Wissenschaftlicher Fachtext

Software Quality Management (QM-VU)
Teil 2

Praktiken als Voraussetzung zur Verkürzung von Releasezyklen qualitativ hochwertiger Software

Name	Matr-Nr. / Kennz.	E-Mail Adresse
Bernhard Fleck	0325551 / 937	bernhard.fleck@gmail.com
Claus Polanka	0225648 / 534	e0225648@student.tuwien.ac.at

Status: Abstract
Datum: 13. Januar 2012

Inhaltsverzeichnis

Abstract

Diese Arbeit beschäftigt sich mit der Ausarbeitung von qualitätssichernden Maßnahmen, die es Software-Unternehmen ermöglichen sollen, Deploymentzyklen drastisch zu verkürzen um rasch auf Veränderungen am Markt reagieren zu können. IT-Unternehmen die nur einmal jährlich eine neue Version ihrer Produkte zur Verfügung stellen, könnten künftig große Probleme haben konkurrenzfähig zu bleiben. Diese Arbeit befasst sich daher mit der Frage, welche Entwicklungspraktiken zu einem bestimmten Vorgehensmodell hinzugefügt bzw. von diesem entfernt werden müssen um die Deploymentgeschwindigkeit von jährlichen auf dreimonatige, auf monatliche, auf wöchentliche, auf tägliche und zu guter Letzt auf stündliche Zyklen zu verkürzen. Da sich die Disziplin des Software-Engineerings nicht verallgemeinern lässt, können bestimmte Techniken für bestimmte Deploymentzyklen positive, für andere jedoch negative Auswirkungen haben. Wenn man sich vorstellen würde, dass man zwei Entwicklungsteams, die mit unterschiedlich langen Deploymentzyklen arbeiten, nach ihren Praktiken befragen würde, was würde man als Antwort bekommen? Eines ist klar, Software-Entwickler müssen prinzipiell dieselben Probleme lösen und zwar von der Idee bis zum tatsächlichen Bereitstellen des Produkts für den Endanwender. Nur die Art und Weise wie die Software umgesetzt wird unterscheidet sich dramatisch je nach Länge des verwendeten Deploymentzyklus. In dieser Arbeit wollen wir detailliert darauf eingehen, welche Techniken notwendig sind, um einerseits hohe Qualität der Software zu garantieren und um andererseits die Entwicklungszyklen drastisch zu verkürzen.

1 Einleitung

Allgemeine Einleitung der Thematik. Vielleicht eine Abgrenzung zwischen Continuous Delivery und Continuous Deployment vornehmen. Konzepte von Entwicklung und Deployment beschreiben. Die Frage herausarbeiten warum kürzere Releasezyklen notwendig/besser sind. Dabei nicht auf die Qualitäsaspekte vergessen.

2 Related Work

Konzepte von Entwicklung und Deployment beschreiben. Die Frage herausarbeiten warum kürzere Releasezyklen notwendig/besser sind. Dabei nicht auf die Qualitäsaspekte vergessen.

3 Best-Practices

Beschreibt und beantwortet die in Related Work erarbeitete Fragestellung indem Best Practices vorgestellt/herausgearbeitet werden. Weiters wird unser Vorgehen beschrieben. Und zwar wie iterativ eine immer kleinere Kadenz erreicht werden kann. In den Unterkapiteln werden die einzelnen Iterationsschritte vorgestellt.

3.1 Jahr auf Quartal

In Unternehmen die nur einmal im Jahr eine neue Version ihres Produktes bereitstellen, kommen häufig lineare, nicht iterative Vorgehensmodelle zum Einsatz. Eines der bekanntesten Beispiele dafür ist das Wasserfallmodell, dass aus verschiedenen Phasen wie z.B.: der Analyse, dem Entwurf, der Realisierung (Implementierung) und dem Testen besteht. Wenn man diese Art von langwierigen Prozess verfolgt, dann ist es für ein Unternehmen relativ schwer auf Veränderungen am weltweiten Markt rasch reagieren zu können. Aktualisiert man nur einmal im Jahr das Produkt läuft man Gefahr, dass die verwendete Technologie höchstwahrscheinlich schon längst wieder veraltet ist. Daher muss ein Unternehmen um am Markt konkurrenzfähig zu bleiben, auf kürzere Release-Zyklen setzen.

Die dabei nächst kürzere Iterationslänge würde z.B.: drei Monate betragen. Dabei könnte ein Unternehmen einmal im Quartal eine neue Version ihrer Software bereitstellen. Die große Herausforderung ist die bisherigen Prozesse so anzupassen, um dieselben Probleme in kürzerer Zeit zu lösen. Eine Vorgehensweise die dabei offensichtlich nicht funktionieren würde, ist nichts im Unternehmen zu verändern, jedoch alle bisherigen Praktiken nun in drei Monaten durchzuführen. Da diese Art der Komprimierung nicht funktionieren kann, müssen daher fundamentale Techniken adaptiert werden um bestimmte Aufgaben zu gewissen Zeitpunkten in der kürzeren Iteration durchzuführen.

3.1.1 Automatisierte Akzeptanztests

Manuelles Testen der Software ist zeitaufwendig und fehleranfällig. Hat man eine Iterationslänge von einem Jahr, kann man jedoch ohne weiteres diese Art von Testen durchführen. Nach Beendigung der Implementierungsphase wird die Software an die QA-Abteilung weitergeleitet, die dann mit dem ausführlichen verifizieren des Produkts beschäftigt ist. Hat ein Unternehmen jedoch nur drei Monate Zeit, dann wäre es zu aufwändig diesen Prozess jedes Quartal wiederholen zu müssen. Die Automatisierung einer Regressions-Test-Suite, die auf *Knopfdruck* ausgeführt werden kann, hilft dem Unternehmen dabei manuelle Tests aus dem Entwicklungsprozess zu entfernen. Dabei erhalten die Entwickler nach kürzester Zeit Feedback über das Funktionsverhalten des Systems. Diese Tests können in Form von Akzeptanztests realisiert werden. Bei einer Iterationslänge von drei Monaten muss die endgültige Form bzw. Geschwindigkeit der Tests nicht perfekt

oder äußerst schnell sein. Wesentlich ist nur, dass diese Tests automatisiert durchgeführt werden können. Die entstehenden Wartezeiten sind dabei für den Projekterfolg nicht kritisch.

3.1.2 Refactoring

Im Wasserfallmodell gibt es eine eigene Phase für den Entwurf der Software. Möchte man alle drei Monate eine neue Version der Software bereitstellen, muss das Designen der Software-Architektur über die gesamte Iteration verteilt werden. Die Zeit für eine eigene Entwurfs-Phase steht bei dieser Iterationslänge nicht zur Verfügung. Da es jetzt kein *Big Design Up- Front* geben kann, und das Design kontinuierlich den Gegebenheiten angepasst werden muss, müssen Softwareentwickler die Technik des *Refactorings* ausgezeichnet beherrschen. Dadurch wird es möglich, große Design-Änderungen in kleinen, sicheren Schritten durchzuführen ohne jedoch dabei das Systemverhalten zu verändern. Entwickler übernehmen dabei die Verantwortung, regelmäßig in den Softwareentwurf des Produkts zu investieren.

3.1.3 Continuous Integration

Auch bei dieser Technik gilt dieselbe Argumentation wie auch schon beim Refactoring. In einer dreimonatigen Iteration bleibt keine Zeit für eine eigene Integrationsphase der Software. Sollte es z.B.: kurz vor einem neuen Release zu Problemen bezüglich des Zusammenbaus des Produkts kommen, die von den Entwicklern während der Implementierung nicht berücksichtigt wurden, könnte eventuell ein weiterer Release-Zyklus erforderlich sein, diese um all diese Probleme zu beheben. Deshalb muss eine Möglichkeit für die kontinuierliche Integration der Software in Form eines Build-Servers geschaffen werden, auf dem die Entwickler täglich ihre erledigten Aufgaben hinzufügen können. Dadurch kann es am Ende der Iteration zu keinen überraschenden Komplikationen bezüglich des Gesamtprodukts kommen.

