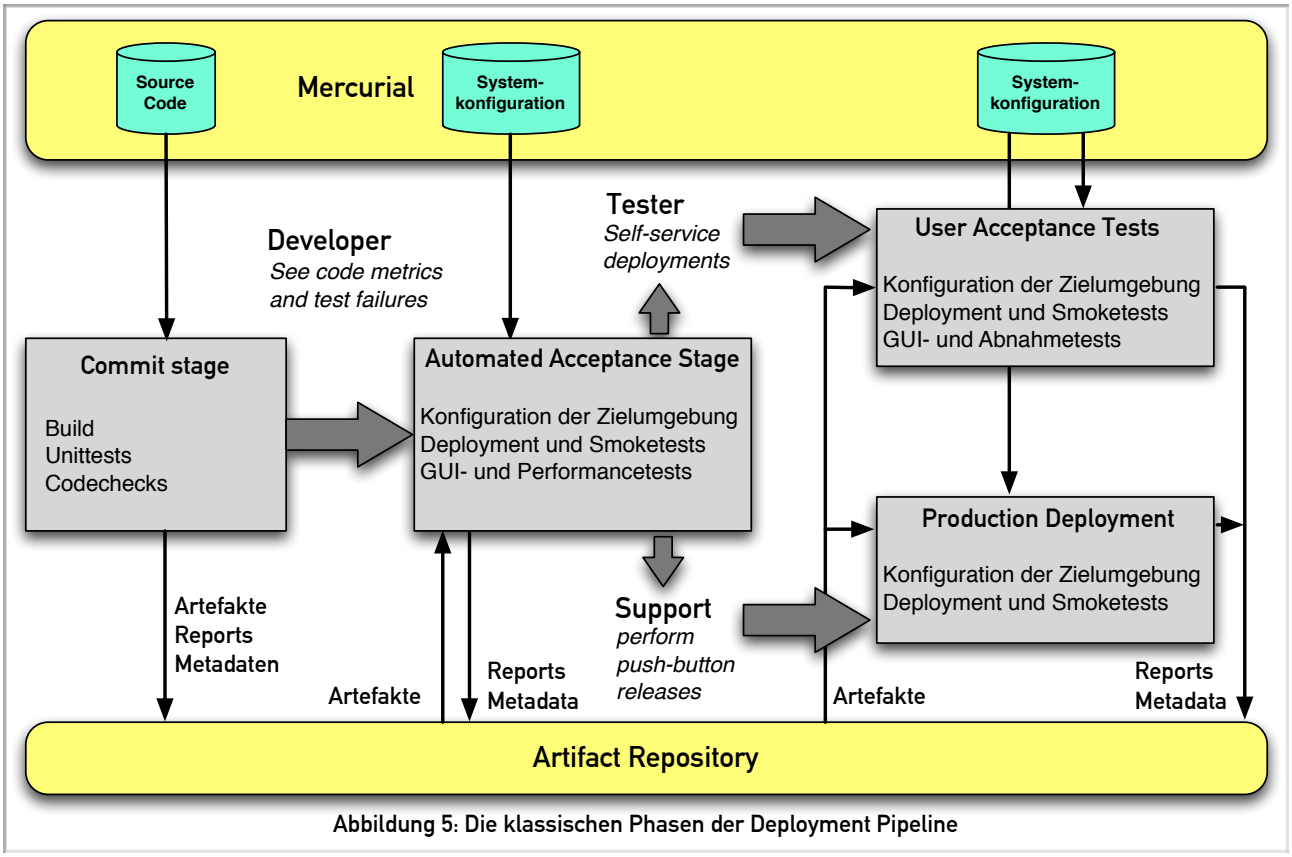

Abbildung 5: Die klassischen Phasen der Deployment Pipeline

[5] Vereinfachtes Beispiel für den Fall von Globalpark

Commit Stage

Der Delivery-Prozess startet, sobald die Entwickler eine neue Änderung in Mercurial committen. Ab diesem Moment reagiert das CI-System auf die Änderung, indem eine neue ***Instanz*** der Deployment Pipeline gestartet wird. In der ersten Phase (**Commit Stage**) wird der Code „kompiliert". Anschließend werden die Unittests und die Codechecks ausgeführt. Sind alle Tests erfolgreich gelaufen, so wird das deployable System in Form eines Artefakts im Artifact Repository abgelegt. Um die Artefakte sowohl für alle User als auch für die nächsten Phasen der Pipeline verfügbar zu machen und um diese besser zu verwalten, können sog. Repository Manager wie z.B. Nexus oder Artifactory eingesetzt werden. Mit Hilfe solcher Systeme können nicht nur Artefakte sondern auch Reports und Metadaten mit Informationen über die „kompilierte" Version mit gespeichert werden.

