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: 15. 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

Softwareunternehmen haben das Ziel Produkte zu schaffen, die ihre potentiellen Abnehmer bei ihrer Tätigkeit, sei es nun geschäftlich oder privat hinsichtlich einer Produktivitätssteigerung zu unterstützen. Dazu muss der Hersteller die genauen Arbeitsabläufe des Käufers genau ermitteln, analysieren und dann versuchen zu optimieren. Das funktioniert nur in engster Zusammenarbeit mit dem Kunden. Das Problem das bei der Softwareentwicklung allerdings heutzutage nach wie vor besteht, ist, dass der potentielle Abnehmer selbst oft nicht weiß, wie das Produkt aussehen soll beziehungsweise was es genau können muss.

Deshalb ist es für den Hersteller ungemein wichtig, dem Kunden so früh wie möglich eine funktionstüchtige erste Version der Software bereitzustellen. Dass diese natürlich nicht bereits die komplette Anzahl aller Features implementiert haben muss, ist an dieser Stelle klar. Jedoch ist es für den Kunden unheimlich hilfreich, wenn er zum Beispiel an einem Prototypen des Produkts gewisse Arbeitsabläufe ausprobieren kann. Dabei wird er sehr schnell feststellen ob ihm diese Version der Benutzeroberfläche gefällt oder nicht.

Das heißt, je früher der Käufer etwas Testbares zum Ausprobieren bekommt, desto besser für ihn als auch für den Softwarehersteller. Einerseits verfeinert der Kunde dadurch die Vorstellung darüber was er eigentlich möchte, andererseits können die Softwareentwickler ihren Entwicklungsprozess daran anpassen, wie sie am besten mit frühen Änderungswünschen umgehen können. Stellt sich nun die Frage, was genau die Probleme bei dem zuvor beschrieben Ablauf sein könnten.

Einerseits müssen die Erwartungen des Kunden bestmöglich durch das Softwareunternehmen zufriedengestellt werden. Eine Voraussetzung dafür ist so früh wie möglich im Produktlebenszyklus Feedback des Abnehmers einzuholen, um für diesen die optimalste, nämlich auf seine Bedürfnisse angepasste, Software zu produzieren. Das heißt die Geschwindigkeit in der das erste (und auch alle weiteren Releases des Produkts) bereitgestellt werden, ist von enormer Bedeutung.

Wie sieht es nun in der Praxis tatsächlich aus? In was für Abständen werden Softwareprodukte zum Beispiel von großen Softwareherstellen wie Microsoft, Apple etc. am Markt veröffentlicht. Kann diese Art von Software überhaupt mit Enterprise-Software verglichen werden? Wenn man aktuelle Bücher über Software- Engineering und Vorgehensmodelle in der Softwareentwicklung studiert, werden keine genauen Beschreibungen und Vorgaben zu Iterationslängen

gemacht. Hin und wieder liest man in aktuellerer Lektüre, dass Releasezyklen von zwei bis vier Wochen empfehlenswert wären, um sinnvolle, nämlich für den Kunden wertvolle, Funktionalität von Software bereitzustellen.

Bedeutet das nun, dass es in der Art und Weise wie Software heutzutage entwickelt wird, bei Iterationslängen nicht unterschieden werden muss? Kann ein Unternehmen, das einmal jährlich eine neue Version ihrer Software veröffentlicht rein theoretisch die Releasezyklen auf einen Monat verkürzen? Wenn man zwei ausgebildete Informatiker mit Hochschulabschluss darüber befragt, was denn genau die beste Methodik sei, um effektiv und effizient Software zu entwickeln, kann es sein, dass man zwei komplett verschiedene Antworten bekommt. Handelt es sich vielleicht dabei um die rapide fortschreitende Technologie die so viele unterschiedliche Vorgehensweisen zur Entwicklung von Software unterstützt? Warum sollte ein Unternehmen nur einmal im Jahr eine neue Version ihres Produkts am Markt bringen, wenn es möglich wäre, die Kunden jede Woche oder sogar jeden Tag mit einer Verbesserung der Software zu beglücken?

Tatsächlich handelt es sich bei der Softwareentwicklung um einen komplexen Prozess der bis heute noch immer auf unterschiedlichste Weise durchgeführt wird. Es wurde noch kein Vorgehensmodell erfunden oder keine Technologie erschaffen, die einen einheitlichen, formalen Entwicklungsprozess von Software ermöglicht. Entwickler müssen nicht nur die Kunst des Software-Engineerings beherrschen, was für sich schon jahrelange Erfahrung beansprucht, sondern auch im Team mit anderen Menschen zusammenarbeiten können, was ebenfalls eine große Herausforderung in vielen gängigen Vorgehensmodellen darstellt. Außerdem muss die Problemdomäne vom Entwickler genau verstanden werden, denn im Endeffekt ist das Produkt ein Hilfsmittel das den Arbeitsprozess des Kunden erleichtern soll.