3.1.4 Subscription Modell

Muss ein Unternehmen nur einmal im Jahr den potentiellen Endkunden von der neuen Version der Software überzeugen, fällt dieses Vorgehen alle drei Monate deutlich schwieriger aus. Man kann den Kunden als Unternehmen nicht dazu bringen, jedes Quartal für eine Aktualisierung der Software erneut zahlen zu lassen. Sollte das Geschäftsmodell der Firma jedoch vorsehen, dass der Kunde für Upgrades des Produkts zahlen muss, dann kann die Iterationslänge nicht auf häufigere Releases umgestellt werden. Leider verliert man dadurch auch die Vorteile des häufigeren Feedbacks des Benutzers, genauer gesagt all die Informationen die man aufgrund der Benutzung des Produkts durch den Endkunden und die Entwicklung des Produkts am Markt erhält. Daher muss das

Geschäftsmodell des Unternehmens ebenfalls angepasst werden und möglicherweise eine Form von Subscription-Modell eingeführt werden. Dabei zahlt der Kunde einmal im Jahr einen Pauschalbetrag und erhält sämtliche Upgrades der Software ohne weitere Bezahlung. Diese Art von Geschäftsmodell ist absolut kritisch für den Erfolg für eine Umstellung auf dreimonatige Release-Zyklen.

3.2 Quartal auf Monat

Bei jedem Übergang von einem längeren zu einem kürzeren Release-Zyklus ist es notwendig, gewisse bis jetzt vielleicht erfolgreich eingeführte Praktiken zu entfernen und neue zu adaptieren.

3.2.1 Programmierer schreiben Tests

Hat man bei einer dreimonatigen Iteration ca. 60 Arbeitstage Zeit die neue Version der Software zu entwickeln, ist es zeitlich nicht weiter tragisch, falls die Ausführungsgeschwindigkeit der Akzeptanztests einen Tag erfordern. In einem monatlichen Zyklus muss jedoch die Häufigkeit des Feedbacks für den Entwickler drastisch erhöht werden. Dabei muss ein Teil der Verifikationsarbeit den Programmierern übergeben werden, sodass diese in kürzeren Zeitabständen Informationen über den Zustand der Software einholen können.

Ein monatlicher Release-Zyklus macht es erforderlich, dass Entwickler selbst Tests schreiben und auch ausführen. Daraus kann man schließen, dass die Anzahl der Akzeptanztests nicht nur reduziert werden kann, sondern auch nicht mehr alle eventuellen Fehler aufdecken müssen, da diese bereits vorher von den Unit-Tests abgefangen werden.

3.2.2 Status Meetings

Auch die Art und Weise wie man andere Teammitglieder über durchgeführte Veränderungen informiert muss bei monatlichen Iterationen angepasst werden. Wurden vielleicht bis jetzt über alle Aktualisierungen der Software Protokolle für den Projekt-Manager geschrieben, der diese wiederum an andere Entwickler als Feedback weiterleitete, muss man nun eine Form des Wissenstransports schaffen, der bei weitem nicht so viel Zeit in Anspruch nimmt. Hat man nur noch 20 Arbeitstage für die Entwicklung eines Upgrades der Software, dann hat diese Art des relativ *schwergewichtigen*, formalen Prozesses keine Daseinsberechtigung.

Man benötigt daher eine Form der täglichen Status-Aktualisierung über Projektveränderungen der einzelnen Teammitglieder. Das kann z.B.: in Form eines täglichen Stand-Up oder Daily-Scrum Meetings erfolgen bei dem jeden Morgen jeder im Team kurz über Neuigkeiten bzw. eventuelle Probleme berichtet.

3.2.3 Task Board

Klassische Planungsprozesse bei denen zu erledigende Aufgaben mehrere Stationen (z.B.: Projektmanager, Analysten, etc.) durchwandern müssen, danach noch eventuell in Form eines Berichts niedergeschrieben werden, bevor sie der Programmierer zu Gesicht bekommt um daran arbeiten zu können, müssen ebenfalls angepasst werden.

Um Sinnvoll innerhalb eines Monats Planen zu können, werden daher transparente, visuelle Techniken benötigt die außerdem noch öffentlich zugänglich sein müssen. Dabei kann eine Art von Stellwand (task board) in Kombination von Karteikarten, wie sehr oft in Scrum Verwendung finden, eingesetzt werden. Dabei repräsentieren die Karten die durchzuführenden Aufgaben die jeweils in entsprechenden Zustandspalten auf der Stellwand platziert werden. Somit hat jedes Teammitglied zu jedem Zeitpunkt der Iteration eine Übersicht, welche Aufgaben noch innerhalb dieses Zyklus zu erledigen sind.

3.2.4 Pay-per-Use Modell

Bei monatlichen Releases kann es erneut sinnvoll sein über eine Anpassung des Geschäftsmodells nachzudenken. Bei dieser Iterationskürze könnte das *Pay- per-Use*-Modell eingeführt werden. Bei dreimonatigen Prozessen kann diese Art von Geschäftsmodell gefährlich sein. Sollte ein Release fehlerhaft sein, könnte das die Einnahmen des Unternehmens verringern, jedoch könnten die Entwickler erst drei Monate später darauf reagieren.

In einmonatigen Prozessen können viel schneller Korrekturen vorgenommen werden. Außerdem kann die Information über die tatsächliche Benutzung des Produkts als wertvolles Feedback angesehen werden. Bei Geld handelt es sich jedoch mit Abstand um das beste Feedback das man außerdem wieder in das Unternehmen investieren kann.

3.2.5 Notwendige Entfernung von Praktiken

Auch bei dieser Geschwindigkeitsüberführung ist es offensichtlich nicht möglich, in der gleichen Art und Weise Software zu entwickeln wie bisher. Jedoch gibt es gewisse Übereinstimmungen wie z.B.: die Akzeptanztests, die allerdings in einem einmonatigen Zyklus wesentlich schneller ablaufen müssen. Es fällt auf, dass gewisse Aufgaben die zuvor vielleicht nur von einer Person durchgeführt wurden, in kürzeren Iterationen von mehreren Teammitgliedern erledigt werden müssen. Auf die Häufigkeit der Durchführung und der Durchführungszeitpunkt verändern sich.

Allerdings wurden in der Überführung von jährlichen zu dreimonatigen Zyklen weitere notwendige Praktiken eingeführt, für die in einmonatigen Phasen keine Zeit mehr vorhanden ist. Diese Techniken waren äußerst hilfreich um die Entwicklungsgeschwindigkeit in

einem ersten Verkürzungsprozess zu erhöhen. Sie haben dabei geholfen regelmäße Upgrades der Software bereitzustellen, jedoch war das Erlernen und Einhalten der Techniken für jedes Teammitglied sehr aufwändig.

3.2.6 QA-Abteilung

Bei einmonatigen Iterationen werden diese Praktiken jedoch zur Last. Z.B.: ist das Vorhandensein einer QA-Abteilung aufgrund der organisatorischen Entfernung nicht mehr möglich. Diese Abteilung darf jedoch nicht mit der Rolle der Tester verwechselt werden.

Natürlich darf auch der psychologische Effekt bezüglich der Reduzierung dieser in längeren Iterationen noch so wichtigen Einrichtung nicht vergessen werden. Ein Entwickler der viel Erfahrung mit dreimonatigen Release-Zyklen besitzt, für den die QA-Abteilung der erste Schritt aus dem Chaos war, d.h. man tatsächlich Software am Ende des Quartals bereitstellen konnte, die noch dazu für den Kunden problemlos funktioniert hat, für diesen Entwickler ist eine QA-Abteilung unverzichtbar. Er kann sich nicht vorstellen, wie man in einem Monat Software erfolgreich bereitstellen soll, ohne der Unterstützung dieses *Fehlerfangnetzes*. Aus der Perspektive dieses Entwicklers sind dessen Argumentationen völlig nachvollziehbar.

Betrachtet man jedoch den folgenden Prozess genauer, bei dem jede Funktionsänderung der Software über den Projektmanager zur QA-Abteilung weitergeleitet wird, damit diese dann notwendige Ressourcen allokieren kann um die erhaltene Anfrage bearbeiten zu können, erkennt man, dass man in einem Monat nicht genug Zeit für diese Vorgehensweise hat.

Obwohl die QA-Abteilung die erhaltenen Anfragen aus Effizienzgründen in Warteschlangen organisiert damit dann ein Tester diese Aufgabe entnehmen und bearbeiten kann, dauert es für den Entwickler viel zu lange, bis er endlich Feedback erhält um darauf reagieren zu können. Wenn man jetzt noch darüber nachdenkt, dass die Programmierer die zurückbekommenen Antworten der Tester ebenfalls ähnlich organisieren und zum Beheben derselbe Prozess erneut durchgeführt werden muss kommt man zum Schluss, dass die Iteration bereits zu Ende ist bevor überhaupt nur eine Aufgabe abgeschlossen wurde. Man könnte natürlich die Software trotz der Tatsache bereitstellen, dass man nicht 100 Prozentig sicher weiß, ob das System Fehler enthält oder nicht. Allerdings ist das Bereitstellen von fehlerhaften Upgrades für den Endkunden auf Dauer nicht tragbar.