Das Hauptziel der Commit Stage ist, Fehler schnellstmöglich zu identifizieren. Sie dient sozusagen als ein Check-In-Gate, das für die Freigabe des Artefakts für die nächsten Phasen zuständig ist. Die Commit Stage sollte im besten Fall ein interaktiver Prozess sein. D.h., dass die Entwickler erst „warten sollten" bis die Tests gelaufen sind, damit sie mit ihren nächsten Tasks weitermachen können. Das ist auch der wesentliche Grund warum die Ausführung der Tests nicht auf dem CI-System stattfindet, sondern auf dedizierten Testrechnern, die eine parallele Ausführung der Tests unterstützen. Die Begründung, warum Entwickler in der Regel nur auf die Ergebnisse der Commit Stage warten sollen und nicht auf die aller Phasen der Pipeline, liegt an der Tatsache, dass die vollständige Ausführung der Pipeline zu „lange" dauert. In der schönen Welt würden alle Tests in wenigen Sekunden durchlaufen, was in der realen Welt nicht immer (nie) zu vertreten ist.

Automated Acceptance Stage

Automatisierte Unittests und Codechecks sind wesentliche Bestandteile jedes agilen Entwicklungsprozesses. Allerdings sind diese bei komplexen Softwareprojekten oft nicht ausreichend. Es gibt in der Praxis genug Beispiele von Softwareprojekten, bei denen tausende von Unitests erfolgreich ausgeführt werden, aber die funktionalen und **vor allem nicht-funktionalen Anforderungen**, wie z.B. Performance und Browserunterstützung, trotzdem nicht erfüllt werden.

Die zweite Phase (**Automated Acceptance Stage**) wird am Ende der Commit Stage angestoßen und dauert in der Regel länger als die erste Phase. Hier wird die Zielumgebung für das Deployment erst vorbereitet. Dies wird z.B. anhand von Konfigurationsdateien (die ebenfalls in Mercurial abgelegt werden (!)) erledigt. Anschließend wird das Artefakt automatisch in die Zielumgebung deployed. Mit Hilfe von Smoketests wird sichergestellt, dass die Umgebung vor und nach dem Deployment gültig ist (z.B. mit Hilfe von Soll-Ist-Vergleichsskripten). Im nächsten Schritt werden GUI- und Performancetests ausgeführt, die die vom Testmanagement definierten Akzeptanz-Kriterien kontinuierlich überprüfen. Auch hier ist extrem wichtig, dass die Tests parallel auf dedizierten Testrechner laufen und besonders aussagekräftig sind.

User Acceptance Tests und Production Deployment

Nachdem das Artefakt die ersten zwei Stages der Pipeline erfolgreich durchlaufen hat, soll jedes Teammitglied in der Lage sein, das Artefakt in den gewünschten Zielumgebungen zu deployen. Tester und Support können nicht nur **selber** entscheiden, welcher Release-Kandidat in welcher Umgebung deployed wird, sondern auch wichtige Informationen über den Status der ersten zwei Phasen des Prozesses (Check-In-Kommentaren) erhalten. Typische Zielumgebungen sind Testumgebungen, bei denen der Tester den Release-Kandidat auf inhaltliche Anforderungen überprüfen kann (UAT)[6]. Wurde der Release-Kandidat von der QA abgenommen, so kann der Tester den RC „promoten" indem er ihn als „QA approved" taggt. Ein „approved RC" ist die Freigabe für den Support, damit der RC in die Produktivumgebung deployed werden kann.

In den meisten Fällen ist die Freigabe eines RC durch die QA nicht immer zwingend notwendig. Viel wichtiger bei diesem Ansatz ist, dass der Workflow (von Check-In bis zum Kunden) nicht in Stein gemeißelt und dadurch unflexibel wird. Wenn z.B ein Critical-Bug-Fix schnellstmöglich an den Kunden gebracht werden muss, muss die Möglichkeit bestehen, den Prozess so zu optimieren, dass mehrere Phasen parallel ausgeführt werden. Es soll z.B. möglich sein, dass unmittelbar nach der Commit Stage mit manuellen explorativen Tests angefangen werden kann.

---

[6] User Acceptance Test

Die Möglichkeit entscheiden zu können welcher RC[7] in welcher Umgebung deployed wird, ist bei der Einführung einer Deployment-Pipeline die oberste Priorität. Nur so kann die Traceability (Version <-> Feature <-> Bugfix <-> Environment) sichergestellt werden. In diesem White Paper wird der Begriff „Self-Service Deployment System (SSD-System)“ eingeführt. Er beschreibt die ersten Überlegungen, wie ein derartiges System anhand eigener Entwicklung umgesetzt werden könnte.

Grundlegende Ideen zur Standardisierung des Deploymentprozesses

Um vorhandenes Know-How auszunutzen, könnte das SSD-System in Form einer Webapplikation umgesetzt werden. Abbildungen 6 und 7 verschaffen einen Überblick wie das System in der Praxis aussehen könnte.