Das kann natürlich nur umgesetzt werden, wenn der Softwarehersteller in enger Zusammenarbeit mit dem Softwareabnehmer in einer einheitlichen Sprache, nämlich die des Kunden, den Problembereich genau analysieren kann. Wenn man all diese Punkte im Vorgehensmodell des Softwareentwicklungsprozesses berücksichtigen möchte (es gibt natürlich noch viele weitere), fängt man an zu verstehen, warum heutzutage noch in so unterschiedlich langen Releasezyklen Software entwickelt wird. Prinzipiell gilt es für die Softwareentwickler die gleichen Probleme in unterschiedlichen Zeitspannen zu lösen. Dabei muss ein Produkt entstehen das den Ansprüchen des Kunden gerecht wird.

Generell gilt, hohe Qualität für wenig Geld in möglichst kurzer Zeit in Software umzuwandeln. Dabei versteht man unter Qualität die Umsetzung von funktionalen als auch nichtfunktionale Anforderungen des Käufers. Dazu gehören zum

Beispiel, dass das ausgelieferte Produkt so wenig Fehler wie möglich enthält, dass es performant ist, dass es ein ansprechendes Äußeres besitzt und natürlich muss es die gewünschten Features beinhalten. Dass natürlich nicht alle Aspekte (Qualität, Geld, Zeit) gleichermaßen gut optimiert werden können, scheint an dieser Stelle nun klar, sonst gäbe es ja nur Softwareunternehmen die stündlich neue Versionen ihrer Produkte kostenlos in höchster Qualität ausliefern würden. Softwareentwickler müssen also Kompromisse eingehen. Diese müssen jedoch so optimiert werden, dass der Kunde einerseits glücklich mit dem Produkt wird, der Hersteller andererseits Geld dabei verdienen kann.

Die folgende Arbeit befasst sich mit den Entwicklungstechniken, den Vorgehensweisen und eventuellen Geschäftsmodellen die für verschiedene Iterationslängen notwendig sind, um erfolgreich Produkte unter den gegebenen Voraussetzungen (Qualität, Geld und Zeit) zu realisieren. Dabei werden die Zeitspannen der einzelnen Phasen von einem Jahr, zu einem Realease innerhalb eines Quartals, zu einem Monat, einer Woche und schließlich zu einem Tag verkürzt. Dabei gilt es stets die Feedbackzyklen innerhalb dieser Iteration zu optimieren. Zum Beispiel kann das durch Automatisierungsprozesse in jeglicher Hinsicht geschehen. Es gilt dabei manuelle Arbeiten so weit wie möglich aus dem Entwicklungsprozess zu entfernen, die unmittelbare Kommunikation innerhalb des Teams zu erhöhen und eine wirkliche Vertrauensbasis zwischen Endkunden und Softwareentwickler zu schaffen. Hohe Produktqualität im Sinne von Fehlerfreiheit wir dabei durch umfangreiche Testvorgänge sichergestellt. Feedback und Kommunikation sind unter anderem die mitunter wichtigsten Eigenschaften eines effektiven und effizienten Vorgehensmodells.

2 Best-Practices

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 zum Beispiel der Analyse, dem Entwurf, der Realisierung (Implementierung) und dem Testen besteht [24]. 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 zum Beispiel 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 dreiMonate 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 [20]. 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 [11]. 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 zum Beispiel 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 Releasezyklus 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 [8]. 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 Endkundenund 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 Releasezyklen.

2.2 Quartal auf Monat

Bei jedem Übergang von einem längeren zu einem kürzeren Releasezyklus ist es notwendig, gewisse bis jetzt vielleicht erfolgreich eingeführte Praktiken zu entfernen und neue zu adaptieren. Diese Tatsache findet im Übrigen bei jeder weiteren Iterationsverkürzung ebenfalls statt.

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 Releasezyklus 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 [15].

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 zum Beispiel 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 [18].

2.2.3 Task Board

Klassische Planungsprozesse bei denen zu erledigende Aufgaben mehrere Stationen (zum Beispiel 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 [18]. 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 zum Beispiel 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. Zum Beispiel 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 Releasezyklen besitzt, für den die QA-Abteilung der erste Schritt aus dem Chaos war, das heißt, dass 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.

Das heißt, dass das Q/A-Department, das kritisch für den Erfolg für jährliche bzw. dreimonatige Releases war, wird zur unüberwindbaren Barriere in einmonatigen Prozessen und muss deshalb verworfen werden. Allerdings ist es nicht das Ziel die Verantwortlichkeiten dieser Abteilung aufzugeben. Sie müssen neu organisiert und verteilt werden. Eine Lösung wäre, dass Tester im Entwicklungsteam integriert werden. Wichtig dabei ist, dass sich alle Beteiligten im selben Raum befinden und ständig miteinander kommunizieren zu können.

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

2.2.7 Mehrfach releaste Versionen

Sofern ein Team unter keinem enormen Zeitdruck leidet und eventuell nicht Unmengen an Kunden besitzt, kann es durchaus möglich sein, mehrere unterschiedliche Versionen der Software zu verwalten. Dieser Aufwand ist jedoch auf keinen Fall zu unterschätzen und sollte daher gut dokumentiert sein.