TODO: Ab hier muss noch bis zum Abschnitt 4 auf Seite 18 fertig ausformuliert werden.

D.h., dass das Q/A-Department, das kritisch für den Erfolg für jährliche bzw. dreimonatige Deployments war, wird zur unüberwindbaren Barriere in einmonatigen Prozessen und muss deshalb verworfen werden. Und wieder müssen dieselben Probleme nur jetzt

ohne Q/A-Department erledigt werden also was machen? Ganz einfach, Tester müssen dem Entwicklungsteam hinzugefügt werden. D.h. im Endeffekt hat man einen großen Raum in denen sich sowohl Tester als auch Entwickler befinden und diese unmittelbar miteinander kommunizieren können.

Nichts von den Vorgehensweisen über Warteschlangen, nichts von dem zeitaufwendigen weiterleiten von Requests über verschiedene Personen, sondern die unmittelbare Kommunikation zwischen Tester und Entwickler sind notwendig. Z.B. nimmt ein Entwickler Veränderungen an der Benutzerschnittstelle vor und berichtet dem Tester davon. Dieser kann sofort überprüfen ob die vorgenommen Änderungen fehlerfrei funktionieren. D.h. der Entwickler erhält nur wenige Augenblicke später sofortiges Feedback des Testers.

3.2.7 Mehrfach releaste Versionen

eines Produkts ist eine weitere Praktik die führe längere Zyklen äußerst hilfreich ist, jedoch bei kürzeren Phasen ein Problem dastellt. Z.B. verkauften wir unsere Software an einen Kunden. Dieser war äußerst zufrieden mit dem Produkt. Nach einiger Zeit fanden wir auch einen weiteren Abnehmer jedoch wollte dieser einige Veränderungen. Somit mussten wir eine neue Version der Software für den zweiten Kunden erstellen.

Wir versuchten die Änderungen auch dem ersten Kunden schmackhaft zu machen, allerdings war dieser mit seiner Version der Software zufrieden. Nach einiger Zeit bekamen wir einen Bug-Report und mussten diesen nun in beiden Versionen unseres Produkts beheben. Nach einigen Monaten kam der dritte Kunde hinzu, der ebenfalls einige Änderungen der Software vornehmen ließ. Nach drei Jahren des erfolgreichen Einsatzes der Software bestellten die Kunden einen Consultant und fragten, warum Änderungen so viel Zeit in Anspruch nahmen.

Das was Problem war, dass wir (vermutlicherweise) mittlerweile sieben verschiedene Versionen des Produkts verkauft hatten und wir in allen dieselben Fehlerbehebungen durchführen mussten. Außerdem hat jede Version eine unterschiedliche Anzahl von Features implementiert, die wir ebenfalls verwalteten mussten. Selbst bei jährlichen Deployments versuchen Software- Unternehmen ältere Versionen ihrer Produkte bei Kunden zu aktualisieren, da die Verwaltung mehrerer Versionen zu komplex geworden ist.

Jedoch kann man in so einer großen Zeitspanne mit diesem Overhead leben. Wenn man jedoch in einem Jahr zwölf neue Releases produziert, funktioniert diese Art von Versionierung nicht mehr. Bei dieser Veränderung handelt es sich nicht nur um eine technische sondern auch um eine Beziehungsänderung zwischen dem Kunden und der Organisation die die Software produziert. Der Kunde muss dem Unternehmen vertrauen können und es ist ja kein Geheimnis, dass viele Kunden genau das nicht tun, nämlich dem Unternehmen trauen, das das Produkt erstellt. Wie man vielleicht schon bemerkt hat, handelt es sich dabei all diesen Veränderungen nicht um ausschließlich technischen Veränderungen. Die CEO-zu- CEO-Beziehung der Unternehmen muss zu einer Partnerschaft abgeändert

werden. Sie müssen einander vertrauen können. Ein neues Release der Software darf nicht zur Qual für den Kunden werden.

Wenn ein Software-Unternehmen innerhalb von monatlichen Releases überleben möchte, darf ein Software-Upgrade kein Hindernis sein. Aus Kundensicht muss für so eine Vertrauensbasis aber zu 100% sichergestellt sein, das kein Datenverlust möglich ist, keine Produktivität verloren geht, man nicht an einem Montag dem 1. Jänner in die Firma kommt und eine komplette Zerstörung vorfindet. Der Kunde muss davon überzeugt sein, dass die Software solide ist. Der Aufbau einer solcher Beziehung zwischen Kunden und dem Software- Unternehmen kann Jahre in Anspruch nehmen. Es handelt sich nicht um einen Prozess der über Nach passieren kann.

3.2.8 Design Dokumente

Wenn man drei Monate für ein Release Zeit hat, kann man ohne Probleme Design-Skizzen produzieren, diese im Team bearbeiten und analysieren. Hat man allerdings nur genau 20 Werktage für das Erstellen einer neuen Version der Software, dann kann das Software-Design nicht im selben Umfang dokumentiert werden. D.h. aber nicht der komplette Verzicht auf das Design. Ganz im Gegenteil, man muss natürlich weiterhin ein ansprechendes Software-Design schaffen und dieses natürlich auch kommunizieren. Allerdings sind Dokumente dafür nicht das richtige Medium.

3.2.9 Change Requests

Change Request Process ist eine weitere organisational-Distance Barriere. Jemand fordert eine Änderung der Software. Diese Forderung wird in irgendein Change-Request-System eingearbeitet. Danach wird sie zum Change-Preventation- Board (Change Control Board) weitergeleitet und um effizient zu bleiben in eine Warteschlange eingereiht wird. Bei dem wöchentlichen Meeting des Boards werden dann sämtliche Forderungen abgelehnt. Stellt sich die Frage warum diese Forderungen, wenn sie sowieso immer abgelehnt werden, nicht unmittelbar beim Eintreffen abgelehnt werden.

Da könnte man sich die wöchentlichen Change-Request-Review-Meetings auch gleich sparen. Aufgrund der Zeitbegrenzung funktioniert diese Art von Prozess sowieso nicht mehr. Jedoch handelt es sich bei einem Change-Request um wertvolles Feedback und wenn man auf dieses innerhalb kurzer Zeit reagieren möchte, und es stehen einem nur 20 Werktage zur Verfügung, muss ein pragmatischer Ansatz verfolgt werden um diese Requests bearbeiten.

Z.B. könnte der Prozess die Change-Requests priorisieren wobei die entstandenen Prioritäten mit denen des Unternehmens harmonieren sollten. Man sollte klarerweise an den wichtigen Features arbeiten und die weniger wichtigen hinten anstellen. D.h. man

muss natürlich weiterhin Entscheidungen über die zu implementierenden Features treffen, jedoch hat die ursprüngliche Unternehmensstruktur die mittels CCB funktionierte, innerhalb von einem Monat nicht genügend Zeit um Prioritäten richtig zu setzen.

3.2.10 Separate Build- und Analyse-Teams

Separate Teams für Analyse und Build-Prozess nehmen ebenfalls zu viel Zeit in Anspruch. Ein Team dass sich ausschließlich um die Analyse kümmert, und danach ihr Ergebnis an die Entwickler weiterleitet, ist viel zu zweitaufwändig um innerhalb eines Monats ein neues Release zu erstellen. D.h. die Softwareentwickler müssen akzeptieren, gemeinsam für diese Dinge Verantwortung zu übernehmen.

Es kann natürlich jemanden in einem Team geben, der sich mit der Verfeinerung des Builds beschäftigt, jedoch gibt es keine extra Abteilung die sich ausschließlich um den Build der Software kümmert. In monatlichen Release- Zyklen bleibt keine Zeit um Build-Errors jemanden anderen zu zuteilen der sie dann behebt. Benötigt man z.B. eine Woche für das Beheben eines solchen Problems, dann ist es in einer 12-wöchigen Phase schmerzhaft, in einer vierwöchigen Phase jedoch nicht tragbar.

Entwickler müssen selbst dafür die Verantwortung übernehmen Build-Probleme zu beheben. Eine Rolle wie die des Build-Managers gibt es in einer monatliche Phase nicht mehr. D.h. die Organisation muss sich diesbezüglich verändern um einen vierwöchigen Release-Zyklus zu unterstützen.

3.3 Monat auf Woche

Es kann natürlich vorkommen, dass Kunden selbst für einen monatlichen Release- Zyklus zu ungeduldig sind und deshalb auf wöchentliche Updates ihrer Software bestehen. D.h. das Entwickler-Team steht wieder vor der Aufgabe, dieselben Probleme in noch kürzer Zeit zu bewältigen. Die erste Frage die sich bei der Umstellung auf wöchentliche Release-Zyklen stellt, wann soll die Software bereitgestellt werden? Freitag?