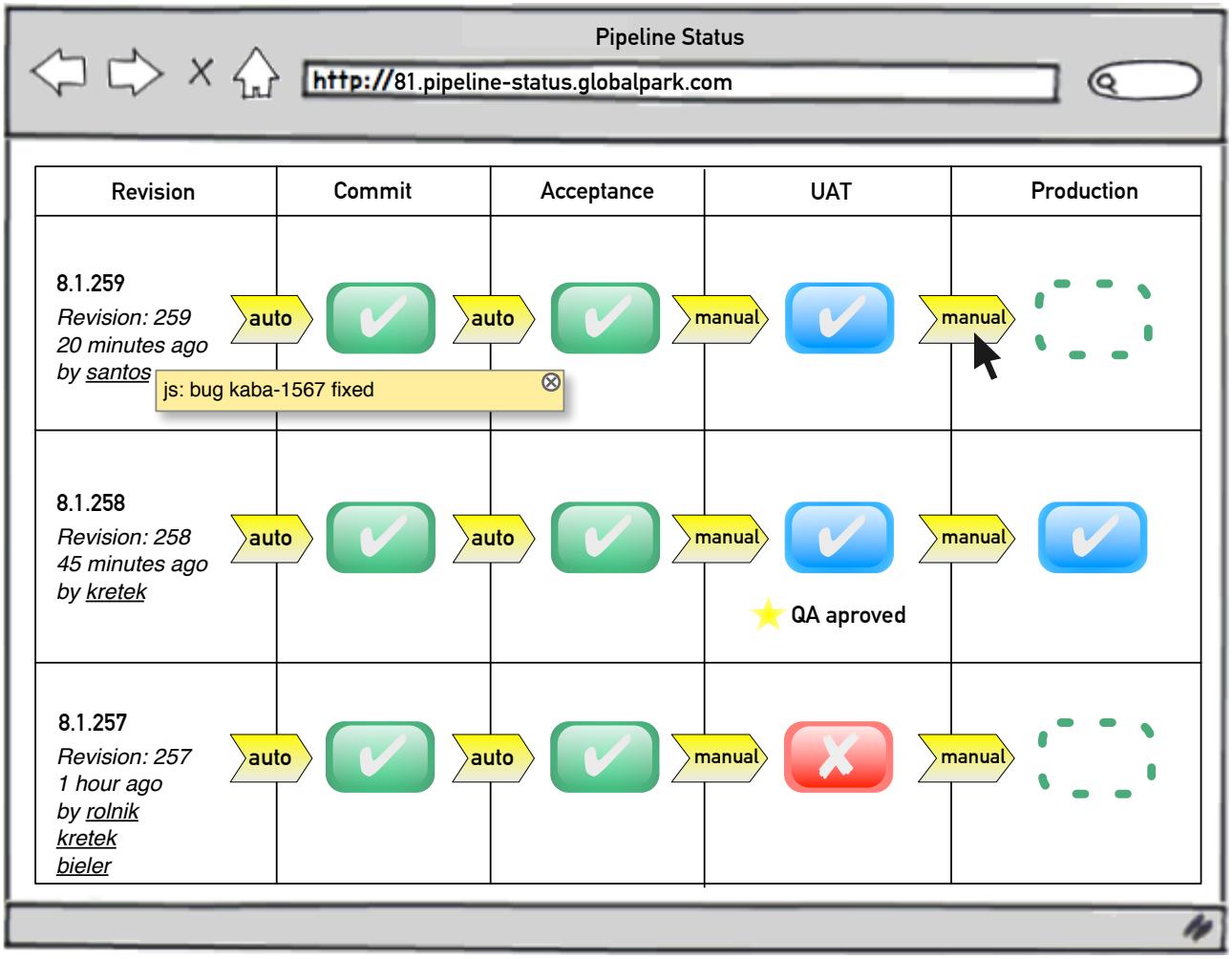

Abbildung 6: Mögliche Darstellung der Pipeline Status

Mit Hilfe eines SSD-Systems sollen Tester in der Lage sein, die Status und Informationen der einzelnen Pipeline-Phasen zu überprüfen und den entsprechenden Release-Kandidaten in die gewünschten Testumgebungen zu deployen (vgl. Abbildung 6). Das gleiche Prinzip gilt für die nächsten Phasen der Pipeline, wobei hier der Support oder die System Engineerings für das Deployment in den Produktivumgebungen zuständig wären.

Die Rollen bei der Deployment Pipeline

Damit der Deploymentprozess nicht unbefugt angestoßen wird, kann ein kleines Benutzerverwaltungssystem mit Rollen und Gruppen eingeführt werden, wie z.B. QA, Operations und Developer, wobei Developers nur in ihren eigenen Umgebungen deployen können. Diese Rollen werden vom Release und Deployment Manager festgelegt.

[7] Release Candidate

Die Deployment-Skripte

Wichtig bei der Deployment-Phase ist, dass der Deployment-Prozess vollkommen automatisiert stattfindet. Unabhängig davon, welcher Anwender den Deploymentprozess anstoßt, es soll immer in der gleichen Form deployed werden. Dafür brauchen wir Skripte, die bestimmte Checks ausführen und den RC in die gewünschten Zielumgebunden ausrollen. In der Praxis wird das Masterskript häufig „Conan, der Deployer“ genannt. Bei Globalpark hat sich die Bezeichnung „ROM“[8] für diese Skriptsammlung etabliert.