Mit jeder zusätzlichen Version der Software steigt die Komplexität der Organisation des Konfigurationsmanagements [19]. Zum Beispiel könnte eine Organisation Software an einen bestimmten Kunden verkaufen. Angenommen dieser ist äußerst zufrieden mit dem Produkt. Nach einiger Zeit wird ein weiterer Abnehmer der Software gefunden, jedoch lässt dieser einige Veränderungen vornehmen. Somit müssen die Entwickler eine zusätzliche Version der Software für den neuen Kunden erstellen.

Natürlich versucht die Organisation die Änderungen auch dem ersten Kunden zu verkaufen, allerdings reicht diesem die ursprüngliche Version der Software. Angenommen nach einiger Zeit erhalten die Entwickler einen Bug-Report und müssen diesen natürlich in beiden Versionen des Produkts beheben. Ziel der Organisation ist natürlich die weitere Kundengewinnung. Jedoch mit jedem zusätzlichen Abnehmer der Software, der ebenfalls einige Änderungen vornehmen lassen möchte steigt der Verwaltungsaufwand. Waren Änderungswünsche des Kunden zu Beginn noch relativ rasch durchzuführen, wir es mit der Zeit immer schwieriger was jedoch dem Abnehmer nicht interessiert.

Aus eigener Erfahrung der Autoren kann es vorkommen, dass nicht einmal mehr die Entwickler selbst wissen, wie viele unterschiedliche Versionen der Software in Betrieb sind. Das führt dazu, dass Fehlerbehebungen bzw. Erweiterungen länger Zeit in Anspruch nehmen, als gäbe es nur eine einheitliche Version des Produkts. Selbst bei jährlichen Releases sollten Software-Unternehmen ältere Versionen ihrer Produkte bei Kunden aktualisieren, da der zusätzliche Verwaltungsaufwand zu kompliziert werden kann. Haben die Entwickler jedoch ein volles Jahr Zeit, ist es nicht kritisch und vielleicht wichtiger, die wenigen Kunden die man hat, mit individuellen Features zufrieden zu stellen.

Wenn das Unternehmen jedoch beschließt, in einem Jahr zwölf neue Releases bereitzustellen, kann diese Art von Konfigurationsmanagement nicht mehr funktionieren. Eine einzige Version des Produktes, die jeder Kunde bekommt, muss dabei das Ziel der Entwickler sein. Dafür muss die Aktualisierung der Software jedoch ohne zusätzliche Komplikationen für den Kunden stattfinden. Das fördert die Vertrauensbasis die zwischen dem Abnehmer des Produkts und dem Hersteller unbedingt vorhanden sein muss.

Bei dieser Veränderung handelt es sich also nicht nur um eine technische sondern auch um eine Beziehungsänderung zwischen dem Kunden und dem Unternehmen die die Software produziert. Der Kunde muss den Entwicklern vertrauen können. Die „CEO-zu-CEO“-Beziehung beider Unternehmen muss zu einer Partnerschaft verändert 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 Prozent sichergestellt sein, das kein Datenverlust möglich ist, keine Produktivität verloren geht, der Kunde nicht am Monatsanfang des neuen Jahres in die Firma kommt und eine komplette Zerstörung vorfindet. Der Kunde muss davon überzeugt sein, dass die Software solide ist.

Der Aufbau einer solchen Beziehung zwischen Kunden und dem Software-Unternehmen kann Jahre in Anspruch nehmen. Es handelt sich nicht um einen Prozess der über Nach passieren kann.

2.2.8 Design Dokumente

In traditionellen Vorgehensmodellen wir das das Design der Software sehr umfangreich dokumentiert. Prinzipiell ist es eine gute Idee, Softwarearchitektur-Skizzen zu erstellen um gewisse Aspekte der Struktur besser verstehen zu können. Dies kann jedoch in einer einmonatigen Iteration nicht im gleichen Umfang durchgeführt werden, wie für jährliche Releases. Das Design der Software muss jedoch weiterhin genau überlegt und geplant werden. Dabei sind jedoch Dokumente nicht das richtige Medium um Wissen zu transportieren.

Eventuell können hier Whiteboard-Skizzen erstellt und mittels Digitalkamera festgehalten werden. Die Bilder können dann in einem Team-Wiki verwaltet werden. In dem Buch [12] wird dies als das Visualisieren einer sogenannten *Broad-Brush* Architektur genannt. Dabei gilt als Daumenregel, dass das Design der Software in wenigen Minuten in Teamarbeit auf einem Whiteboard skizziert wird.

2.2.9 Anforderungsänderungen (Change Requests)

Analysiert man den Prozess wie in einem traditionellen Vorgehensmodell Änderungsanforderungen verarbeitet werden, sieht man schnell, dass es sich dabei um eine organisatorische Barriere handelt. Im folgenden Abschnitt wird ein solcher Ablauf kurz beschrieben. Zum Beispiel fordert ein Kunde eine Änderung der Software aufgrund eines Fehlers. Die verantwortliche Abteilung arbeitet diese Anfrage wird in irgendein entsprechendes System ein (zum Beispiel JIRA). Sofern es sich beim Kundenwusch um eine Erweiterung handelt, wird diese zum *Change Control Board* [19] weitergeleitet. Um effizient zu bleiben

wird die Anforderung in eine Verarbeitungswarteschlange eingereiht. Bei dem wöchentlichen Meeting des erwähnten Boards werden dann sämtliche Forderungen eventuell abgelehnt.