Nein, der letzte Tag der Woche ist ein äußerst schlechter Tag um neue Produktaktualisierungen dem Kunden zur Verfügung zu stellen, genauso am Tag bevor Weihnachten. Die Mitte der Woche scheint eine gute Idee zu sein. D.h. Jeden Mittwoch wird eine neue Version der Software bereitgestellt, sodass der Kunde mit dieser seine Arbeit verrichten kann. Dafür müssen wieder einmal die Organisation und der Zeitpunkt der Lösung angepasst werden um einen einwöchigen Release-Zyklus gerecht zu werden. Das Entwickler-Team also fünf Tage Zeit um ein für den Kunden wertvolles Inkrement zu erstellen. Wie soll man das anstellen? Erster Schritt ist eine Atomated Data Migration. Automatische Datenmigration

Bei einem einmonatigen Release-Zyklus hat man ein paar Tage Zeit um das DB- Schema anzupassen. Keinem Entwickler mach diese Art von Spaß, jedoch bleibt genügend Zeit

um es manell anzupassen. Innerhalb von einer Woche geht sich diese Arbeit allerdings nicht aus. Manuelle Anpassungen um Datenmigrationen durchzuführen sind einfach zu kostspielig. Eine voll automatisierte *1 -Button-Push* Datenmigration ist unabdingbar um Daten von einem alten zu einem neuen Zustand überzuführen.

Falls man einen Service 24/7 bereitstellen muss, dann muss man noch zusätzliche Dinge tun. Mehr davon bei täglichen Releases. Bei dieser Phase darf Datenmigration auf jeden Fall kein Problem mehr darstellen. Vor allem handelt es sich hierbei nur um ein Engineering-Problem. Dabei unterstützt ein Datastore automatisierte Datenmigrierung vielleicht besser als ein anderer. Dafür ist ja Engineering da, man nimmt die Zutaten die man hat und wendet sie passend am bestehenden Problem an.

3.3.1 Temporäre Branches

In einem einmonatigen Zyklus können Entwickler Branches für Änderungen der Software anlegen und diese nicht unmittelbar wieder in den Main-Branch zurück mergen. D.h. die Programmierer arbeiten z.B. über einen Zeitraum von einer Woche an einer Änderung in einem neu erstellten Branch und migrieren diesen dann nach Fertigstellung wieder zurück in den Trunk.

Dabei kann es zu komplizierteren Merges kommen, die einiges an Zeit beanspruchen. Genau diese Zeit ist in einem einwöchigen Zyklus nicht mehr vorhanden. Daher können Entwickler maximal temporäre Branches anlegen, die maximal nach ein paar Stunden wieder in den Main-Branch zurückmigriert werden. Dadurch verhindert man komplizierte Merges die aufgrund einer hohen Anzahl von Konflikten auftreten könnten.

Es stellt sich die Frage, ob bei temporären Branches nur halbfertige Änderungen wieder in die Main-Line zurückmigriert werden. Jedoch überwiegt die Verhinderung der auftretenden Kosten bei komplizierten Merges diese Tatsache. D.h. man hat keine andere Wahl als kurzweilige, temporäre Branches anzulegen da ansonsten einwöchige Release-Zyklen keine Chance auf Erfolg hätten.

Ein wichtige Begabung die Entwickler bei immer kürzeren Zyklenen haben sollte, ist die immer feinere Unterteilung von Arbeiten, sodass die Software nach einer Änderung trotzdem noch funktioniert. Z.B. hat man ein Refactoring von ca. 4000 LoC vor sich. Diese Änderungen müssen nicht alle auf einmal geschehen. Die Begabung eines Entwickler liegt nun bei der Unterteilung der vorzunehmenden Änderungen. Diese müssen so proportioniert werden, sodass nach einer Integrierung die Software noch voll funktionsfähig ist. Das Problem ist also die Reihenfolge der Änderungen sorgfältig zu bestimmen.

3.3.2 Keystoning

Hat man das Problem, dass ein zu implementierendes Feature mehr als eine Woche Zeit in Anspruch nimmt, man hat allerdings wöchentliche Releases, kann man zuerst die

notwendige Funktionalität erstellen, die der Endbenutzer nicht zu Gesicht bekommt. Im nächsten Release-Zyklus kann man dann die notwendigen UI- Anpassungen durchführen und

gewährleistet dadurch die Einhaltung der fünftägigen Software-Aktualisierung. D.h. man sollte bei einwöchigen Releases die sichtbaren Änderungen immer zum Schluss durchführen. Natürlich entsteht dadurch ein gewisses Risiko, da man ja Code in Produktion ausgeliefert hat, der nicht aufgerufen werden kann. Diesen Preis muss man aber für wöchentliche Releases bezahlen.

3.3.3 Kanban

Die Vorteile von Kanban werden bei wöchentlichen Release-Zyklen so richtig wertvoll. Genau wie all die bereits erwähnten Techniken kann Kanban natürlich auch bei länger andauernden Phasen zum Einsatz kommen. Bei einwöchigen Iterationen macht eine Art des Pull-Modells auf jeden Fall Sinn. Hat man in jedem Release-Zyklus eine fixe Anzahl von Tasks die zu erledigen sind und dieser Plan kann aufgrund von irgendwelchen Einflüssen nicht eingehalten werden, dann kann das zu Folgefehlern führen, sodass der Gesamtplan gefährdet ist.

Im Gegensatz dazu bietet Kanban die Möglichkeit mittels des Pull-Modells, dass ein Entwickler z.B. am Montag in der Früh eine neue Aufgabe vom Aufgaben-Stack entnimmt und diese abarbeitet. Sollte dieser Task mehr Zeit als angenommen beanspruchen, gefährdet dies keine anderen Aufgaben. Dadurch müssen keine Planänderungen durchgeführt bzw. keine entsprechenden Meetings einberufen werden.

3.3.4 One-Button-Deploy

Will man wöchentliche Release-Zyklen realisieren wird ein *One-Button- Deploy* unvermeidlich. Selbst wenn man einen vollen Monat Zeit hat, wird ein zweistündiger, manueller Deploy, bei dem noch dazu hin und wieder Fehler auftreten können, wirklich zur Qual für jeden Entwickler. Wenn man sich jetzt vorstellt, dass man für jeden Release diese fehleranfällige Arbeit erledigen muss, wird man schnell einsichtig, dass die Zeit dafür zu kostbar ist.

Auch hier gilt wieder, dass ein *One-Button-Release* ebenfalls für jährliche Entwicklungs-Phasen eingesetzt werden kann. Jedoch ist diese Technik bei dieser Geschwindigkeit nicht kritisch. Bei wöchentlichen Releases ist es auf jeden Fall kritisch.

3.3.5 Kein separates Test Team

In wöchentlichen Release-Zyklen unterscheidet man nicht mehr zwischen einem separaten Test-Team und einem Entwickler-Team. Selbst dieser geringe Abstand zwischen den beiden Teams wir in dieser kurzen Zeit zu aufwendig um ihn zu unterstützen. Der typische Arbeitsablauf bei dem ein Entwickler dem Tester neu hinzugefügte Funktionalität zum Testen bereitstellt, und dieser dann selbstständig das erwartete Verhalten der Software verifiziert.

Der Entwickler erhält dann unmittelbares Feedback vom Tester auf das er dann in entsprechender Art und Weise reagieren muss. Dieses Request-Response Verfahren ist zu zeitintensiv um in einer Woche eingesetzt zu werden. D.h. es gibt das Team das sowohl neue Funktionalität bereitstellt, also auch Feedback für diese erzeugt, auf das das Team entsprechend reagieren muss. Somit gilt, das Team muss für Implementierung und Testen gemeinsam Verantwortung übernehmen.

3.3.6 Keine Up-front Usability

Jeder Spezialbereich der Software-Entwicklung hat das Verlangen vor allen anderen berücksichtigt zu werden. Z.B. wird gerne die Software-Architektur als wichtigstes Kriterium für die erfolgreiche Erstellung des Produkts angesehen. Die Architektur sollte zu Beginn entworfen werden, danach kann die restliche, oft als nicht so wesentlich empfundene Arbeit umgesetzt werden. Genau dieselben Ansichten haben die Designer, die Tester, etc. In wöchentlichen Release-Zyklen kann diese Kurzsichtigkeit nicht unterstützt werden.

D.h. Entwickler können nicht fünf Tage auf das Architekturdokument warten, wenn man nur genau eine Woche Zeit hat, die gewünschten Features bereitzustellen. Daher muss die Aufgabe des Usability-Designs über die komplette Phase verteilt umgesetzt werden im Gegensatz zu einer Aufgabe, die nur zu Beginn des Zyklus stattfindet.