Mit diesen Skripten soll möglich sein, alle wichtigen Überprüfungen durchzuführen, die für das sichere Deployen eines Releases notwendig sind (wie z.B. Verfügbarkeit der Umgebung, Upgrademöglichkeit und Vergleich vom Soll-, Ist-Zustand).

Jedes Mal wenn der manuelle Deployment-Prozess angestoßen wird (Mauszeiger in Abbildung 6), wird der Anwender zu der sog. Deployment-Ansicht weitergeleitet. Mit Hilfe von Skripten und Konfigurationsdateien kann der User den Host sowie die Installation, die geupdatet/geupgraded werden soll, auswählen (Abbildung 7). Beim Auswählen einer Installation werden **alle(!)** für das Deployment relevanten Informationen geladen. Der Anwender kann z.B. sehen, welches Release z.Z. aktiv ist sowie welche Releases für diese Installationen zur Verfügung stehen und zwar unabhängig davon, ob es sich um eine Build- oder Mercurial/CVS-Installation handelt. Für die Ausführung der Skripte soll der Installationstyp gleichgültig sein. Überprüfungen erfolgen im Hintergrund und bleiben somit dem Anwender verborgen.

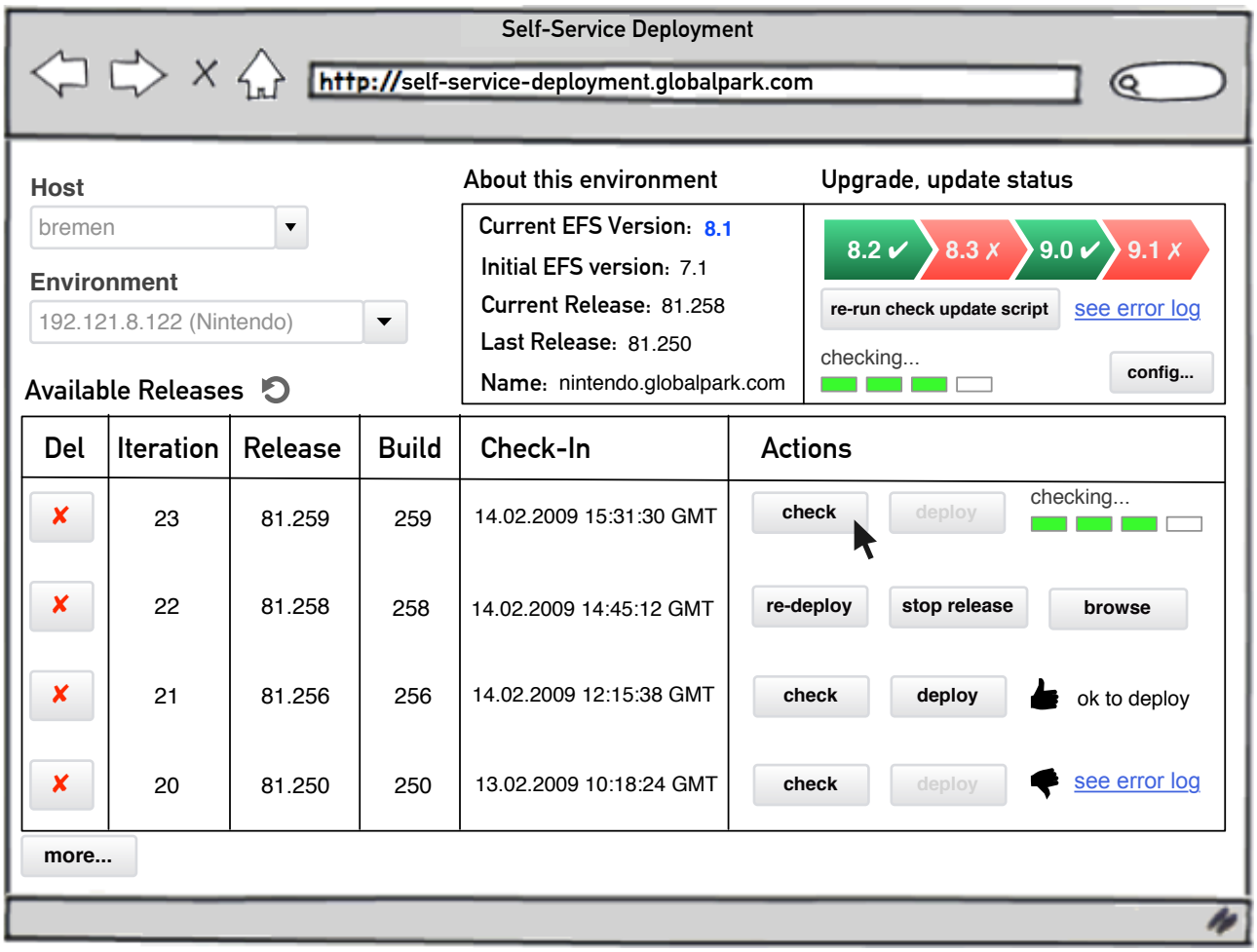

Abbildung 7: Self-Service deployment

[8] Rollout Management 2.0

Weiterführende Möglichkeiten

Zusätzlich zu der Deployment-Ansicht können wichtige Informationen anhand von Überprüfungsskripten für den Support und System Engineerings bereitgestellt werden. Eine wichtige Information wäre z.B. zu wissen, wie kompatibel eine Installation mit den nächsten (Major-)Releases ist. Dafür könnte das aktuelle „check upgrade script" erweitert werden, um Soll-Ist-Vergleiche „on the fly" ausführen zu können. Bedingung dafür ist, dass jede Installation einen Ist-Zustand hat, der anhand von Konfigurationsdateien und Checks definiert wird. Um den Soll-Zustand anhand von Konfigurationsdateien und Checks zu definieren, können Muster-Installationen, die immer up-to-date sind, eingesetzt werden (mehr zum Thema Soll-Ist-Vergleich in diesem Abschnitt).

Wesentliche Vorteile eines SSD-Systems:

i. Kontrolle über die Systemkonfiguration:
Je weniger Kontrolle wir über die Umgebungen haben, in denen unsere Releases deployed werden, desto größer ist die Wahrscheinlichkeit, dass unbekannte Probleme auftreten. Im Idealfall möchten wir jedes einzelne Bit, das deployed wird, nachvollziehen können. Mit Hilfe eines SSD-Systems ist die aktuelle Systemkonfiguration für jeden vor und nach dem Deployment sichtbar.