Aufgrund der Zeitbegrenzung von einem Monat funktioniert diese Art von Prozess nicht mehr. Jedoch handelt es sich bei einem Änderungsanforderungen um wertvolles Feedback. 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 Anforderungen bearbeiten zu können.

Zum Beispiel könnten die Entwickler in Zusammenarbeit mit dem Kunden die gewünschten Anforderungen priorisieren, wobei die entstandenen Prioritäten mit denen des Unternehmens harmonieren sollten. Klarerweise sollten die Entwickler an den wichtigen Features arbeiten und die weniger wichtigen hinten anstellen. Das heißt, man muss weiterhin Entscheidungen über die zu implementierenden Features treffen, jedoch hat die ursprüngliche Unternehmensstruktur die mittels Change Control Board funktioniert hat, innerhalb von einem Monat nicht genügend Zeit um Prioritäten richtig zu setzen.

2.2.10 Separate Konfigurationsmanagement- und Analyse-Teams

Diese Art von Separierung nimmt ebenfalls zu viel Zeit von einmonatigen Iterationen in Anspruch. Ein Team das sich ausschließlich um die Analyse kümmert, und danach ihr Ergebnis an die Entwickler weiterleitet, ist viel zu zweitaufwändig um innerhalb eines Monats ein neue Version der Software zu erstellen. Softwareentwickler müssen akzeptieren, dass das komplette Team die Verantwortung für das Konfigurationsmanagement und die Analyse übernehmen muss [19].

Es kann natürlich Personen in einem Team geben, die sich mit der Verfeinerung des Build-Prozesses beschäftigen, jedoch gibt es keine extra Abteilung die sich ausschließlich um diesen Teil der Software kümmert. In monatlichen Iterationen bleibt keine Zeit um Konfigurationsfehler bezüglich des Builds jemanden anderen zu zuteilen der sie dann darum kümmert. Benötigt man zum Beispiel eine Woche für das Beheben eines solchen Problems, dann ist es in einer zwölfwöchigen Phase schmerzhaft, in einer vierwöchigen Iteration jedoch erfolgsgefährdend.

Entwickler müssen dafür die Verantwortung übernehmen Konfigurationsprobleme zu beheben. Eine Rolle wie die des Build-Managers gibt es in einer monatlichen Iteration nicht mehr. Das heißt, dass sich die Organisation diesbezüglich verändern muss, um einen vierwöchigen Releasezyklus zu unterstützen.

2.3 Monat auf Woche

Es kann natürlich vorkommen, dass Kunden selbst für einen monatlichen Releasezyklus zu ungeduldig sind und deshalb auf wöchentliche Updates ihrer Software bestehen. Das heißt, das Entwicklerteam steht wieder vor der Aufgabe, dieselben Probleme in noch kürzerer Zeit zu bewältigen. Die erste Frage die sich bei der Umstellung auf wöchentliche Iterationen stellt, ist, an welchem Tag die neue Version bereitgestellt werden soll.

Zum Beispiel ist der letzte Tag der Woche oder ein Tag bevor Weihnachten, ein äußerst schlechter um neue Produktaktualisierungen dem Kunden zur Verfügung zu stellen. Die Mitte der Woche scheint eine gute Idee zu sein. Das heißt, 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 Lösungen der verschiedenen Probleme angepasst werden, um einwöchigen Iterationen gerecht zu werden. Das Entwicklerteam hat also fünf Tage Zeit, um ein für den Kunden wertvolles Inkrement zu erstellen. Wie sollte dabei vorgegangen werden? Ein erster Schritt könnte eine automatische Datenmigration sein.

2.3.1 Automatische Datenmigration

Bei einem einmonatigen Releasezyklus können ein paar Tage dafür verwendet werden, um eventuell am Datenbankschema Änderungen vorzunehmen. Keinem Entwickler macht diese Art von Arbeit Spaß, jedoch bleibt genügend Zeit um manuelle Anpassungen durchzuführen. Innerhalb von einer Woche geht sich diese Arbeit allerdings nicht aus. Manuelle Anpassungen um Datenmigrationen durchzuführen sind in Bezug auf Zeit einfach zu kostspielig. Eine voll automatisierte „Ein-Knopfdruck“-Datenmigration ist unabdingbar um Daten von einem alten in einen neuen Zustand überzuführen.

Falls man einen Service 24 Stunden, sieben Tage die Woche bereitstellen muss, dann müssen Softwareentwickler noch zusätzliche Dinge tun. Diese werden genauer bei täglichen Releases beschrieben. Hat man nur eine Woche Zeit um eine

neue Version der Software bereitzustellen, darf Datenmigration kein Problem mehr darstellen. Vor allem handelt es sich bei diesem Problem „nur“ um ein technisches Problem bei dem keine zwischenmenschlichen Konflikte entstehen sollten.