3.3.7 Aktiver Release-Branch

Bei wöchentlichen Iterationen wird die Pflege eines separaten Release-Banches, der aktiv verändert wird, zu zeitaufwändig und muss daher entfernt werden. Hat man einen Monat Zeit, und man muss eventuell einen Software-Patch einspielen der z.B. am 1. eines Monats deployt worden ist, ist es nicht kritischen diesen Patch auch im Entwickler-Branch nachzuziehen.

D.h. in diesem Fall sind zwei Branches noch ohne Weiteres umsetzbar. Bei fünftätigen Zyklen wird es zu aufwendig den Release-Brach aktiv zu aktualisieren, da man ja ständig die Software verändert. Jede Änderung doppelt zu warten ist in wöchentlichen Iterationen zu kostspielig. D.h. Kunden müssen sich eine Woche, dafür maximal eine Woche, auf einen Bug-Fix gedulden.

3.4 Woche auf Tag

Neben den technischen Herausforderungen für tägliche Releases erhält der Entwickler den angenehmen Nebeneffekt, dass die tägliche Arbeit bereits am nächsten Tag von irgendeinem Kunden bereits verwendet wird. Diese Tatsache darf als Motivationsfaktor für den Softwareentwickler nicht unterschätzt werden. Sehr oft haben Entwickler das Gefühl, dass ihre erledigte Arbeit für niemanden spürbar von Bedeutung sind.

3.4.1 Immunization

Bei täglichen Deployments ist diese Arbeit bereits am nächsten Tag von Bedeutung. Natürlich stellt sich dabei die Frage wie mit Fehlern umgegangen wird. Ist es überhaupt noch möglich bei dieser Kürze der Iteration Fehler Softwareentwickler zu akzeptieren? Sichtlich kann es sich ein Entwickler nicht mehr leisten, einen schlechten Tag zu haben.

D.h. sollte ein Team-Mitglied z.B. die Nacht zu vor zu wenig Schlaf bekommen haben und daher das Gefühl haben es sollte heute nicht programmieren, dann sollte es sich seinem Gefühl fügen. Dabei handelt es sich tatsächlich im Endeffekt um einen Produktivitätsgewinn wenn ein Entwickler bei schlechter Tagesverfassung nicht echten Code implementiert.

Sollte doch entwickelt werden und es werden Fehler erzeugt, dann sieht man diese unmittelbar am nächsten Tag, nämlich nachdem der Kunde die fehlerhafte Version der Software verwendet hat. Die Kernaussage ist, wenn man sich nicht im Stande fühlt 100 Prozentig konzentriert zu Programmieren, dann sollte man es bei täglichen Releases auch nicht tun.

Es gibt sicherlich genug andere Aufgaben die an solchen Tagen verrichtet werden kann und sei es nur jemanden anderen bei seiner Aufgabe zu unterstützen. Man kann sich metaphorisch dabei vorstellen, dass man bei täglichen Deployments das brüchige Eis unter seinen Zehenspitzen fühlt. Man darf nicht zu stark auftreten sonst bricht diese dünne Eisschicht sofort unter einem zusammen. Um all dies jedoch umsetzen zu können, muss es unmittelbar nach einem fehlerhaften Deployment möglich sein, automatisch eine frühere Version der Software herzustellen. Diese Art des Rollbacks nennt man Immunization.

Eine weitere Variante wie man mit fehlerhaften Releases umgehen kann ist, dass nach einem Deployment eine Art Monitor der Software automatisch das Team benachrichtigt, dass es ein Problem mit dem System gibt. Daraufhin stoppen alle Entwickler ihre aktuellen Arbeiten und beheben den Fehler worauf eine neue Version der Software deployt werden kann.

3.4.2 Data-Informed-Usability

Man trifft bei täglichen Releasese dieselben Usability-Entscheidungen aber diesmal kann man mit Hilfe von echten Daten entscheiden. A/B-Testing ist eine Art von diesem Vorgehen bei dem z.B. an einem Tag eine bestimmte Version der Software released wird und danach Daten eingeholt werden. Am nächsten Tag deployt man eine abgeänderte Version und analysiert wieder die entstandenen Informationen. Z.B. könnte man die Transaktionenanzahl messen oder wie lange Benutzer auf einer Seite geblieben sind. Man kann also die Daten die man generiert dazu verwenden, um Usability-Entscheidungen zu treffen. Somit lassen sich UI-Konflikte aufgrund von messbaren Daten vermeiden.

3.4.3 Feature-Flags

Die Idee dabei ist, dass man zur Laufzeit wenn man z.B. die Software auf 100 Server verteilt hat, und nebenbei hat man eine Konsole, die es jemanden erlaubt Features ein- und auszuschalten. D.h. man eine globale Anzahl von Flags, die es möglichen machen, gewisse Funktionalitäten für gewisse Benutzer freizuschalten bzw. zu deaktivieren.

Dabei kann man taktisch vorgehen wenn man z.B. ein neues Features implementiert jedoch zehn weitere neue Features ebenfalls mit deployt werden. Nun deaktiviert man sein Features mittels Konsole, wartet eine Zeit lang ab bis die anderen Features ihre Wirkung gezeigt haben und aktiviert das sein Feature wieder. Jetzt kann man den Output analysieren den das Feature verursacht hat. Man erkennt leicht, dass bei täglichen Deployments die Laufzeitkonfiguration der Software wirklichen Mehrwert einbringt.

Listing 1: Beispiel eines Feature Flags bei Flickr [6]

```
if ($cfg.enable_unicorn_polo) {
    // do something new and amazing here.
}
else {
    // do the current boring stuff.
}
```

3.4.4 One-Piece-Flow

Kanban ist derzeit in der Entwickler-Community sehr beliebt. Einer der Gründe ist, dass man den Arbeits-Flow des gesamten Teams auf einen Blick erkennen kann. Dabei limitiert man die Arbeit die in Bearbeitung ist und kennzeichnet dies anhand von Karteikarten auf einem sogenannten Kanban-Board. Dabei besteht dieses Board aus mehreren Spalten die eine gewisse Bedeutung haben wie z.B. die Aufgaben die gerade in Arbeit sind oder die, die gerade getestet werden.

Bei täglichen Deployments hat man allerdings nicht mehr die Zeit, Aufgaben von einer Spalte zur nächsten weiterzureichen. Entwickler treffen eine hohe Anzahl an verschiedenen Entscheidungen bei täglichen Deployments im Gegensatz zu längeren Release-Zyklen. Das liegt daran, dass die Entwickler wirklich schnelles Feedback benötigen um entsprechend schnell darauf reagieren zu können. Dabei nehmen sie viele verschiedene Rollen ein z.B. die des Designers oder des Implementierers oder die des Performance-Testers, etc. D.h. die einzelnen Stationen bei Kanban werden zu einem *One-Piece-Flow* wobei der Entwickler dafür die Verantwortung übernehmen muss, neue Funktionalitäten durch all diese Stationen zu führen um sie anschließend zu releasen.

3.4.5 Multi-Level-Staging

Bei täglichen Releases bleibt keine Zeit mehr, die Software durch verschiedene Stages zu führen. D.h. der Flow von einer Entwickler-Box über eine Integrations-Box wo einige Tests durchgeführt werden, über eine Testing-Box, über eine Pre-Production-Box und zu guter Letzt die Produktionsumgebung. Software durch all diese Stationen zu führen, benötigt zu viel Zeit wenn man täglichen deployn möchte.

D.h. man muss wie immer dieselben Problemen in kürzerer dadurch lösen, indem man Verantwortungen verändert damit man dasselbe Feedback bekommt nur basiert auf einer kürzeren Pipeline. D.h. ideal wären eine Development-Box und eine Produktionsumgebung.

3.4.6 Operations Department

Heutzutage gibt es noch sehr oft eine Trennung zwischen dem Operations- Department und dem Entwicklungs-Team in einem IT-Unternehmen. Dabei mussten sich Entwickler nicht um Infrastrukturangelegenheiten kümmern. Bei täglichen Releases lässt sich diese Aufgabe jedoch nicht mehr so einfach delegieren. Wieder einmal gibt es nicht genug Zeit für den Roundtrip zwischen Operations- Team und Entwickler-Team. Natürlich gibt es auch weiterhin Operations- Engineers deren Job es ist, das System zu monitoren und gewisse Prozesse zu automatisieren und Entwickler zu schulen um Frameworks und Tools zu erstellen, sodass jeder Verantwortung für Operations-Angelegenheiten übernehmen kann.