Mit diesen Informationen können Tester Umgebungen besser überwachen und Probleme präziser diagnostizieren. Diese Möglichkeit führt zu schnellerem Feedback und somit implizit zu schnelleren Bug-Fixes. Je komplexer und verteilter das System wird und je spezifischer die Systemkonfigurationen werden, desto wichtiger ist die Möglichkeit, diese unter Kontrolle zu haben.

ii. Demokratisierung des Deploymentprozesses:
Mit der Standardisierung und Automatisierung des Deploymentprozesses wird der Delivery-Prozess demokratisiert. Tester, Entwickler und Support müssen sich nicht mehr auf Emails oder Jira-Tickets verlassen, um einen RC in den gewünschten Umgebungen zu deployen oder um zu wissen, ob die Umgebung überhaupt dafür geeignet ist.

Tester können genau entscheiden, welche Release-Version sie in welchen Testumgebungen deployen möchten und zwar ohne irgendwelche technische Expertise dafür haben zu müssen. Da der Deployment-Prozess nur das Drücken eines Knopfes bedeutet, kann die Version, die gerade getestet wird, öfters geändert werden. Im optimalen Fall können Tester zu vorigen Versionen rollbacken, um Vergleichstests „on demand" ausführen zu können.

Aber nicht nur das Development profitiert von einem SSD-System. Auch Sales oder Marketing können das „Knaller-Feature" per Knopf-Druck in Staging-Umgebungen und Showrooms deployen, um den Kunden vom Mehrwert des Produkts schneller zu überzeugen. Und das ohne die technische Hintergründe des Deploymentprozesses kennen zu müssen.

iii. Risikominimierung beim Deploymentprozess:
Die Sicherheit, dass das SSD-System immer robuster und dadurch auch das Risiko von auftretenden Problemen minimiert wird, wird durch den Prozess an sich gewährleistet. Dadurch, dass immer der gleiche Prozess für die Auslieferung der Software verwendet wird, wird dieser kontinuierlich (mehrmals täglich) getestet und optimiert.

iv. Software per Knopfdruck:
Wie in der Einleitung dieses White Paper erwähnt, verfolgt der Ansatz von Continuous Delivery das Ziel, Software per Knopfdruck bereitzustellen, um Feedback von Kunden und Stakeholder schnellstmöglich einzuholen sowie die mit jedem Release verbundene Risiken zu minimieren. Die Einführung eines SSD-Systems ist die letzte Instanz bei der Einführung einer Deployment Pipeline und kann zum Erreichen des Continuous-Delivery-Ziels erheblich beitragen.

Was kann im Auslieferungsprozess noch optimiert werden?

