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: 18. Dezember 2011

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 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.

2.1 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.

2.1.1 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.

2.1.2 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.

2.1.3 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 Zusam-

menbaus 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.

2.1.4 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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: Fertig ausformulieren
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.

2.3 Monat auf Woche

2.4 Woche auf Tag

3 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.

TODO: Referenzen zu den Quellen einbauen.

3.1 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. Derzeit sind bei IMVU zirka 50 technische Mitarbeiter angestellt. 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:

- 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 2.1.3 auf Seite 4 gezeigt: Continuous Integration. Als Technologie kommt hier Buildbot[2] zum Einsatz. Um die Vorteile von Continuous Integration voll ausnützen zu können wird beim Entwickeln selbst *Commit Early Commit Often* praktiziert. 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.

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

- Selenium Core wird mit einem eigens entwickelten API Wrapper für die Verhaltens-Tests eingesetzt

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

- 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. 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. 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. Diese Messdaten beinhalten Werte für CUP-, Speicher- und Netzwerk-Last, aber auch Business Metriken kommen zum Einsatz. 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.

Noch die Durchlaufzeit der Test Suite anmerken u. dass sie ca. 50 Deployments am Tag schaffen würden.

3.2 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.

[3] nextdoor.fi: http://http://www.nextdoor.fi

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.

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, messen und um diesen steuerbar zu machen. Näheres zu Kanban siehe Abschnitt 2.3 auf Seite 8.
Als Basis für neue Funktionalität dienen *Minimum Marketable Features* welche nach Priorität sortiert in einer *Product Queue* landen. 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.

Hier vielleicht noch erklären was genau ein Minimum Marketable Feature im Gegensatz zu einem *normalen* Feature ist. Zusätzlich kann noch gezeigt werden wie Huitale überhaupt zu den MMFs kommt (Brainstorming, Customer Development, etc.).

Ein *MMF* wird dann als fertig entwickelt angesehen, wenn es dafür eine ausreichende Anzahl qualitativ hochwertiger Unit- und Akzeptanz-Tests gibt und diese in der Continuous Integration Umgebung fehlerfrei ausgeführt wurden. Weiters findet eine automatisierte Qualitätssicherung mittels statischer Codeanalyse (Checkstyle[4] und PMD[5]). Als letzter Schritt findet ein *Peer Review* statt. 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 3.1). 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).

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.
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.

[4]Checkstyle: http://checkstyle.sourceforge.net/
[5]PMD: http://pmd.sourceforge.net/

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*, also des *Minimum Viable Products* dauerte nur 120 Mann-Tage.

TODO: *Minimum Viable Product* kurz erklären.

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.

3.3 Digg 4

3.4 WiredReach

3.5 Wealthfront

4 Schlussfolgerungen

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.

Literatur

[1] Puneet Agarwal. „Continuous SCRUM : Agile Management of SAAS Products“. In: *Proceedings of the 4th India Software Engineering Conference*. 2011, S. 51–60.

[2] Jasper Boeg. *Priming Kanban*. Trifork, 2011, S. 81.

[3] T Chow und D Cao. „A survey study of critical success factors in agile software projects“. In: *Journal of Systems and Software* 81.6 (Juni 2008), S. 961–971.

[4] Lisa Crispin und Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison Wesley Signature Series. Addison-Wesley, 2008, S. 576.

[6] lead time: Die Zeit vom Erfassen eines Features bis zur Fertigstellung

[5] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley Signature Series. Addison-Wesley Professional, 2007, S. 283.

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

[7] 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).

[8] Martin Fowler und Kent Beck. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.

[9] Markus Gärtner. *Software G Forces - The Effects of Acceleration*. 4. Nov. 2010. URL: http://www.shino.de/2010/11/04/software-g-forces-the-effects-of-acceleration/ (besucht am 18. 12. 2011).

[10] Jez Humble. *Continuous Delivery vs Continuous Deployment*. 13. Aug. 2010. URL: http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/ (besucht am 18. 12. 2011).

[11] Jez Humble, Chris Read und Dan North. „The Deployment Production Line“. In: *Proceedings of the conference on AGILE 2006*. IEEE Computer Society, 2006, S. 113–118.

[12] Jez Humble und MoleskyJoanne. „Why Enterprises Must Adopt Devops to Enable Continuous Delivery“. In: *Cutter IT Journal* 24.8 (2011), S. 6–12.

[13] J Humble und D Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison Wesley Signature Series. Addison-Wesley, 2010.

[14] Jingyue Li, Nils B Moe und Tore Dybå. „Transition from a Plan-Driven Process to Scrum – A Longitudinal Case Study on Software Quality“. In: *Science And Technology* (2010).

[15] Fergal Mccaffery u. a. „An Agile process model for product derivation in software product line engineering“. In: *Journal of Software Maintenance and Evolution: Research and Practice* (2010).

[16] Subhas Chandra Misra, Vinod Kumar und Uma Kumar. „Identifying some important success factors in adopting agile software development practices“. In: *Journal of Systems and Software* 82.11 (Nov. 2009), S. 1869–1890.

[17] Roman Pichler. *Scrum - Agiles Projektmanagement erfolgreich einsetzen*. dpunkt.-verlag, 2009, S. 184.

[18] Kenneth Pugh. *The Triad: A Tale of Lean-Agile Acceptance Test Driven Development*. Addison-Wesley, 2010, S. 345.

[19] W. W. Royce. „Managing the development of large software systems: concepts and techniques“. In: *Proceedings of the 9th international conference on Software Engineering*. ICSE '87. IEEE Computer Society Press, 1987, S. 328–338.

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

[21] Jiangping Wan. „Empirical Research on Critical Success Factors of Agile Software Process Improvement“. In: *Journal of Software Engineering and Applications* 03.12 (2010), S. 1131–1140.