3.4.7 Keine Status Meetings

Obwohl es eine der wichtigen agilen Praktiken ist, bleibt einem Entwickler keine Zeit darauf zu warten, dass man von jemanden anderen am nächsten Tag im Stand-Up Meeting erfährt, wie weit er mit seiner Aufgabe ist. Waren sie bei monatlichen Releases eine tolle Sache, so sind sie bei täglichen Iterationen eher ein Hindernis.

Natürlich muss weiterhin kommuniziert werden, jedoch auf eine andere Art und Weise. Dabei müssen die Entwickler natürlich im selben Raum sitzen und mit der Tatsache klar kommen, dass man bei der Arbeit unterbrochen wird. Vielleicht hat man auch einen IRC-Channel, irgendeine Art von Social-Media wie eine Facebook-Group die einem eine Real-Time Übersicht über die Arbeit eines jeden einzelnen Entwicklers gibt zu jedem beliebigen Zeitpunkt gibt.

4 Success Stories / Case Studies

Im Folgenden sollen einige Firmen kurz vorgestellt werden, welche Continuous Deployment nicht nur aktiv sondern auch erfolgreich einsetzen. Dabei werden ebenfalls die entwickelten Produkte kurz vorgestellt und besonderes Augenmerk auf angewandte Methoden und Workflows gerichtet. Sofern Informationen darüber vorhanden sind welche unterstützenden Technologien verwendet wurden, werden diese ebenfalls erwähnt.

4.1 IMVU

Als erstes Beispiel dient IMVU[1], eine soziale Online Community, in der in einer virtuellen Realität mit Hilfe von 3D Avataren kommuniziert, Spiele gespielt und eigene Inhalte erschaffen und ausgetauscht werden können.

IMVU war eines der ersten *Lean Startup* Unternehmen welche Continuous Deployment aktiv einsetzten. Dabei ist dies aber nicht vorab im Ganzen geplant worden, sondern inkrementell entstanden [5]. Derzeit sind bei IMVU zirka 50 technische Mitarbeiter angestellt [3]. Ein wichtiger Punkt warum bei, vor allem so vielen Entwicklern, Continuous Deployment funktioniert, ist, dass es ein zentraler Bestandteil der Firmenkultur ist.
Als Vorteile von Continuous Deployment werden von IMVU die folgenden Punkte genannt [3]:

- Regression wird sehr rasch erkannt
- Fehler können schneller behoben werden, da zwischen dem einspielen eines Fehlers und der Meldung über ein Problem nicht viel Zeit vergeht
- Der Release einer neuen Version erzeugt keinen zusätzlichen Overhead
- Als Feedback bekommen sie sofort messbare Kerndaten von echten Kunden

Workflow

Eine wichtige Grundvoraussetzung für Continuous Deployment bei IMVU ist wie schon in Abschnitt 3.1.3 auf Seite 5 gezeigt: Continuous Integration. Als Technologie kommt hier Buildbot[2] zum Einsatz [8]. Um die Vorteile von Continuous Integration voll ausnützen zu können wird beim Entwickeln selbst *Commit Early Commit Often* praktiziert

[1] IMVU: `http://www.imvu.com/`
[2] Buildbot: `http://trac.buildbot.net/`

[5]. Ist ein Feature fertig entwickelt, oder ein Bug behoben worden, werden zuerst lokale Tests auf der Entwicklermaschine durchgeführt. Wenn all diese Tests positiv durchlaufen wurden, wird der neue Code in die Versionsverwaltung eingespielt [3].

Erst jetzt werden sämtliche Tests der Test-Suite angestoßen. Zurzeit sind dies zirka 15.000 Tests aus den Bereichen Unit-, Funktions- und Verhaltenstests [3]. Dabei werden die folgenden Technologien eingesetzt:

- Selenium Core wird mit einem eigens entwickelten API Wrapper für die Verhaltens-Tests eingesetzt
- YUI Test wird für Browser basierte JavaScript Unit-Tests verwendet
- PHP SimpleTest
- Erlang EUnit
- Python UnitTests

Schlägt nur einer der Tests fehl wird der zuletzt eingespielte Code zurückgesetzt. Es ist zu beachten, dass nicht nur die Masse an Tests, sprich die Testabdeckung, wichtig für IMVU ist, sondern auch die Qualität der Tests. Ein weiteres Merkmal ihrer Firmenkultur ist nämlich das Schreiben von qualitativ hochwertigen Tests [5]. Durch diese Maßnahmen, also dem Schreiben von gründlichen hochwertigen Tests, welche sich auf alle Aufgabenbereiche verteilen, schafft es IMVU ein separates Qualitätssicherungsteam überflüssig zu machen.

Nachdem sämtliche Tests erfolgreich durchgeführt wurden, wird ein eigen entwickeltes Build-Skript angestoßen um den neuen Code in die Produktionsumgebung einzuspielen. IMVUs Produktionsumgebung besteht aus einem Custer mit derzeit zirka 700 Servern [3]. Das Build-Skript verteilt zwar den Code im gesamten Cluster, umgestellt werden zunächst aber nur eine gewisse Prozent Anzahl an Servern. Die Umstellung auf den neuen Code erfolgt recht simpel per Symlink.

Durch ein ständig aktives Monitoring werden fortlaufend Messwerte über den Gesundheitszustand des Clusters gesammelt. Siehe dazu auch Abschnitt 3.4.1 auf Seite 15. Diese Messdaten beinhalten Werte für CUP-, Speicher- und Netzwerk-Last, aber auch Business Metriken kommen zum Einsatz [5], [8]. Findet nach einer gewissen Zeitspanne keine Regression des Clusters statt wird der neue Code auf allen Servern aktiv geschalten. Durch das begleitende Monitoring könnte so noch immer jederzeit auf die vorherige Version zurückgewechselt werden.

Kurz sei noch erwähnt wie bei IMVU mit den relationalen Datenbanken verfahren wird. Da ein Datenbank Schema Rollback nur schwer möglich ist, bzw. das Verändern des Schemas einen schwerwiegenden Eingriff darstellt, durchlaufen Schemamodifikationen, im Gegensatz zum Code Deployment, einen formalen Review Prozess. Müssen tatsächlich die Strukturen der Tabellen angepasst werden, bleiben die alten Tabellen weiterhin bestehen und es werden einfach neue Tabellen mit der neuen Struktur erstellt. Die Daten werden dann per *Copy on Read* bzw. per Hintergrund-Job migriert [3].

IMVU schafft nach dieser Methode derzeit zirka 50 Deployments pro Tag, was mitunter auch an der geringen Durchlaufzeit der Test Suite liegt – diese liegt unter 10 Minuten [5]. Auf einen wichtigen Vorteil für IMVU bezüglich Continuous Deployment zurückkommend fasst dies Timothy Fitz wie folgt zusammen:

> *„This is a software release process implementation of the classic Fail Fast pattern. The closer a failure is to the point where it was introduced, the more data you have to correct for that failure.“* [4]

4.2 Huitale

Huitale ist ein finnisches *Lean Startup* Unternehmen aus Helsinki welches Dienstleistungen im agilen Umfeld für Training und Consulting, aber auch Softwareentwicklung, anbietet. Sie können sich dabei auf ihr eigen entwickeltes Produkt *nextdoor.fi*[3] berufen. Nextdoor.fi ist eine online Plattform um Dienstleistungen im Haushaltsservicebereich anzubieten und einzukaufen. Die Plattform hat derzeit zirka 2.000 aktive Benutzer und im Monat ungefähr 30.000 Besucher [11].

Huitale hat während der Entwicklung von *nextdoor.fi* einen angepassten lean Software Entwicklungsprozess umgesetzt, welcher im Folgenden kurz vorgestellt wird.

Workflow

Der Softwareentwicklungsprozess von Huitale setzt viele Elemente von *Kanban* ein um ihren Prozess zu visualisieren und mess- und steuerbar zu machen. Näheres zu Kanban siehe Abschnitt 3.3 auf Seite 11.

Als Basis für neue Funktionalität dienen *Minimum Marketable Features*. Ein *Minimum Marketable Feature* ist im Gegensatz zu einem normalen Feature ein minimales Set an Funktionalität welches dem Kunden einen Mehrwert bietet. Dieser Mehrwert muss nicht unbedingt als reiner Gewinn gemessen werden, damit könnten zum Beispiel auch Kostenersparnisse oder erhöhte Kundenzufriedenheit gemeint sein [**norefyet**].

Sämtliche ermittelten *MMFs* landen nach Priorität sortiert in einer *Product Queue*. Diese Queue hat ein *Work in Progress* Limit von 7. Befinden sich nur noch 2 *MMFs* in der Queue können neue aufgenommen werden. Diese Queue orientiert sich also ganz stark an dem Puffer Konzept aus Kanban. Die Entwicklungsabteilung selbst hat ein *Work in Progress* Limit von 2 [11].