Es sind einige Optimierungsmöglichkeiten, die im Zusammenhang mit der Einführung von Continuous Delivery unbedingt berücksichtigt werden sollten:

Backing out Changes:
Bei der Einführung einer Deployment Pipeline ist es unabdingbar, dass wir Änderungen rollbacken können, falls überraschende Probleme in der Produktion oder in den Testumgebungen auftreten. Wir wissen allerdings, dass diese Möglichkeit, gerade bei Legacy-Systemen wie z.B. EFS, nicht immer besteht. Sobald die „Healself“ auf einer Installation ausgeführt und dabei Änderungen in der DB, im DB-Schema oder im Dateisystem vorgenommen wurden, können wir nur in den seltensten Fällen ohne weiteres rollbacken. In den nächsten Abschnitten werden einige grundlegende Ideen zum Thema dargelegt:

DB-Migration

In der Praxis existieren mehreren Ansätze, die uns erlauben „hot deployments“ und „hot rollbacks“ zu machen. Einer dieser Ansätze beschreibt die Trennung des DB-Migrationsprozesses vom Deploymentprozess wie in Abbildung 9 zu sehen ist. Dieser Ansatz ist besonders interessant, wenn das System öfters und ohne regelmäßige Änderungen in der DB released wird. Die Kompatibilität des Systems zur DB-Version wird hierbei anhand Konfigurationsdateien definiert und mit Hilfe von Checkskripten „on the fly“ ermittelt. Für die Versionierung der Datenbank können sog. Data Base Change Management Systeme wie z.B. Liquibase eingesetzt werden. Der wesentliche Vorteil von DBCM-Systeme wie Liquibase sind die Möglichkeiten die Datenbank zu versionieren, Datenbank-Versionen zu „diffen“ und die Unterschiede in Form von ausführbaren Changesets auszugeben sowie die automatische Generierung von Rollback-Statements für typische Update-Statements (create, rename, add, drop, usw.).

Eine weitere Möglichkeit ist, das System nicht nur für neue DB-Versionen kompatibel zu halten, sondern auch für die aktuelle Version (vgl. EFS 8.1-245 in Abbildung 9). Somit haben wir immer noch die Möglichkeit die Kompatibilität des Systems mit der aktuellen Version zu testen. Sind wir sicher, dass das System mit der aktuellen DB-Version einwandfrei funktioniert, so können wir eine DB-Migration durchführen und die nächste Version des Systems deployen (vgl. EFS 8.1-246 in Abbildung 9). Stellen wir ein Problem mit der nächsten Version fest, müssen wir nur rollbacken.

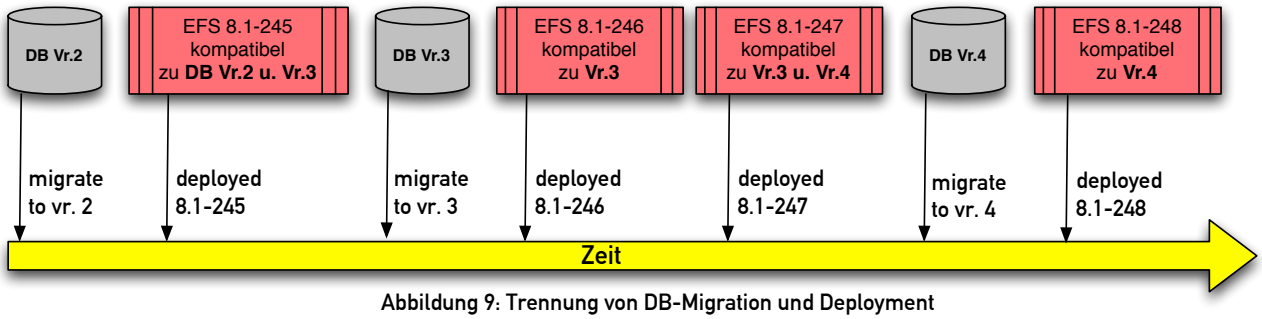

Abbildung 9: Trennung von DB-Migration und Deployment

Check-Update-Skript

Um den Deploymentprozess „on the fly“ anstoßen zu können, müssen wir in der Lage sein, die aktuelle Version der Software mit allen möglichen (Major-)Releases zu vergleichen. Zur Zeit werden Änderungen in der Healself und im Dateisystem mit Hilfe eines Check-Upgrade-Skripts überprüft. Das Problem bei diesem Ansatz ist, dass Überprüfungen explizit im Check-Upgrade-Skript eingetragen werden müssen. Vergisst ein Entwickler eine wichtige Überprüfung einzubauen, so wird diese vom Check-Upgrade-Skript nicht berücksichtigt. Je länger die Zeit zwischen den Releases, desto größer ist die Wahrscheinlichkeit, dass wichtige Überprüfungen vergessen werden.

Ein weiteres Problem bei diesem Ansatz ist, dass die Migrationstests erst am Ende der Entwicklungsphase ausgeführt werden. Das widerspricht der Idee von Continuous Delivery, da die Software nicht zu jedem Zeitpunkt releasefähig ist, sondern nur nachdem aufwändige Migrationstests ausgeführt wurden. Interessanter wäre in diesem Fall die Healself mit Hilfe von deklarativen Statements zu definieren.