Da eine Vielzahl an Datenbankprodukten existieren, unterstützt eine Datenbank die automatisierte Datenmigrierung vielleicht besser als ein andere. Dafür ist Software-Engineering da, man nimmt die Zutaten die man hat und wendet sie passend am bestehenden Problem an.

2.3.2 Temporäre Verzweigungen (Branches)

In einem einmonatigen Zyklus können Entwickler Branches für Änderungen der Software anlegen und diese nicht unmittelbar wieder in den Haupt-Branch (Trunk) zurückintegrieren [19]. Das heißt die Programmierer arbeiten zum Beispiel über einen Zeitraum von einer Woche an einer Änderung in einem neu erstellten Branch und migrieren diesen dann nach Fertigstellung der Änderungen wieder zurück in den Trunk.

Dabei kann es zu komplizierteren Merges kommen, die einiges an Zeit beanspruchen können. Genau diese Zeit ist in einwöchigen Iterationen nicht mehr vorhanden. Daher sollten Entwickler maximal temporäre Branches anlegen dürfen, die nach ein paar Stunden Arbeit wieder in den Haupt-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 eventuelle Tatsache. Das heißt, Entwickler haben keine andere Wahl als kurzweilige, temporäre Branches anzulegen, da ansonsten einwöchige Releasezyklen keine Chance auf Erfolg hätten.

Eine wichtige Begabung die Entwickler bei immer kürzeren Iterationen haben sollten, ist die immer feinere Unterteilung von Arbeiten, sodass die Software nach einer Änderung trotzdem noch funktioniert. Zum Beispiel hat man ein Refactoring von zirka 4000 Lines of Code vor sich. Diese Änderungen müssen nicht alle auf einmal geschehen. Die Begabung eines Entwicklers 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 wählen.

2.3.3 Keystoning

Haben Entwickler das Problem, dass ein zu implementierendes Feature mehr als eine Woche Zeit in Anspruch nehmen würde, das heißt mehr als eine Iteration benötigen würde, könnten sie zuerst die notwendige Funktionalität erstellen, die der Endbenutzer nicht zu Gesicht bekommt.

Im nächsten Releasezyklus werden dann die notwendigen UI-Anpassungen durchgeführt und gewährleistet dadurch, die Einhaltung der fünftägigen Software-Aktualisierung. Das heißt, Entwickler sollten 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 Iterationen in Kauf nehmen.

2.3.4 Kanban

Die Vorteile von Kanban [3] kommen bei wöchentlichen Releasezyklen noch stärker zur Geltung. 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 Releasezyklus eine fixe Anzahl von Tasks die zu erledigen sind, das heißt einen fixen Scope, und dieser kann aufgrund von irgendwelchen Einflüssen nicht umgesetzt werden, dann kann das zu Folgefehlern in der Planung führen, sodass der Gesamtplan gefährdet ist.

Im Gegensatz dazu bietet Kanban die Möglichkeit mittels des Pull-Modells, dass ein Entwickler zum Beispiel 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 beziehungsweise keine entsprechenden Meetings einberufen werden.

2.3.5 One-Button-Deploy

Will ein Unternehmen wöchentliche Iterationen realisieren wird ein „Ein-Knopfdruck"-Deployment verpflichtend um den Erfolg des Projekts zu garantieren [14]. Selbst wenn Entwickler für ein Release einen vollen Monat Zeit haben, wird ein zweistündiger, manueller Deployment-Prozess, bei dem noch dazu hin und wieder Fehler auftreten können, zur Qual für denjenigen, der die Aktualisierung durchführen muss. Manuelle, fehleranfällige Aufgaben, die noch dazu zu viel Zeit in Anspruch nehmen, müssen prinzipiell in kurzen Iterationen vermieden werden.

Auch hier gilt wieder, dass diese Technik der vollständigen Automatisierung des Deployment-Prozesses auch für längere Iterationen eingesetzt werden kann. Jedoch ist dies bei dieser Geschwindigkeit nicht kritisch. Bei wöchentlichen Releases ist es auf jeden Fall kritisch und daher eine Grundvoraussetzung für den Erfolg.

2.3.6 Kein separates Test Team

In wöchentlichen Releasezyklen gib es keine Trennung zwischen einem Test- und einem Entwicklerteam. Selbst dieser geringe Abstand zwischen den beiden Teams wäre 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, dieser dann selbstständig das erwartete Verhalten der Software verifiziert um danach dem Entwickler Feedback zu geben damit dieser darauf entsprechend reagieren kann, muss angepasst werden.

Dieses Request-Response Verfahren ist zu zeitintensiv um in einer Woche eingesetzt zu werden. Das heißt, das gesamte Team ist dafür verantwortlich, dass sowohl neue Funktionalität bereitstellt, Feedback dafür zu erzeugt und entsprechend darauf reagiert wird [4].

2.3.7 Keine Up-front Usability

Jeder Spezialbereich der Softwareentwicklung hat das Verlangen vor allen anderen berücksichtigt zu werden. Zum Beispiel wird gerne die Softwarearchitektur als wichtigstes Kriterium für die erfolgreiche Erstellung des Produkts angesehen. Die Architektur sollte zu Beginn des Projekts entworfen werden, danach

kann die restliche, oft als nicht so wesentlich empfundene Arbeit umgesetzt werden.