Ein *MMF* wird dann als fertig entwickelt angesehen, wenn es dafür eine ausreichende Anzahl qualitativ hochwertiger Unit- und Akzeptanztests gibt und diese in der Continuous Integration Umgebung fehlerfrei ausgeführt wurden. Weiters findet eine automatisierte Qualitätssicherung mittels statischer Codeanalyse (Checkstyle[4] und PMD[5]) statt [11].

[3] nextdoor.fi: `http://http://www.nextdoor.fi`
[4] Checkstyle: `http://checkstyle.sourceforge.net/`
[5] PMD: `http://pmd.sourceforge.net/`

Der letzte verpflichtende Schritt besteht aus einem *Peer Review*. Werden all diese Schritte erfolgreich durchlaufen gilt ein *MMF* als *done* und wird in 24h Zyklen deployed.
Das Continuous Deployment Konzept selbst ist sehr stark an das von IMVU angelehnt (siehe Abschnitt 4.1). Auch hier gibt es ein 24/7 Monitoring des Produktivsystems mit der Möglichkeit, die täglich statt findenden Backups, jederzeit automatisiert wiedereinzuspielen (Immune System mit automatischen Rollbacks 3.4.1).

Die Einteilung des Teams war ursprünglich stark an SCRUM mit einem *Single Product Owner* angelehnt. Dies wurde aber zu Gunsten einer zwei Team Strategie aufgegeben. Jetzt gibt es ein *Problem Team* und ein *Solution Team*. Dem *Problem Team* steht der CEO vor und besteht weiters aus dem CTO, Marketing & Sales und User Experience Experten. Der CTO steht zusätzlich dem *Solution Team* vor, welches auch die Entwickler beinhaltet [11].
Eigene Teams für das Testen oder den laufenden Betrieb gibt es nicht. Diese Aufgaben werden vom *Solution Team* mit übernommen. Bemerkenswert ist auch dass es bei Huitale keine Vollzeit angestellten Entwickler gibt [10].

Die Ergebnisse dieses Vorgehens sprechen für sich. Die *lead time*[6] neuer Features kann Aufgrund der ständig gemessenen und ausgewerteten Daten sehr genau angegeben werden. Im Durchschnitt beträgt diese derzeit 8 Tage, bei kleineren Features zirka 3 Tage. Das Entwickeln einer ersten *Public Beta* dauerte mit dieser Methode nur 120 Manntage [11].

Durch das Monitoring werden auftretende Bugs sehr rasch erkannt und können in der Regel innerhalb einer Stunde korrigiert werden. Möglich wurde all dies einerseits durch eiserne Disziplin der Mitarbeiter, als auch durch Erfahrung. Sämtliche Entwickler hatten bereits Erfahrung mit agilen Entwicklungsmethoden. Auch Kanban wird als wichtige Stütze im Finden von Verbesserungspotenzial des Prozesses angegeben. In vier Jahren gab es bei täglichen Deployments insgesamt nur 2 schwerere Bugs [11].

4.3 Digg

Digg[7] ist eine *Social News* Online-Plattform, dessen Grundfunktionalität das Bewerten von Artikeln durch die Benutzer ist. Dabei können die Artikel entweder positiv (*digging*) oder negativ (*burrying*) bewertet werden. Am 25. August 2010 wurde offiziell die Version 4 der Website frei gegeben [9], was zu reger Kritik seitens der Benutzer geführt hat, da die Website zu Beginn des Upgrades teilweise unerreichbar, bzw. instabil war. Weiters wurde durch das Upgrade ein neues Design eingeführt und bekannte Funktionen herausgenommen [7].

Aus Entwicklersicht ist das Upgrade auf Version 4 insofern interessant, da mit dieser Version auch ein Continuous Deployment Prozess realisiert wurde [1]. Seit August 2010 gibt es daher bei Digg keine geplanten Releases mehr. Alle Änderungen, also Bug-Fixes

[6] lead time: Die Zeit vom Erfassen eines Features bis zur Fertigstellung
[7] Digg: `http://www.digg.com`

und neue Features gehen sofort live. Auch bei Digg wird dabei auf manuelles Testen der Änderungen verzichtet.

Als große Herausforderung wird der Balanceakt zwischen der Agilität und Geschwindigkeit von Continuous Deployment und den Anforderungen an Stabilität und Verlässlichkeit genannt. Dessen ungeachtet gelten die geringe Durchlaufzeit und das unmittelbare Feedback bezüglich Änderungen bei Digg als immens wertvoll [1].

Workflow

Bei Digg wurde schon, bevor Continuous Deployment eingeführt wurde, ein Ökosystem aus Entwicklungstools und Techniken eingesetzt welche den Schritt zu Continuous Deployment erheblich erleichterten. Als Versionierungssystem wurde Git verwendet, die UI-Tests verwendeten Selenium und wurden per Continuous Integration mittels Hudson ausgeführt. Hudson wurde weiters für das Build-Management verwendet. Das Management und Monitoring der Infrastruktur wurde mittles *puppet* umgesetzt.

Als wichtigste Ergänzung dieses Ökosystems wird Gerrit[8] genannt [1]. Gerrit ist ein Code Review System für Git basierte Softwareprojekte. Dabei werden Änderungen, so genannte *patchsets*, nach einem Commit nicht sofort in den Master Branch der Versionsverwaltung eingespielt, sondern von Gerrit zum Review freigegeben. Erst nachdem diese Änderungen auch von anderen Entwicklern abgesegnet wurden, kann der neue Code in den Master Branch eingespielt werden.

Gerrit bietet zusätzlich für jedes *patchset* auch ein *Verified Flag*. Über dieses Flag lässt sich Gerrit sehr gut mit Continuous Integration Systemen wie Hudson verbinden, indem das Flag entsprechend dem Ausgang von automatisierten Tests gesetzt wird. Digg verwendet für die Integration von Gerrit und Hudson das *Gerrit Trigger* Plugin für Hudson.

Der generelle Ablauf bei Digg ist jetzt wie folgt: Nachdem ein neues *patchset* eingespielt wurde, wird dies zum Review freigegeben. Zusätzlich werden zunächst ein Subset an Unit- und UI-Tests ausgeführt (Vortests). Der Grund warum nicht gleich sämtliche Tests ausgeführt werden ist die Zeit die die komplette Testsuite zum Durchlauf benötigen würde. Falls die Änderungen durch das Review freigegeben und die Tests fehlerfrei durchlaufen wurden, findet ein Merge to Master statt, die Änderungen landen also im Hauptzweig der Versionsverwaltung. Erst jetzt werden sämtliche Unit-Tests ausgeführt und anschließend der neue Build in eine interne Staging Umgebung eingespielt. Von dort wird, nach erfolgreichem Absolvieren sämtlicher UI- Tests, in die Produktionsumgebung deployed.

Andrew Bayer, einem Entwickler bei Digg zufolge [1], ist dieser Prozess bei weitem nicht perfekt, so muss immer noch recht oft manuell eingegriffen werden nachdem ein Build bei den Vortests nicht durchgekommen ist. Als Grund wird hier die Instabilität einiger

[8] Gerrit: `http://code.google.com/p/gerrit/`

Tests genannt. Dennoch werden das Review-System und die Möglichkeit rasch Bug-Fixes einzuspielen positiv hervorgehoben.

4.4 Flickr

Flickr[9] ist eine Hosting- Plattform für Fotos und Videos. Dabei können Fotos und Videos hochgeladen, verwaltet und für andere zum Austausch freigegeben werden. Flickr bietet zusätzlich eine über Webservices realisierte API an. Die Firma wurde 2005 von *Yahoo!* übernommen und beschäftigte Mitte 2011 zirka 25 Angestellte [2].

Flickr setzt bei der Weiterentwicklung und Wartung ihrer Plattform einen über die Zeit inkrementell entstandenen Continuous Deployment Przess ein. Interessant an diesem Prozess ist das hohe Maß an Veranwortung welches jedem einzelnen Enwickler selbst zugetraut wird, denn jeder Entwickler ist bei Flickr gleichzeitig auch Release Manager. Über ein webbasiertes Deployment- Tool ist es allen Entwicklern möglich das Deployment eines neuen Builds zu starten. Über dieses Tool wird derzeit in der Regel 10 mal am Tag eine neue Version deployed. Es gibt aber auch Tage an denen keine neue Version deployed wird. Das Maximum wären zirka 50 Deployments am Tag [2].