Dadurch, dass Änderungen deklarativ in der Healself definiert werden, wären explizite Überprüfungen im Check-Upgrade-Skript nicht mehr notwendig. Das Check-Upgrade-Skript würde nur die notwendige Anweisungen für einen Soll-Ist-Vergleich beinhalten und somit unabhängig von der System-Version (wie z.B. check_upgrade_70_to_71.php) gemacht werden. Der Stand einer Installation könnte dann mit Hilfe von statischen Konfigurationsdateien, der Healself, der DB-Version und ggfs. von dynamischen Checks ermittelt werden. Kennt das Check-Upgrade-Skript den exakten Stand einer Installation, so kann es Soll-Ist-Vergleiche ausführen und die Kompatibilität einer Version zu (Major-)Releases in Laufzeit ermitteln.

Da der zukünftige Stand einer Version nur bekannt ist, nachdem die Healself ausgeführt wurde, könnten Muster-Installationen, die immer up-to-date sind und alle Module beinhalten, für die Vergleichstests eingesetzt werden.

Metriken:

Schnelles Feedback steht im Mittelpunkt jedes Auslieferungsprozesses. Das Einholen von Feedback in den früheren Entwicklungsphasen gibt uns die Möglichkeit schneller zu reagieren und somit den Auslieferungsprozess bestmöglich zu optimieren. Was wir allerdings dafür benötigen sind Metriken, an den wir uns orientieren können. Das Problem ist nur: Was sollen wir nun alles messen?

Das Verhalten des Development-Teams kann, je nachdem was gemessen und als Feedback eingeholt wird, erheblich beeinflusst werden. Mehrere Studien deuten darauf hin, dass wenn LOC[9] gemessen werden, tendieren Entwickler dazu weniger Code zu schreiben (Code Clean Development). Fangen wir an, die Anzahl der gefixten Bugs zu messen, so tendieren Tester dazu Tickets schneller zu schließen und Entwickler Bugs schneller zu fixen.

Metriken wie z.B. die Anzahl fehlgeschlagener Unittests, die Anzahl offener oder gefixter Bugs, LOC, usw. helfen uns sicherlich bei der Kontrolle und Optimierung der Softwarequalität weiter. Allerdings liefern diese Metriken alleine, keine Antwort auf die wichtigste Frage des Auslieferungsprozesses - „Wie lange brauchen wir, um eine Änderung zu releasen, die nur eine Code-Zeile betrifft?“ -. Um diese Frage beantworten zu können, benötigen wir vielmehr eine globale Metrik, die uns ermöglicht, zu identifizieren, ob der Auslieferungsprozess ***als ganzer*** optimiert werden kann.

Bei vielen Globalpark-Projekten konzentrieren wie uns zu sehr auf die Erhebung bekannter Qualitätsmetriken. Allerdings bringt uns die Erkenntnis, ob ein Bug schnellstmöglich gefixt wird oder ob unsere Unittests stabil laufen, nicht viel.

Wenn wir trotzdem ca. 4 Monate benötigen, um einen Bugfix oder Feature an den Kunden zu bringen, haben wir den Auslieferungsprozess dadurch längst nicht optimiert. Wenn wir uns aber auf die Optimierung des Prozesses an sich „von Check-In bis zur Auslieferung“ fokussieren, fordern wir implizit nicht nur die Verbesserung der Qualität, sondern auch anderer wichtiger Aspekte des gesamten Prozesses. Die Einführung einer Deployment Pipeline kann uns bei der Feststellung möglicher Engpässe im Auslieferungsprozess (von Check-In bis zur Auslieferung) unterstützen, indem der Prozess transparent gemacht wird (jeder kann sehen, was uns davon abhält, die Software an den Kunden zu bringen). Sind die Engpässe bekannt, so können diese nach dem Demingkreis-Prinzip (Plan-Do-Check-Act) optimiert werden.

Qualitätsmetriken sollen also **nicht (!)** als Maß für die Qualität des Auslieferungsprozesses gelten. Vielmehr sollen sie für die Verdeutlichung der „Software-Robustheit“ verwendet werden. Wichtig hierbei ist, dass der „Software-Gesundheitszustand“ allen Beteiligten sichtbar gemacht wird. Dafür können Qualitätsmanagement-Software wie z.B. Sonar eingesetzt werden. Mit Hilfen von Sonar können alle wichtigen Qualitätsmetriken eines Softwareprojektes zusammengefasst und auf einem Dashboard dargestellt werden. Die Visualisierung ist leicht verständlich und kann sogar vom Management für demonstrativen Zwecken verwendet werden (vgl. Abbildungen 8 und 9).