Genau dieselben Ansichten haben Analysten, Designer, Tester, etc. In wöchentlichen Iterationen kann diese Kurzsichtigkeit nicht unterstützt werden. Das heißt, Entwickler können nicht die komplette Zeit einer einwöchigen Iteration auf das Architekturdokument warten. Wann sollen da noch die notwendigen Features umgesetzt werden? Daher muss die Aufgabe des Usability-Designs, anstatt nur zu Beginn des Zyklus, über die komplette Iteration verteilt umgesetzt werden.

2.3.8 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 um eventuell einen Software-Patch einzuarbeiten, ist es nicht zeitkritische diesen auch im Entwickler-Branch nachzuziehen.

Das heißt, in längeren Iterationen ist die Pflege von zwei separaten Branches noch ohne weiteres umsetzbar [19]. 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. Der Vorteil für Kunden dabei ist, sie müssen sich maximal eine Woche auf einen Bug-Fix gedulden.

2.4 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 ist.

2.4.1 Immunization

Bei täglichen Deployments ist diese bereitgestellte Funktionalität schon am nächsten Tag für den Kunden anwendbar. Natürlich stellt sich dabei die Frage wie dabei mit Fehlern umgegangen wird. Ist es überhaupt noch möglich bei

dieser Kürze der Iteration Fehler von Softwareentwicklern zu akzeptieren? Hat Teammitglied einmal einen schlechten Tag, das heißt, es fühlt sich nicht besonders, können dabei Fehler produziert werden, die zu fatalen Folgen führen.

Wenn ein Entwickler das Gefühl haben es sollte an diesem Tag lieber nicht zu programmieren, dann sollte er es auch wirklich nicht tun. Dabei handelt es sich im Endeffekt um einen Produktivitätsgewinn wenn bei schlechter Tagesverfassung kein Produktionscode implementiert wird.

Sollte doch trotz schlechter Tagesverfassung entwickelt werden und es werden Fehler erzeugt, dann sieht der Kunde diese unmittelbar am nächsten Tag, nämlich nachdem er 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 [22].

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.

2.4.2 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 zum Beispiel an einem Tag eine bestimmte Version der Software released wird und danach Daten eingeholt werden [17]. Am nächsten Tag deployt man eine abgeänderte Version und analysiert wieder die entstandenen Informationen. Zum Beispiel könnte man die Transaktionsanzahl 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-Diskussionen die auf rein subjektiven Tatsachen beruhen und eventuell zu Konflikten führen können, aufgrund von messbaren Daten vermeiden.

2.4.3 Feature-Flags

Die Idee dabei ist, dass man eine globale Anzahl von Flags konfiguriert hat, die es möglichen machen, gewisse Funktionalitäten für gewisse Benutzer freizuschalten bzw. zu deaktivieren [1]. Zum Beispiel kann ein Entwickler über eine Konsole neue Features für eine bestimmte Anzahl an Servern, auf denen die Software installiert ist, ein- beziehungsweise ausschalten. Dabei sind die restlichen Server von dieser Änderung nicht betroffen und funktionieren wie bisher.

Dabei kann natürlich taktisch vorgegangen werden. Möchte ein Entwickler die Auswirkungen seines implementierten Features auf die Software messen, jedoch weitere Features beim nächsten Release mitdeployt werden, kann das eigene Feature vorerst deaktiviert werden. Nachdem dann das Update mit all seinen Änderungen Wirkung gezeigt hat, kann der Entwickler das Feature aktivieren und damit beginnen die Daten zu analysieren die das Feature verursacht hat.

Man erkennt leicht, dass bei täglichen Aktualisierungen der Software die Laufzeitkonfiguration der Software wirklichen Mehrwert einbringt.

Listing 1: Beispiel eines Feature Flags bei Flickr [13]

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

2.4.4 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 (Work in Progress) 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 zum Beispiel die Aufgaben die gerade in analysiert werden oder die, die gerade getestet werden.

Bei täglichen Deployments hat man allerdings nicht genügend Zeit, Aufgaben von einer Spalte zur nächsten weiterzureichen. Entwickler treffen bei täglichen Deployments eine hohe Anzahl an verschiedenen Entscheidungen, im Gegensatz zu längeren Iterationen. Das liegt daran, dass die Entwickler unmittelbares Feedback benötigen um entsprechend schnell darauf reagieren zu können.

Dabei nehmen sie viele verschiedene Rollen ein. Zum Beispiel die Rolle des Designers oder des Programmierers oder die des Performance-Testers, etc. Das heißt, die einzelnen Stationen bei Kanban werden zu einem „One-Piece Flow“ wobei der Entwickler selbst dafür die Verantwortung übernehmen muss, neue Funktionalitäten durch all diese Stationen zu führen um sie anschließend freizugeben [25].