Möglich wird dies durch eine Firmenphilosophie die eine professionelle Einstellung [2] zur Softwareentwicklung fördert. Da aber Menschen eben auch Fehler machen gibt es zusätzlich Maßnahmen um dem entgegenzuwirken. Wie schon in Abschnitt 3.4.1 auf Seite 15 vorgestellt, spielt Immunization dabei eine Wichtige Rolle. Daher wird beim Planen von neuen Features zum Beispiel schon eng mit dem Operations-Team zusammengearbeitet um die richtigen Metriken auszuwählen, die den ordnungsgemäßen Betrieb des Features sicherstellen können. Weiters gibt es die Möglichkeit das letzte Deployment per *1 Klick* Lösung wieder rückgängig zu machen.

Workflow

Bei der Entwicklung selbst werden keine *Branches* in der Versionsverwaltung (Flickr verwendet Subversion) angelegt. Das heißt alles, auch neue Features, landen sofort im Hauptzweig (HEAD). Dies ist für ein schnelles Bugfixing zwar von Vorteil, bei neuen Features, deren Entwicklung teilweise Monate in Anspruch nehmen, stellt dies jedoch ein Problem dar. Fertig entwickelte Features die noch nicht zum Release freigegeben sind würden so einfach angezeigt werden. Schlimmer noch wären halbfertige, die Website möglicherweise in negativer Art beeinträchtigende, Features die bereits einer breiten Öffentlichkeit zugänglich sind. Als Lösung kommen hier die in Abschnitt 3.4.3 auf Seite 16 vorgestellten Feature Flags zum Einsatz [6].

Weiters werden bei Flickr Feature Flags auch eingesetzt um Code gezielt für verschiedene Umgebungen (Entwicklung, Staging oder Produktion) freizuschalten. Die Einstellungen

[9] Flickr: `http://www.flickr.com`

welche Features aktiviert sind und welche nicht werden dabei recht einfach über Konfigurationsdateien realisiert. Der Vorteil daran ist, dass sobald ein Feature fertig entwickelt wurde, die Aktivierung auf sämtlichen Servern durch eine einzige Zeile in einer Konfigurationsdatei geschieht. Die Steuerung der Features ist aber nicht auf verschiedene Umgebungen begrenzt, sie können sogar für einzelne Benutzer freigeschalten werden. Der einzige Nachteil der Feature Flags ist, dass Code teilweise doppelt vorhanden ist und nach dem Launch eines neuen Features der alte Code manuell entfernt werden muss. Dies geschieht in der Regel zirka 3 Monate nach dem Launch [2].

Ein anderer interessanter Aspekt wie bei Flickr mit neuen Features umgegangen wird sind *Dark Launches* [2]. Dabei werden neue Features komplett vom User Interface bis zur Datenbank ausgeführt, dem Benutzer wird davon aber nichts angezeigt. So können die Auswirkungen die neue Features auf das Gesamtsystem haben vorab mit echtem Feedback getestet werden.

Wie schon erwähnt tragen die Entwickler bei Flickr sehr viel Verantwortung. Dies schlägt sich schon beim Testen nieder. So liegt die Hauptverantwortung für das Testen bei jedem Entwickler selbst [2]. Es gibt keine allumfassende Test Suite die automatisiert durch Continuous Integration sämtliche Bereiche der Anwendung gründlich testet. Ausführliche Unit- und Loadtests werden von den Entwicklern direkt in der Entwicklungsumgebung durchgeführt.

Es existiert zwar eine Stagingumgebung die nach jedem Commit aktualisiert wird, dort werden aber nur noch rudimentäre Tests durchgeführt. So zum Beispiel ob die verwendeten Datenbankreferenzen auch auf die Produktionsdatenbanken zeigen und nicht auf die Entwicklungsumgebung. Auch Performance Tests werden in dieser Stagingumgebung durchgeführt. Als Continuous Integration Tool kommt hier Hudson zum Einsatz und die automatisierten Tests benötigen in dieser Umgebung derzeit nur zirka 3 Minuten.

Über das zuvor kurz angesprochene Deployment Tool kann anschließend der in der Stagingumgebung befindliche Build deployed werden. Dabei werden dem Entwickler der die neue Version frei gibt noch einmal sämtliche Änderungen am Quellcode angezeigt, die er oder andere zwischenzeitlich vorgenommen haben. Damit nicht mehrere Entwickler gleichzeitig ein neues Release freigeben können wird das Deployment Tool für andere gesperrt sobald ein Entwickler anfängt damit zu arbeiten.

Auch diesem Prozess wird eingeräumt dass er nicht perfekt sei. So äußerte sich Ross Harmes darüber folgendermaßen:

> *„This style of development isn't all rainbows and sunshine.* (...) *But overall, we find it helps us develop new features faster and with fewer bugs.“* [6]

Michael Deerkoski zufolge [2] war es ihnen aber mit diesem Prozess möglich, einen Bug innerhalb von 35 Minuten ab der Meldung durch einen Benutzer zu beheben. *Zu beheben* bedeutet in diesem Zusammenhang natürlich dass der Bugfix bereits in der Produktionsumgebung eingespielt ist.

5 Schlussfolgerungen

Siehe Tabelle 1 auf Seite 26.

Kurze Zusammenfassung der „Ergebnisse“, bzw. Auflisten der Vorteile wenn man sich an die genannten Best Practices hällt. Dabei kann man sich auch stark an dem Abstract orientieren.

> *„Does this mean we're never going to introduce bugs to our live site? Of course not - but we're going to keep the number of bugs to hit the live site to a minimum, and we've made it easy and fast to get bug fixes live as well.“* [1]

Literatur

[1] Andrew Bayer. *Continuous Deployment, Code Review and Pre-Tested Commits on Digg4*. 22. Juli 2010. URL: http://about.digg.com/blog/continuous-deployment-code-review-and-pre-tested-commits-digg4/ (besucht am 18. 12. 2011).

[2] Michael Deerkoski. *Continuous Deployment at Flickr*. 1. Juni 2011. URL: http://sna-projects.com/blog/2011/06/continuous-deployment-flickr/ (besucht am 03. 01. 2012).

[3] Brett G. Durrett. *Scaling with Continuous Deployment*. 2010-06-29. URL: http://www.slideshare.net/bgdurrett/sds-2010-continuous-deployment-at-imvu (besucht am 17. 12. 2011).

[4] Timothy Fitz. *Continuous Deployment*. 2009-02-08. URL: http://timothyfitz.wordpress.com/2009/02/08/continuous-deployment/ (besucht am 18. 12. 2011).

[5] Timothy Fitz. *Continuous Deployment at IMVU: Doing the impossible fifty times a day*. 10. Feb. 2009. URL: http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/ (besucht am 18. 12. 2011).

[6] Ross Harmes. *Flipping Out*. 2. Dez. 2009. URL: http://code.flickr.com/blog/2009/12/02/flipping-out/ (besucht am 03. 01. 2012).

[7] Mathew Ingram. *Digg Redesign Met with a Thumbs Down*. 26. Aug. 2010. URL: http://gigaom.com/2010/08/26/digg-redesign-met-with-a-thumbs-down/ (besucht am 10. 01. 2012).

[8] Eric Ries. *Continuous deployment in 5 easy steps*. 30. März 2009. URL: http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html (besucht am 17. 12. 2011).

[9] Kevin Rose. *Digg Version 4*. 25. Aug. 2010. URL: http://about.digg.com/blog/digg-version-4/ (besucht am 10. 01. 2012).

Tabelle 1: Übersicht über die Best Practices

	Jahr auf Quartal	**Quartal auf Monat**	**Monat auf Woche**	**Woche auf Tag**
Organisatorische Aspekte		Programmierer schreiben Tests	Temporäre Branches	One-Piece-Flow
		Status Meetings	Kanban	kein Operations Department
		Task Board	Kein separates Test Team	keine Status Meetings
		keine QA-Abteilung	Keine Up-front Usability	
		kein Change Management		
Werkzeug- und Methodenunterstützung	Automatisierte Akzeptanztests	keine mehrfach releasten Versionen	Keystoning	Immunization
	Refactoring	keine separaten Design Dokumente	One-Button-Deploy	Data-Informed-Usability
	Continuous Integration	keine separaten Build- und Analyse-Teams	Kein aktiver Release-Branch	Feature-Flags
				kein Multi-Level-Staging
Geschäftliche Aspekte	Subscription Modell	Pay-per-Use Modell		

[10] Marko Taipale. *Continuous Deployment – Nextdoor.fi*. 3. März 2011. URL: http://huitale.blogspot.com/2011/03/presentation-continuous-deployment.html (besucht am 11. 12. 2011).

[11] Marko Taipale. „Huitale - A Story of a Finnish Lean Startup“. In: *Lean Enterprise Software and Systems*. Springer Berlin Heidelberg, 2010, S. 111–114.