[9] Lines of Code

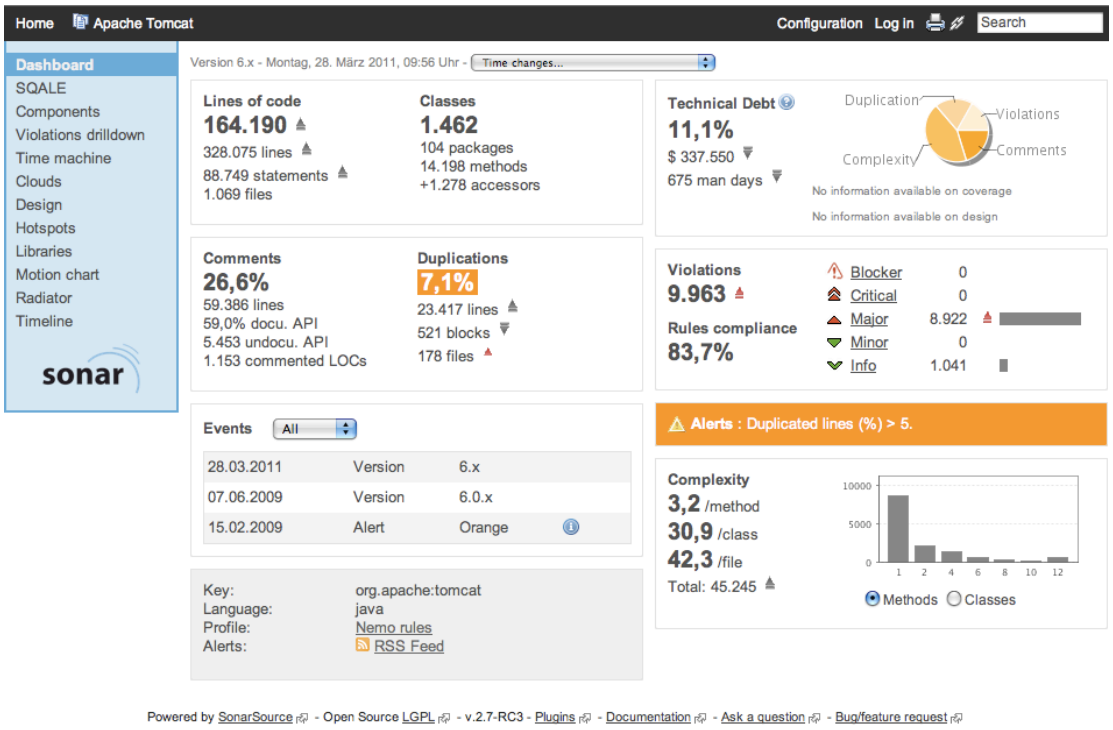

Abbildung 8: Sonar Dashboard

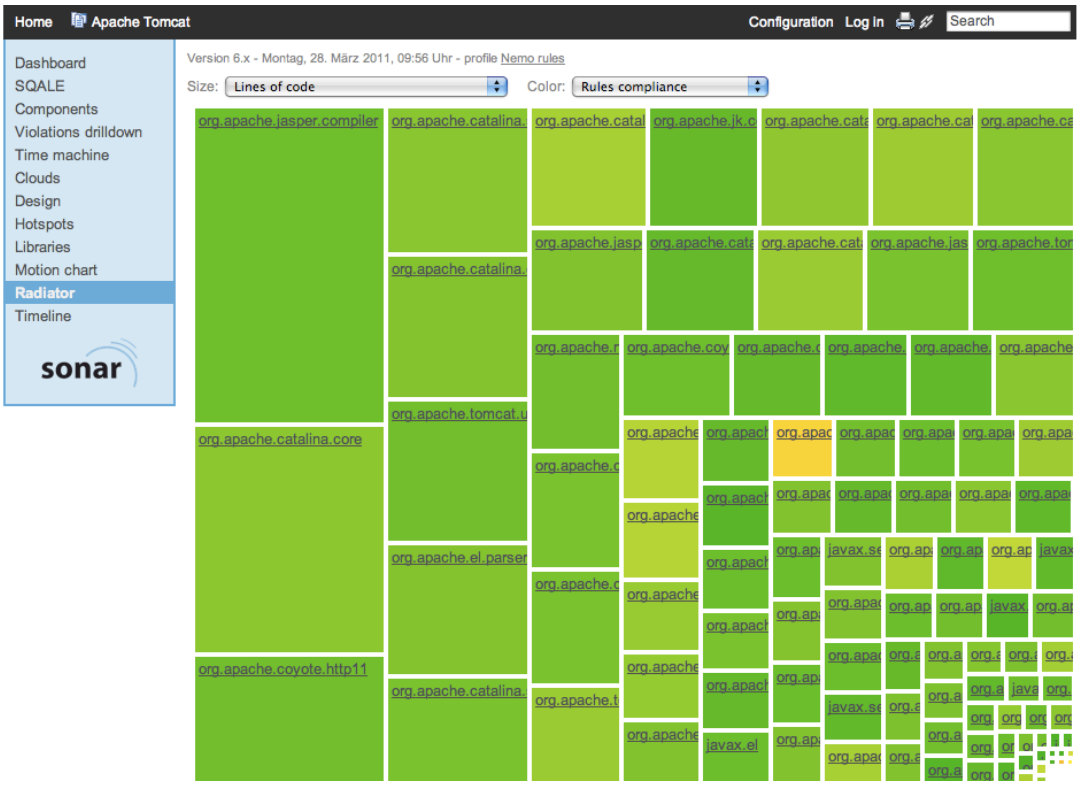

Abbildung 9: Sonar Radiator