2.4.5 Multi-Level-Staging

Bei täglichen Releases bleibt keine Zeit mehr, die Software durch verschiedene Stages zu führen. Das heißt, 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, ist nicht mehr möglich. Software durch all diese Stationen zu führen benötigt zu viel Zeit wenn man täglich neue Versionen bereitstellen möchte.

Das heißt, 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. Ideal wären die Verwendung einer Development-Box und einer Produktionsumgebung.

2.4.6 Operations Department

Heutzutage gibt es noch sehr oft eine Trennung zwischen dem Operations-Department oder auch System-Administration genannt, und dem Entwicklungsteam in einem IT-Unternehmen. Dabei mussten sich Entwickler nicht um Infrastrukturangelegenheiten kümmern. Bei täglichen Zyklen lässt sich diese Aufgabe jedoch nicht mehr so einfach delegieren.

Bei täglichen Releases reicht die Zeit nicht mehr um den Umweg zwischen Operations-Team und Entwicklerteam zu unterstützen. Natürlich gibt es auch weiterhin Infrastrukturspezialisten deren Job es ist, das System zu überwachen, gewisse Prozesse zu automatisieren, Entwickler zu schulen und um Frameworks und Tools zu erstellen, sodass jeder Verantwortung für Operations-Angelegenheiten übernehmen kann [1].

2.4.7 Keine Status Meetings

Obwohl es eine der wichtigsten agilen Praktiken ist, bleibt einem Entwickler bei täglichen Releases keine Zeit einen ganzen Tag darauf zu warten, dass man über den aktuellen Arbeitsfortschritt eines jeden Teammitglieds informiert wird. Waren Stand-Up Meetings bei monatlichen Iterationen absolute Voraussetzung um Status-Updates zu bekommen, so sind sie bei täglichen Zyklen eher ein Hindernis.

Natürlich muss weiterhin intensiv innerhalb des Teams kommuniziert werden, jedoch auf eine andere Art und Weise. Grundvoraussetzung dabei ist, dass die Entwickler im selben Raum sitzen und mit der Tatsache klar kommen, dass man ständig bei der Arbeit unterbrochen werden kann. 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 zu jedem beliebigen Zeitpunkt gibt.

3 Success Stories

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.

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 [10]. Derzeit sind bei IMVU zirka 50 technische Mitarbeiter angestellt [7]. 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 [7]:

- 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 8 gezeigt: Continuous Integration. Als Technologie kommt hier Buildbot[2] zum Einsatz [21]. Um die Vorteile von Continuous Integration voll ausnützen zu können wird beim Entwickeln selbst *Commit Early Commit Often* praktiziert [10]. 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 [7].

Erst jetzt werden sämtliche Tests der Test-Suite angestoßen. Zurzeit sind dies zirka 15.000 Tests aus den Bereichen Unit-, Funktions- und Verhaltenstests [7]. 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 [10]. 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 Cluster mit derzeit zirka 700 Servern [7]. 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 2.4.1 auf Seite 20. Diese Messdaten beinhalten Werte für CPU-, Speicher- und Netzwerklast, aber auch Business Metriken kommen zum Einsatz [10], [21]. 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 [7].

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 [10]. 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.“* [9]

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. 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 [27].

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 2.3 auf Seite 16.

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 [6].

Sämtliche ermittelten *MMFs* landen nach Priorität sortiert in einer *Product Queue*. Diese Queue hat ein *Work in Progress* Limit von sieben. Befinden sich nur noch zwei *MMFs* in der Queue können neue aufgenommen werden. Diese Queue orientiert sich also stark an dem Puffer Konzept aus Kanban. Die Entwicklungsabteilung selbst hat ein *Work in Progress* Limit von zwei [27].

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

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 [27]. 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 3.1). Auch hier gibt es ein 24/7 Monitoring des Produktivsystems mit der Möglichkeit, die täglich stattfindenden Backups, jederzeit automatisiert wiedereinzuspielen (Immune System mit automatischen Rollbacks 2.4.1).

Die Organisation 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 [27].
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 [26].

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 [27].

Durch das Monitoring werden auftretende Bugs sehr rasch erkannt und können in der Regel innerhalb einer Stunde korrigiert werden. Möglich wurde all dies durch disziplinierte und erfahrene Mitarbeiter. 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 zwei schwerere Bugs [27].

[4] Checkstyle: `http://checkstyle.sourceforge.net/`
[5] PMD: `http://pmd.sourceforge.net/`
[6] lead time: Die Zeit vom Erfassen eines Features bis zur Fertigstellung

3.3 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 [23], 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 [16].

Aus Entwicklersicht ist das Upgrade auf Version 4 insofern interessant, da mit dieser Version auch ein Continuous Deployment Prozess realisiert wurde [2]. Seit August 2010 gibt es daher bei Digg keine geplanten Releases mehr. Alle Änderungen, also Bug-Fixes 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 [2].

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 [2]. 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.

[7] Digg: `http://www.digg.com`
[8] Gerrit: `http://code.google.com/p/gerrit/`

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 Dauer 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 [2], 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 Tests genannt. Dennoch werden das Review-System und die Möglichkeit rasch Bug-Fixes einzuspielen positiv hervorgehoben.

3.4 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 [5].

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

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

deployed. Es gibt aber auch Tage an denen keine neue Version deployed wird. Das Maximum wären zirka 50 Deployments am Tag [5].

Möglich wird dies durch eine Firmenphilosophie die eine professionelle Einstellung zur Softwareentwicklung fördert [5]. Da aber Menschen auch Fehler machen gibt es zusätzliche Maßnahmen um dem entgegenzuwirken. Wie schon in Abschnitt 2.4.1 auf Seite 20 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 2.4.3 auf Seite 22 vorgestellten Feature Flags zum Einsatz [13].

Weiters werden bei Flickr Feature Flags eingesetzt um Code gezielt für verschiedene Umgebungen (Entwicklung, Staging oder Produktion) freizuschalten. Die Einstellungen 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 möglich ist. 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 [5].

Ein anderer interessanter Aspekt wie bei Flickr mit neuen Features umgegangen wird sind *Dark Launches* [5]. 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 [5]. 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.“* [13]

Michael Deerkoski zufolge [5] war es ihnen 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.

4 Schlussfolgerungen

In dieser Arbeit haben wir versucht Techniken und Methoden aufzuzeigen welche es ermöglichen Entwicklungszyklen drastisch zu verkürzen. Ausgehend von einem jährlichen Entwicklungszyklus wurden schrittweise Maßnahmen vorgestellt um die Kadenz der Releases zunächst auf vierteljährlich, monatlich, wöchentlich und zu guter Letzt auf täglich umzustellen. Dabei wurden nicht immer nur neue Praktiken hinzugefügt sondern, sofern erforderlich auch bestehende als schlecht aufgezeigt und für obsolet ja sogar hinderlich (Ballast) erklärt. Eine Übersicht aller erwähnten Techniken und Methoden ist in Tabelle 1 auf Seite 33 aufgelistet. Dabei wurden die einzelnen Praktiken nach den folgenden Kategorien zusammengefasst:

- Organisatorische Aspekte
- Werkzeug- und Methodenunterstützung
- Geschäftliche Aspekte

Tabelle 1: Übersicht über die Best Practices

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

In weiterer Folge wurden *Success Stories* vorgestellt welche viele bzw. Teile der in dieser Arbeit dargelegten Praktiken einsetzen, und damit erfolgreich die Releasezyklen ihrer Software stark verkürzt haben. Die Arbeitsabläufe der in den *Success Stories* beschriebenen Firmen sind dabei nicht selten inkrementell entstanden.

Zusammenfassend konnten laut den *Success Stories* die folgenden Vorteile gewonnen werden, wenn man es schafft seine Releasezyklen drastisch, bis hin zum *Continuous Deployment*, zu verkürzen:

- Fehler können rasch behoben werden da zwischen dem einspielen eines Fehlers und der Meldung über ein Problem nicht viel Zeit vergeht
- Regression wird sehr rasch erkannt
- Der Release einer neuen Version erzeugt keinen zusätzlichen Overhead (Releases werden sozusagen Non-Events)
- Unmittelbares Feedback
- Feedback besteht aus sofort messbaren Kerndaten von echten Kunden

Dass die vorgestellten Arbeitsweisen nicht perfekt sind wird von den Betroffenen offen zugegeben. Die dabei erworbenen Vorteile überwiegen aber die noch bestehenden Probleme. Siehe dazu auch das folgende Zitat von Andrew Bayer (Flickr):

> *„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.“* [2]

Literatur

[1] John Allspaw und Jesse Robbins. *Web Operations: Keeping the Data On Time*. O'Reilly Media, 2010.

[2] 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).

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

[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.

[5] 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).

[6] Mark Denne und Jane Cleland-Huang. *Software by Numbers: Low-Risk, High-Return Development*. Prentice Hall, 2003.

[7] 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).

[8] 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.

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

[10] 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).

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

[12] Steve Freeman und Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison Wesley Signature Series. Addison-Wesley Professional, 2009.

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

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

[15] Andrew Hunt und Dave Thomas. *Pragmatic Unit Testing in Java with JUnit The Pragmatic Starter Kit - Volume 2*. The Pragmatic Bookshelf, 2003.

[16] 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).

[17] Avinash Kaushik. *Web Analytics 2.0: The Art of Online Accountability and Science of Customer Centricity*. Sybex, 2009.

[18] Roman Pichler. *Scrum Agiles Projektmanagement erfolgreich einsetzen*. dpunkt.verlag, 2008.

[19] Gunther Popp. *Konfigurationsmanagement mit Subversion, Ant und Maven Grundlagen für Softwarearchitekten und Entwickler*. dpunkt.verlag, 2008.

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

[21] 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).

[22] Eric Ries. *Five Whys*. 13. Nov. 2008. URL: http://www.startuplessonslearned.com/2008/11/five-whys.html (besucht am 15. 01. 2012).

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

[24] 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.

[25] Jeff Sutherland. *An Alternative to Kanban: One-Piece Continuous Flow*. 2011-11-20. URL: http://scrum.jeffsutherland.com/2011/11/alternative-to-kanban-one-piece.html (besucht am 15. 01. 2012).

[26] 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).

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